Vulkan 1.0 - A Specification

Contents

Chapter 1

Introduction

This chapter is Informative except for the sections on Terminology and Normative References.

This document, referred to as the "Vulkan Specification" or just the "Specification" hereafter, describes the Vulkan graphics system: what it is, how it acts, and what is required to implement it. We assume that the reader has at least a rudimentary understanding of computer graphics. This means familiarity with the essentials of computer graphics algorithms and terminology as well as with modern GPUs (Graphic Processing Units).

The canonical version of the Specification is available in the official Vulkan Registry, located at URL

http://www.khronos.org/registry/vulkan/

1.1 What is the Vulkan Graphics System?

Vulkan is an API (Application Programming Interface) for graphics and compute hardware. The API consists of many commands that allow a programmer to specify shader programs, compute kernels, objects, and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

1.1.1 The Programmer's View of Vulkan

To the programmer, Vulkan is a set of commands that allow the specification of *shader programs* or *shaders*, *kernels*, data used by kernels or shaders, and state controlling aspects of Vulkan outside the scope of shaders. Typically, the data represents geometry in two or three dimensions and texture images, while the shaders and kernels control the processing of the data, rasterization of the geometry, and the lighting and shading of *fragments* generated by rasterization, resulting in the rendering of geometry into the framebuffer.

A typical Vulkan program begins with platform-specific calls to open a window or otherwise prepare a display device onto which the program will draw. Then, calls are made to open *queues* to which *command buffers* are submitted. The command buffers contain lists of commands which will be executed by the underlying hardware. The application can also allocate device memory, associate *resources* with memory and refer to these resources from within command buffers. Drawing commands cause application-defined shader programs to be invoked, which can then consume the data in the resources and use them to produce graphical images. To display the resulting images, further platform-specific commands are made to transfer the resulting image to a display device or window.

1.1.2 The Implementor's View of Vulkan

To the implementor, Vulkan is a set of commands that allow the construction and submission of command buffers to a device. Modern devices accelerate virtually all Vulkan operations, storing data and framebuffer images in high-speed memory and executing shaders in dedicated GPU processing resources.

The implementor's task is to provide a software library on the host which implements the Vulkan API, while mapping the work for each Vulkan command to the graphics hardware as appropriate for the capabilities of the device.

1.1.3 Our View of Vulkan

We view Vulkan as a pipeline having some programmable stages and some state-driven fixed-function stages that are invoked by a set of specific drawing operations. We expect this model to result in a specification that satisfies the needs of both programmers and implementors. It does not, however, necessarily provide a model for implementation. An implementation must produce results conforming to those produced by the specified methods, but may carry out particular computations in ways that are more efficient than the one specified.

1.2 Filing Bug Reports

Issues with and bug reports on the Vulkan Specification and the API Registry can be filed in the Khronos Vulkan Github repository, located at URL

http://github.com/KhronosGroup/Vulkan

Please tag issues with appropriate labels, such as *Specification*, *Ref Pages* or *Registry*, to help us triage and assign them appropriately.

1.3 Terminology

The key words **must**, **must not**, **required**, **shall**, **shall not**, **should**, **should not**, **recommend**, **may**, and **optional** in this document are to be interpreted as described in RFC 2119:

http://www.ietf.org/rfc/rfc2119.txt

must
This word, or the terms **required** or **shall**, mean that the definition is an absolute requirement of the specification.

must not
This phrase, or the phrase **shall not**, means that the definition is an absolute prohibition of the specification.

should
This word, or the adjective **recommended**, means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

should not
This phrase, or the phrase **not recommended**, means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

may
: This word, or the adjective **optional**, means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option must be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option must be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides).

The additional terms **can** and **cannot** are to be interpreted as follows:

can
: This word means that the particular behavior described is a valid choice for an application, and is never used to refer to implementation behavior.

cannot
: This word means that the particular behavior described is not achievable by an application. For example, an entry point does not exist, or shader code is not capable of expressing an operation.

Note
There is an important distinction between **cannot** and **must not**, as used in this Specification. **Cannot** means something the application literally is unable to express or accomplish through the API, while **must not** means something that the application is capable of expressing through the API, but that the consequences of doing so are undefined and potentially unrecoverable for the implementation.

1.4 Normative References

Normative references are references to external documents or resources to which implementers of Vulkan must comply.

IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008, http://dx.doi.org/10.1109/IEEESTD.2008.4610935, August, 2008.

A. Garrard, *Khronos Data Format Specification, version 1.1*, https://www.khronos.org/registry/dataformat/specs/1.1/dataformat.1.1.html, Khronos Ratified Specification, December ??, 2015.

J. Kessenich, *SPIR-V Extended Instructions for GLSL, Version 1.00*, https://www.khronos.org/registry/spir-v/, December ??, 2015.

J. Kessenich and B. Ouriel, *The Khronos SPIR-V Specification, Version 1.00*, https://www.khronos.org/registry/spir-v/, December ??, 2015.

Chapter 2

Fundamentals

This chapter introduces fundamental concepts including the Vulkan execution model, API syntax, queues, pipeline configurations, numeric representation, state and state queries, and the different types of objects and shaders. It provides a framework for interpreting more specific descriptions of commands and behavior in the remainder of the Specification.

2.1 Execution Model

This section outlines the execution model of a Vulkan system.

Vulkan exposes one or more *devices*, each of which exposes one or more *queues* which may process work asynchronously to one another. The queues supported by a device are divided into *families*, each of which supports one or more types of functionality and may contain multiple queues with similar characteristics. Queues within a single family are considered *compatible* with one another, and work produced for a family of queues can be executed on any queue within that family. This specification defines four types of functionality that queues may support: graphics, compute, transfer, and sparse memory management.

Note
It is possible that a single device may report multiple similar queue families rather than, or as well as reporting multiple members of one or more of those families. This indicates that while members of those families have similar capabilities, they are *not* directly compatible with one another.

Device memory is explicitly managed by the application. Each device may advertise one or more heaps, representing different areas of memory. Memory heaps are either device local or host local, but are always visible to the device. Further detail about memory heaps is exposed via memory types available on that heap. Examples of memory areas that might be available on an implementation include *device local* (memory that is physically connected to the device), *device local, host visible* (device local memory that is visible to the host) and *host local, host visible* (memory that is local to the host and visible to the device and host). On other architectures, there may only be a single heap that can be used for any purpose.

A Vulkan application controls a set of devices through the submission of command buffers which have recorded device commands issued via Vulkan library calls. The content of command buffers is specific to the underlying hardware and is opaque to the application. Once constructed, a command buffer can be submitted once or many times to a queue for execution. Multiple command buffers can be built in parallel by employing multiple threads within the application.

Command buffers submitted to different queues may execute in parallel or even out of order with respect to one another. Command buffers submitted to a single queue respect the submission order, as described further in Queue Operation. Command buffer execution by the device is also asynchronous to host execution. Once a command buffer is submitted to a queue, control may return to the application immediately. Synchronization between the device and host, and between different queues is the responsibility of the application.

2.1.1 Queue Operation

Vulkan queues provide an interface to the execution engines of a device. Commands are recorded into command buffers ahead of execution time. These command buffers are then submitted to queues for execution. Command buffers submitted to a single queue play back the commands in the order they were recorded, both within and across command buffer boundaries. Work performed by those commands respects the ordering guarantees provided by explicit and implicit dependencies, as described below. Work submitted to separate queues may execute in any relative order unless otherwise specified. Therefore, the application must explicitly synchronize work between queues when needed.

In order to control relative order of execution of work both within a queue and across multiple queues, Vulkan provides several synchronization primitives, which include *semaphores*, *events*, *pipeline barriers*, and *fences*. These are covered in depth in Synchronization and Cache Control. In broad terms, semaphores are used to synchronize work across queues or across coarse-grained submissions to a single queue, events and barriers are used to synchronize work within a command buffer or sequence of command buffers submitted to a single queue, and fences are used to synchronize work between the device and the host.

Note
Implementations have significant freedom to overlap execution of work submitted to a queue, and this is common due to deep pipelining and parallelism in Vulkan devices.

Work is submitted to queues using queue submission commands that typically take the form **`vkQueue*`** (e.g. `vkQueueSubmit`, `vkQueueBindSparse`), and usually take a list of semaphores upon which to wait before work begins and a list of semaphores to signal once work has completed. Unless otherwise ordered by semaphores, command buffer execution from multiple queue submissions done using the `vkQueueSubmit` command may overlap (but not be reordered), sparse binding operations done using the `vkQueueBindSparse` command from multiple batches may overlap or be reordered, and command buffer submissions and sparse binding operations may overlap or be reordered against operations of the other type.

Command buffer boundaries, both between primary command buffers of the same or different batches or submissions as well as between primary and secondary command buffers, do not introduce any implicit ordering constraints. In other words, submitting the set of command buffers (which can include executing secondary command buffers) between any semaphore or fence operations plays back the recorded commands as if they had all been recorded into a single primary command buffer, except that the current state is reset on each boundary.

Commands recorded in command buffers either perform actions (draw, dispatch, clear, copy, query/timestamp operations, begin/end subpass operations), set state (bind pipelines, descriptor sets, and buffers, set dynamic state, push constants, set render pass/subpass state), or perform synchronization (set/wait events, pipeline barrier, render pass/subpass dependencies). Some commands perform more than one of these tasks. State setting commands update the "current state" of the command buffer. Some commands that perform actions (e.g. draw/dispatch) do so based on the current state set cumulatively since the start of the command buffer. The work involved in performing action commands is often allowed to overlap or to be reordered, but doing so must not alter the state to be used by each action command. In general, action commands are those commands that alter framebuffer attachments, read/write buffer or image memory, or write to query pools.

Synchronization commands introduce explicit execution and memory dependencies between two sets of action commands, where the second set of commands depends on the first set of commands. These dependencies enforce that both the execution of certain pipeline stages in the later set occur after the execution of certain stages in the source set, and that the effects of memory accesses performed by certain pipeline stages occur in order and are visible to each other. When not enforced by an explicit dependency or otherwise forbidden by the specification, action commands may overlap execution or execute out of order, and may not see the side effects of each other's memory accesses.

Submitting command buffers and sparse memory operations, signaling fences, and signaling and waiting on semaphores each provide Implicit Ordering Guarantees. Signaling a fence or semaphore each guarantees that the previous commands have completed execution and that memory writes from those commands are available to future commands. Waiting on a semaphore or submitting command buffers after a fence has been signaled each guarantees that previous writes that were available are also visible to subsequent commands.

Within a subpass of a render pass instance, for a given (x,y,layer,sample) sample location, the following stages are guaranteed to execute in *API order* for each separate primitive that includes that sample location:

- depth bounds test
- stencil test, stencil op and stencil write
- depth test and depth write
- occlusion queries
- blending, logic op and color write

where the API order sorts primitives:

- First, by the action command that generates them.
- Second, by the order they are processed by primitive assembly.

Within this order, implementations also sort primitives:

- Third, by an implementation-dependent ordering of new primitives generated by tessellation, if a tessellation shader is active.
- Fourth, by the order new primitives are generated by geometry shading, if geometry shading is active.
- Fifth, by an implementation-dependent ordering of primitives generated due to the polygon mode.

The device executes command buffers from queues asynchronously from the host. Control is returned to an application immediately following command buffer submission to a queue. The application must synchronize work between the host and device as needed.

As part of each submission to a queue, a list of semaphores upon which to wait, and a list of semaphores to signal is provided along with the list of command buffers to execute. This is covered in more detail in Section 5.4.

2.2 Object Model

The devices, queues, and other entities in Vulkan are represented by Vulkan objects. At the API level, all objects are referred to by handles. There are two classes of handles, dispatchable and non-dispatchable. *Dispatchable* handle types are a pointer to an opaque type. This pointer may be used by layers as part of intercepting API commands, and thus each API command takes a dispatchable type as its first parameter. Each object of a dispatchable type has a unique handle value.

Non-dispatchable handle types are a 64-bit integer type whose meaning is implementation-dependent, and may encode object information directly in the handle rather than pointing to a software structure. Objects of a non-dispatchable type may not have unique handle values within a type or across types. If handle values are not unique, then destroying one such handle must not cause identical handles of other types to become invalid, and must not cause identical handles of the same type to become invalid if that handle value has been created more times than it has been destroyed.

All objects created or allocated from a VkDevice (i.e. with a VkDevice as the first parameter) are private to that device, and must not be used on other devices.

2.2.1 Object Lifetime

Objects are created or allocated by *Create* and *Allocate* commands, respectively. Once an object is created or allocated, its "structure" is considered to be immutable, though the contents of certain object types is still free to change. Objects are destroyed or freed by *Destroy* and *Free* commands, respectively.

Objects that are allocated (rather than created) take resources from an existing pool object or memory heap, and when freed return resources to that pool or heap. While object creation and destruction are generally expected to be low-frequency occurences during runtime, allocating and freeing objects can occur at high frequency. Pool objects help accommodate improved performance of the allocations and frees.

It is an application's responsibility to track the lifetime of Vulkan objects, and not to destroy them while they are still in use.

Application-owned memory is immediately consumed by any Vulkan command it is passed into. The application can alter or free this memory as soon as the commands that consume it have returned.

The following object types are consumed when they are passed into a Vulkan command and not further referenced by the objects they are used to create. They can be freed at any time they are not in use by an API command:

- VkShaderModule
- VkPipelineCache
- VkPipelineLayout

VkDescriptorSetLayout objects may be referenced by descriptor sets allocated using that layout, and those descriptor sets must not be updated with `vkUpdateDescriptorSets` after the descriptor set layout has been destroyed. Otherwise, descriptor set layouts can be freed any time they are not in use by an API command.

The application must not destroy any other type of Vulkan object until any uses of that object by the device (such as via command buffer execution) have completed.

The following Vulkan objects can be destroyed when no command buffers using the object are executing:

- VkEvent
- VkQueryPool
- VkBuffer
- VkBufferView
- VkImage

- VkImageView
- VkPipeline
- VkSampler
- VkDescriptorPool
- VkFramebuffer
- VkRenderPass
- VkCommandPool
- VkDeviceMemory
- VkDescriptorSet

The following Vulkan objects can be destroyed when work on the queue that uses the object has been completed:

- VkFence
- VkSemaphore
- VkCommandBuffer
- VkCommandPool

In general, unrelated objects can be destroyed or freed in any order, even if one object is referenced by another (e.g. use of a resource in a view, use of a view in a descriptor set, use of an object in a command buffer, binding of a memory allocation to a resource), as long as an object that references a freed object is not further used in any way except to be destroyed or to be reset in such a way that it no longer references the other object (such as resetting a command buffer). If the object has been reset, then it can be used as if it never referenced the freed object. An exception to this is when there is a parent/child relationship between objects. In this case, the application must not destroy a parent object before its children, except when the parent is explicitly defined to free its children when it is destroyed (i.e. for pool objects, as defined below).

VkCommandPool objects are parents of VkCommandBuffer objects. VkDescriptorPool objects are parents of VkDescriptorSet objects. VkDevice objects are parents of many object types (all that take a VkDevice as a parameter to their creation).

The following Vulkan objects have specific restrictions for when they can be destroyed:

- VkQueue objects cannot be explicitly destroyed. Instead, they are implicitly destroyed when the VkDevice object they are retrieved from is destroyed.
- Destroying a pool object implicitly frees all objects allocated from that pool. Specifically, destroying VkCommandPool frees all VkCommandBuffer objects that were allocated from it, and destroying VkDescriptorPool frees all VkDescriptorSet objects that were allocated from it.
- VkDevice objects can be destroyed when all VkQueue objects retrieved from them are idle, and all objects created from them have been destroyed. This includes the following objects:
  - VkFence
  - VkSemaphore
  - VkEvent
  - VkQueryPool

- VkBuffer
- VkBufferView
- VkImage
- VkImageView
- VkShaderModule
- VkPipelineCache
- VkPipeline
- VkPipelineLayout
- VkSampler
- VkDescriptorSetLayout
- VkDescriptorPool
- VkFramebuffer
- VkRenderPass
- VkCommandPool
- VkCommandBuffer
- VkDeviceMemory

- VkPhysicalDevice objects cannot be explicitly destroyed. Instead, they are implicitly destroyed when the VkInstance object they are retrieved from is destroyed.
- VkInstance objects can be destroyed once all VkDevice objects created from any of its VkPhysicalDevice objects have been destroyed.

2.3 Command Syntax

The Specification describes Vulkan commands as functions or procedures using C99 syntax. Language bindings for other languages such as C++ and Javascript may allow for stricter parameter passing, or object-oriented interfaces.

With few exceptions, Vulkan uses the standard C types for parameters (int types from stdint.h, etc). Exceptions to this are using VkResult for return values, using VkBool32 for boolean values, VkDeviceSize for sizes and offsets pertaining to device address space, and VkFlags for passing bits or sets of bits of predefined values.

Commands that create Vulkan objects are of the form **`vkCreate*`** and take Vk*CreateInfo structures with the parameters needed to create the object. These Vulkan objects are destroyed with commands of the form **`vkDestroy*`**.

The last in-parameter to each command that creates or destroys a Vulkan object is *`pAllocator`*. The *`pAllocator`* parameter can be set to a non-`NULL` value such that allocations for the given object are delegated to an application provided callback; refer to the Memory Allocation chapter for further details.

Commands that allocate Vulkan objects owned by pool objects are of the form **`vkAllocate*`**, and take Vk*AllocateInfo structures. These Vulkan objects are freed with commands of the form **`vkFree*`**. These objects do not take allocators; if host memory is needed, they will use the allocator that was specified when their parent pool was created.

Information is retrieved from the implementation with commands of the form **`vkGet*`**, and commands are recorded into a command buffer with commands of the form **`vkCmd*`**.

2.4 Threading Behavior

Vulkan is intended to provide scalable performance when used on multiple host threads. All commands support being called concurrently from multiple threads, but certain parameters, or components of parameters are defined to be *externally synchronized*. This means that the caller must guarantee that no more than one thread is using such a parameter at a given time.

More precisely, Vulkan commands use simple stores to update software structures representing Vulkan objects. A parameter declared as externally synchronized may have its software structures updated at any time during the host execution of the command. If two commands operate on the same object and at least one of the commands declares the object to be externally synchronized, then the caller must guarantee not only that the commands do not execute simultaneously, but also that the two commands are separated by an appropriate memory barrier (if needed).

Note
Memory barriers are particularly relevant on the ARM CPU architecture which is more weakly ordered than many developers are accustomed to from x86/x64 programming. Fortunately, most higher-level synchronization primitives (like the pthread library) perform memory barriers as a part of mutual exclusion, so mutexing Vulkan objects via these primitives will have the desired effect.

Many object types are *immutable*, meaning the objects cannot change once they have been created. These types of objects never need external synchronization, except that they must not be destroyed while they are in use on another thread. In certain special cases, mutable object parameters are internally synchronized such that they do not require external synchronization. One example of this is the use of a VkPipelineCache in **`vkCreateGraphicsPipelines`** and **`vkCreateComputePipelines`**, where external synchronization around such a heavyweight command would be impractical. The implementation must internally synchronize the cache in this example, and may be able to do so in the form of a much finer-grained mutex around the command. Any command parameters that are not labeled as externally synchronized are either not mutated by the command or are internally synchronized. Additionally, certain objects related to a command's parameters (e.g. command pools and descriptor pools) may be affected by a command, and must also be externally synchronized. These implicit parameters are documented as described below.

Parameters of commands that are externally synchronized are listed below.

Externally Synchronized Parameters

- The *`instance`* parameter in `vkDestroyInstance`
- The *`device`* parameter in `vkDestroyDevice`
- The *`queue`* parameter in `vkQueueSubmit`
- The *`fence`* parameter in `vkQueueSubmit`
- The *`memory`* parameter in `vkFreeMemory`
- The *`memory`* parameter in `vkMapMemory`
- The *`memory`* parameter in `vkUnmapMemory`
- The *`buffer`* parameter in `vkBindBufferMemory`

- The *image* parameter in `vkBindImageMemory`
- The *queue* parameter in `vkQueueBindSparse`
- The *fence* parameter in `vkQueueBindSparse`
- The *fence* parameter in `vkDestroyFence`
- The *semaphore* parameter in `vkDestroySemaphore`
- The *event* parameter in `vkDestroyEvent`
- The *event* parameter in `vkSetEvent`
- The *event* parameter in `vkResetEvent`
- The *queryPool* parameter in `vkDestroyQueryPool`
- The *buffer* parameter in `vkDestroyBuffer`
- The *bufferView* parameter in `vkDestroyBufferView`
- The *image* parameter in `vkDestroyImage`
- The *imageView* parameter in `vkDestroyImageView`
- The *shaderModule* parameter in `vkDestroyShaderModule`
- The *pipelineCache* parameter in `vkDestroyPipelineCache`
- The *dstCache* parameter in `vkMergePipelineCaches`
- The *pipeline* parameter in `vkDestroyPipeline`
- The *pipelineLayout* parameter in `vkDestroyPipelineLayout`
- The *sampler* parameter in `vkDestroySampler`
- The *descriptorSetLayout* parameter in `vkDestroyDescriptorSetLayout`
- The *descriptorPool* parameter in `vkDestroyDescriptorPool`
- The *descriptorPool* parameter in `vkResetDescriptorPool`
- The *descriptorPool* member of the *pAllocateInfo* parameter in `vkAllocateDescriptorSets`
- The *descriptorPool* parameter in `vkFreeDescriptorSets`
- The *framebuffer* parameter in `vkDestroyFramebuffer`
- The *renderPass* parameter in `vkDestroyRenderPass`
- The *commandPool* parameter in `vkDestroyCommandPool`
- The *commandPool* parameter in `vkResetCommandPool`
- The *commandPool* member of the *pAllocateInfo* parameter in `vkAllocateCommandBuffers`
- The *commandPool* parameter in `vkFreeCommandBuffers`
- The *commandBuffer* parameter in `vkBeginCommandBuffer`

- The *commandBuffer* parameter in `vkEndCommandBuffer`
- The *commandBuffer* parameter in `vkResetCommandBuffer`
- The *commandBuffer* parameter in `vkCmdBindPipeline`
- The *commandBuffer* parameter in `vkCmdSetViewport`
- The *commandBuffer* parameter in `vkCmdSetScissor`
- The *commandBuffer* parameter in `vkCmdSetLineWidth`
- The *commandBuffer* parameter in `vkCmdSetDepthBias`
- The *commandBuffer* parameter in `vkCmdSetBlendConstants`
- The *commandBuffer* parameter in `vkCmdSetDepthBounds`
- The *commandBuffer* parameter in `vkCmdSetStencilCompareMask`
- The *commandBuffer* parameter in `vkCmdSetStencilWriteMask`
- The *commandBuffer* parameter in `vkCmdSetStencilReference`
- The *commandBuffer* parameter in `vkCmdBindDescriptorSets`
- The *commandBuffer* parameter in `vkCmdBindIndexBuffer`
- The *commandBuffer* parameter in `vkCmdBindVertexBuffers`
- The *commandBuffer* parameter in `vkCmdDraw`
- The *commandBuffer* parameter in `vkCmdDrawIndexed`
- The *commandBuffer* parameter in `vkCmdDrawIndirect`
- The *commandBuffer* parameter in `vkCmdDrawIndexedIndirect`
- The *commandBuffer* parameter in `vkCmdDispatch`
- The *commandBuffer* parameter in `vkCmdDispatchIndirect`
- The *commandBuffer* parameter in `vkCmdCopyBuffer`
- The *commandBuffer* parameter in `vkCmdCopyImage`
- The *commandBuffer* parameter in `vkCmdBlitImage`
- The *commandBuffer* parameter in `vkCmdCopyBufferToImage`
- The *commandBuffer* parameter in `vkCmdCopyImageToBuffer`
- The *commandBuffer* parameter in `vkCmdUpdateBuffer`
- The *commandBuffer* parameter in `vkCmdFillBuffer`
- The *commandBuffer* parameter in `vkCmdClearColorImage`
- The *commandBuffer* parameter in `vkCmdClearDepthStencilImage`
- The *commandBuffer* parameter in `vkCmdClearAttachments`

- The *commandBuffer* parameter in `vkCmdResolveImage`
- The *commandBuffer* parameter in `vkCmdSetEvent`
- The *commandBuffer* parameter in `vkCmdResetEvent`
- The *commandBuffer* parameter in `vkCmdWaitEvents`
- The *commandBuffer* parameter in `vkCmdPipelineBarrier`
- The *commandBuffer* parameter in `vkCmdBeginQuery`
- The *commandBuffer* parameter in `vkCmdEndQuery`
- The *commandBuffer* parameter in `vkCmdResetQueryPool`
- The *commandBuffer* parameter in `vkCmdWriteTimestamp`
- The *commandBuffer* parameter in `vkCmdCopyQueryPoolResults`
- The *commandBuffer* parameter in `vkCmdPushConstants`
- The *commandBuffer* parameter in `vkCmdBeginRenderPass`
- The *commandBuffer* parameter in `vkCmdNextSubpass`
- The *commandBuffer* parameter in `vkCmdEndRenderPass`
- The *commandBuffer* parameter in `vkCmdExecuteCommands`

There are also a few instances where a command can take in a user allocated list whose contents are externally synchronized parameters. In these cases, the caller must guarantee that at most one thread is using a given element within the list at a given time. These parameters are listed below.

Externally Synchronized Parameter Lists

- Each element of the *pWaitSemaphores* member of each element of the *pSubmits* parameter in `vkQueueSubmit`
- Each element of the *pSignalSemaphores* member of each element of the *pSubmits* parameter in `vkQueueSubmit`
- Each element of the *pWaitSemaphores* member of each element of the *pBindInfo* parameter in `vkQueueBindSparse`
- Each element of the *pSignalSemaphores* member of each element of the *pBindInfo* parameter in `vkQueueBindSparse`
- The *buffer* member of each element of the *pBufferBinds* member of each element of the *pBindInfo* parameter in `vkQueueBindSparse`
- The *image* member of each element of the *pImageOpaqueBinds* member of each element of the *pBindInfo* parameter in `vkQueueBindSparse`

- The *image* member of each element of the *pImageBinds* member of each element of the *pBindInfo* parameter in `vkQueueBindSparse`
- Each element of the *pFences* parameter in `vkResetFences`
- Each element of the *pDescriptorSets* parameter in `vkFreeDescriptorSets`
- The *dstSet* member of each element of the *pDescriptorWrites* parameter in `vkUpdateDescriptorSets`
- The *dstSet* member of each element of the *pDescriptorCopies* parameter in `vkUpdateDescriptorSets`
- Each element of the *pCommandBuffers* parameter in `vkFreeCommandBuffers`

In addition, there are some implicit parameters that need to be externally synchronized. For example, all *commandBuffer* parameters that need to be externally synchronized imply that the *commandPool* that was passed in when creating that command buffer also needs to be externally synchronized. The implicit parameters and their associated object are listed below.

Implicit Externally Synchronized Parameters

- All VkQueue objects created from *device* in `vkDeviceWaitIdle`
- Any VkDescriptorSet objects allocated from *descriptorPool* in `vkResetDescriptorPool`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdBindPipeline`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdSetViewport`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdSetScissor`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdSetLineWidth`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdSetDepthBias`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdSetBlendConstants`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdSetDepthBounds`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdSetStencilCompareMask`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdSetStencilWriteMask`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdSetStencilReference`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdBindDescriptorSets`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdBindIndexBuffer`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdBindVertexBuffers`

- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdDraw`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdDrawIndexed`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdDrawIndirect`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdDrawIndexedIndirect`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdDispatch`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdDispatchIndirect`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdCopyBuffer`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdCopyImage`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdBlitImage`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdCopyBufferToImage`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdCopyImageToBuffer`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdUpdateBuffer`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdFillBuffer`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdClearColorImage`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdClearDepthStencilImage`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdClearAttachments`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdResolveImage`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdSetEvent`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdResetEvent`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdWaitEvents`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdPipelineBarrier`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdBeginQuery`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdEndQuery`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdResetQueryPool`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdWriteTimestamp`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdCopyQueryPoolResults`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdPushConstants`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdBeginRenderPass`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdNextSubpass`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdEndRenderPass`
- The VkCommandPool that *commandBuffer* was allocated from, in `vkCmdExecuteCommands`

2.5 Errors

Vulkan is a layered API. The lowest layer is the core Vulkan layer, as defined by this Specification. The application can use additional layers above the core for debugging, validation, and other purposes.

One of the core principles of Vulkan is that building and submitting command buffers should be highly efficient. Thus error checking and validation of state in the core layer is minimal, although more rigorous validation can be enabled through the use of layers.

The core layer assumes applications are using the API correctly. Except as documented elsewhere in the Specification, the behavior of the core layer to an application using the API incorrectly is undefined, and system errors (possibly including program termination) may occur. However, implementations must ensure that incorrect usage by an application does not affect the integrity of the operating system, the Vulkan implementation, or other Vulkan client applications in the system, and does not allow one application to access data belonging to another application. Applications can request stronger robustness guarantees by enabling the *`robustBufferAccess`* feature as described in Chapter 29.

Validation of correct API usage is left to validation layers. Applications should be developed with validation layers enabled, to help catch and eliminate errors. Once validated, released applications should not enable validation layers by default.

2.5.1 Valid Usage

Certain usage rules apply to all commands in the API unless explicitly denoted differently for a command. These rules are as follows.

Any input parameter to a command that is an object handle must be a valid object handle, unless otherwise specified. An object handle is valid if:

- It has been created or allocated by a previous, successful call to the API. Such calls are noted in the main specification.
- It has not been deleted or freed by a previous call to the API. Such calls are noted in the main specification.
- Any objects referenced by that object, either as part of creation or usage, must also be valid.

The reserved handle VK_NULL_HANDLE can be passed in place of valid object handles when *explicitly called out in the specification*. Any command that creates an object successfully must not return VK_NULL_HANDLE. It is valid to pass VK_NULL_HANDLE to any **`vkDestroy*`** or **`vkFree*`** command, which will silently ignore these values.

Any parameter that is a pointer must be a valid pointer. A pointer is valid if it points at memory containing values, of the number and type(s) expected by the command.

Any parameter that is an enumerant must be a valid value for that enumerant type. A value is valid for an enumerant if:

- The value is defined as part of the enumerant type.
- The value is not one of the special values defined for an enumerant type, which are suffixed with _BEGIN_ RANGE, _END_RANGE, _RANGE_SIZE or _MAX_ENUM.

Any parameter that is a flag value must be a valid combination of bit flags. A valid combination is either zero or the bitwise OR of valid bit flags. A bit flag is valid if:

- The value is defined as part of the bits type, where the bits type is obtained by taking the flag type and replacing the trailing Flags with FlagBits. For example, a flag value of type `VkColorComponentFlags` must contain only values selected from the bit flags in `VkColorComponentFlagBits`.
- The flag is allowed in the context in which it is being used. For example, in some cases, certain bit flags or combinations of bit flags are mutually exclusive.

Any parameter that is a structure containing a VkStructureType *`sType`* member must have a value of *`sType`* matching the type of the structure. The correct value is described for each structure type, but as a general rule, the name of this value is obtained by taking the structure name, stripping the leading Vk, prefixing each capital letter with _, converting the entire resulting string to upper case, and prefixing it with VK_STRUCTURE_TYPE. For example, structures of type VkImageCreateInfo must have a *`sType`* value of VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO.

The values VK_STRUCTURE_TYPE_LOADER_INSTANCE_CREATE_INFO and VK_STRUCTURE_TYPE_LOADER_DEVICE_CREATE_INFO are reserved for internal use by the loader, and don't have corresponding Vulkan structures in this specification.

Any parameter that is a structure containing a void* *`pNext`* member must have a value of *`pNext`* that is either `NULL`, or points to a valid structure that is defined by an enabled extension. Extension structures are not described in the base Vulkan specification, but either in layered specifications incorporating those extensions, or in separate vendor-provided documents.

The above rules also apply recursively to members of structures provided as input to a command, either as a direct argument to the command, or themselves a member of another structure.

Specifics on valid usage of each command are covered in their individual sections.

2.5.2 Return Codes

While the core Vulkan API is not designed to capture incorrect usage, some circumstances still require return codes. Commands in Vulkan return their status via return codes that are in one of two categories:

- Successful completion codes are returned when a command needs to communicate success or status information. All successful completion codes are positive values.
- Run time error codes are returned when a command needs to communicate a failure that could only be detected at run time. All run time error codes are negative values.

All return codes in Vulkan are reported via VkResult return values. The possible codes are:

```
typedef enum VkResult {
    VK_SUCCESS = 0,
    VK_NOT_READY = 1,
    VK_TIMEOUT = 2,
    VK_EVENT_SET = 3,
    VK_EVENT_RESET = 4,
    VK_INCOMPLETE = 5,
    VK_ERROR_OUT_OF_HOST_MEMORY = -1,
    VK_ERROR_OUT_OF_DEVICE_MEMORY = -2,
    VK_ERROR_INITIALIZATION_FAILED = -3,
    VK_ERROR_DEVICE_LOST = -4,
    VK_ERROR_MEMORY_MAP_FAILED = -5,
    VK_ERROR_LAYER_NOT_PRESENT = -6,
    VK_ERROR_EXTENSION_NOT_PRESENT = -7,
    VK_ERROR_FEATURE_NOT_PRESENT = -8,
    VK_ERROR_INCOMPATIBLE_DRIVER = -9,
    VK_ERROR_TOO_MANY_OBJECTS = -10,
    VK_ERROR_FORMAT_NOT_SUPPORTED = -11,
    VK_ERROR_SURFACE_LOST_KHR = -1000000000,
    VK_SUBOPTIMAL_KHR = 1000001003,
    VK_ERROR_OUT_OF_DATE_KHR = -1000001004,
    VK_ERROR_INCOMPATIBLE_DISPLAY_KHR = -1000003001,
    VK_ERROR_NATIVE_WINDOW_IN_USE_KHR = -1000008000,
    VK_ERROR_VALIDATION_FAILED_EXT = -1000011001,
} VkResult;
```

Performance-critical commands generally do not have return codes. If a run time error occurs in such commands, the implementation will defer reporting the error until a specified point. For commands that record into command buffers (**`vkCmd*`**) run time errors are reported by **`vkEndCommandBuffer`**.

2.5.2.1 Success Codes

The success codes are:

Error	Meaning	Commands
VK_SUCCESS	Command successfully completed	Any
VK_NOT_READY	A fence or query has not yet completed	`vkGetFenceStatus`, `vkGetQueryPoolResults`
VK_TIMEOUT	A wait operation has not completed in the specified time	`vkWaitForFences`
VK_EVENT_SET	An event is signaled	`vkGetEventStatus`
VK_EVENT_RESET	An event is unsignaled	`vkGetEventStatus`
VK_INCOMPLETE	A return array was too small for the result	`vkEnumerateInstanceExtensionProperties`, `vkEnumerateInstanceLayerProperties`, `vkEnumerateDeviceExtensionProperties`, `vkEnumerateDeviceLayerProperties`

2.5.2.2 Error Codes

The run time error codes are:

Error	Meaning	Commands
VK_ERROR_OUT_OF_HOST_MEMORY	A host memory allocation has failed.	Any
VK_ERROR_OUT_OF_DEVICE_MEMORY	A device memory allocation has failed.	Any

Error	Meaning	Commands
VK_ERROR_DEVICE_LOST	The logical or physical device has been lost. See Lost Device	`vkQueueSubmit`, `vkWaitForFences`, `vkQueueWaitIdle`, `vkDeviceWaitIdle`, `vkGetQueryPoolResults`, `vkGetEventStatus`, `vkGetFenceStatus`, `vkCreateDevice`
VK_ERROR_INITIALIZATION_FAILED	Initialization of a object could not be completed for implementation-specific reasons.	`vkCreateInstance`, `vkEnumeratePhysicalDevices`, `vkCreateDevice`
VK_ERROR_MEMORY_MAP_FAILED	Mapping of a memory object has failed.	`vkMapMemory`
VK_ERROR_LAYER_NOT_PRESENT	A requested layer is not present or could not be loaded.	`vkCreateInstance`, `vkCreateDevice`
VK_ERROR_EXTENSION_NOT_PRESENT	A requested extension is not supported.	`vkCreateInstance`, `vkCreateDevice`
VK_ERROR_FEATURE_NOT_PRESENT	A requested feature is not supported.	`vkCreateDevice`
VK_ERROR_INCOMPATIBLE_DRIVER	The requested version of Vulkan is not supported by the driver or is otherwise incompatible for implementation-specific reasons.	`vkCreateInstance`
VK_ERROR_TOO_MANY_OBJECTS	Too many objects of the type have already been created.	`vkAllocateMemory`, `vkCreateSampler`
VK_ERROR_FORMAT_NOT_SUPPORTED	Requested format is not supported on this device.	`vkGetPhysicalDeviceImageFormatProperties`, `vkCreateImage`

If a command returns a run time error, it will leave any result pointers unmodified.

Out of memory errors do not damage any currently existing Vulkan objects. Objects that have already been successfully created can still be used by the application.

2.6 Numeric Representation and Computation

Implementations normally perform computations in floating-point, and must meet the range and precision requirements defined under *'Floating-Point Computation*' below.

These requirements only apply to computations performed in Vulkan operations outside of shader execution, such as texture image specification and sampling, and per-fragment operations. Range and precision requirements during shader execution differ and are specified by the Precision and Operation of SPIR-V Floating-Point Operations section.

In some cases, the representation and/or precision of operations is implicitly limited by the specified format of vertex or texel data consumed by Vulkan. Specific floating-point formats are described later in this section.

2.6.1 Floating-Point Computation

Most floating-point computation is performed in SPIR-V shader modules. The properties of computation within shaders are constrained as defined by the Precision and Operation of SPIR-V Floating-Point Operations section.

Some floating-point computation is performed outside of shaders, such as viewport and depth range calculations. For these computations, we do not specify how floating-point numbers are to be represented, or the details of how operations on them are performed. The remainder of this section applies only to computations performed outside shaders.

We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in 10^5. The maximum representable magnitude for all floating-point values must be at least 2^{32}. $x \cdot 0 = 0 \cdot x = 0$ for any non-infinite and non-NaN x. $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$.

Occasionally, further requirements will be specified. Most single-precision floating-point formats meet these requirements.

The special values Inf and $-Inf$ encode values with magnitudes too large to be represented; the special value NaN encodes "Not A Number" values resulting from undefined arithmetic operations such as $0/0$. Implementations may support Infs and NaNs in their floating-point computations.

Any representable floating-point value is legal as input to a Vulkan command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is unspecified, but must not lead to Vulkan interruption or termination. In [IEEE 754] arithmetic, for example, providing a negative zero or a denormalized number to an Vulkan command must yield deterministic results, while providing a NaN or Inf yields unspecified results.

2.6.2 16-Bit Floating-Point Numbers

A 16-bit floating-point number has a 1-bit sign (S), a 5-bit exponent (E), and a 10-bit mantissa (M). The value V of a 16-bit floating-point number is determined by the following:

$$V = \begin{cases} (-1)^S \times 0.0, & E = 0, M = 0 \\ (-1)^S \times 2^{-14} \times \frac{M}{2^{10}}, & E = 0, M \neq 0 \\ (-1)^S \times 2^{E-15} \times \left(1 + \frac{M}{2^{10}}\right), & 0 < E < 31 \\ (-1)^S \times Inf, & E = 31, M = 0 \\ NaN, & E = 31, M \neq 0 \end{cases}$$

If the floating-point number is interpreted as an unsigned 16-bit integer N, then

$$S = \left\lfloor \frac{N \bmod 65536}{32768} \right\rfloor$$
$$E = \left\lfloor \frac{N \bmod 32768}{1024} \right\rfloor$$
$$M = N \bmod 1024.$$

Any representable 16-bit floating-point value is legal as input to a Vulkan command that accepts 16-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or NaN) to such a command is unspecified, but must not lead to Vulkan interruption or termination. Providing a denormalized number or negative zero to Vulkan must yield deterministic results.

2.6.3 Unsigned 11-Bit Floating-Point Numbers

An unsigned 11-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 6-bit mantissa (M). The value V of an unsigned 11-bit floating-point number is determined by the following:

$$V = \begin{cases} 0.0, & E = 0, M = 0 \\ 2^{-14} \times \frac{M}{64}, & E = 0, M \neq 0 \\ 2^{E-15} \times \left(1 + \frac{M}{64}\right), & 0 < E < 31 \\ Inf, & E = 31, M = 0 \\ NaN, & E = 31, M \neq 0 \end{cases}$$

If the floating-point number is interpreted as an unsigned 11-bit integer N, then

$$E = \left\lfloor \frac{N}{64} \right\rfloor$$
$$M = N \bmod 64.$$

When a floating-point value is converted to an unsigned 11-bit floating-point representation, finite values are rounded to the closest representable finite value.

While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 65024 (the maximum finite representable unsigned 11-bit floating-point value) are converted to 65024. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 11-bit floating-point value is legal as input to a Vulkan command that accepts 11-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or NaN) to such a command is unspecified, but must not lead to Vulkan interruption or termination. Providing a denormalized number to Vulkan must yield deterministic results.

2.6.4 Unsigned 10-Bit Floating-Point Numbers

An unsigned 10-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 5-bit mantissa (M). The value V of an unsigned 10-bit floating-point number is determined by the following:

$$V = \begin{cases} 0.0, & E = 0, M = 0 \\ 2^{-14} \times \frac{M}{32}, & E = 0, M \neq 0 \\ 2^{E-15} \times \left(1 + \frac{M}{32}\right), & 0 < E < 31 \\ Inf, & E = 31, M = 0 \\ NaN, & E = 31, M \neq 0 \end{cases}$$

If the floating-point number is interpreted as an unsigned 10-bit integer N, then

$$E = \left\lfloor \frac{N}{32} \right\rfloor$$
$$M = N \bmod 32.$$

When a floating-point value is converted to an unsigned 10-bit floating-point representation, finite values are rounded to the closest representable finite value.

While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 64512 (the maximum finite representable unsigned 10-bit floating-point value) are converted to 64512. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative *NaN* are converted to positive *NaN*.

Any representable unsigned 10-bit floating-point value is legal as input to a Vulkan command that accepts 10-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or *NaN*) to such a command is unspecified, but must not lead to Vulkan interruption or termination. Providing a denormalized number to Vulkan must yield deterministic results.

2.6.5 General Requirements

Some calculations require division. In such cases (including implied divisions performed by vector normalization), division by zero produces an unspecified result but must not lead to Vulkan interruption or termination.

2.7 Fixed-Point Data Conversions

When generic vertex attributes and pixel color or depth components are represented as integers, they are often (but not always) considered to be *normalized*. Normalized integer values are treated specially when being converted to and from floating-point values, and are usually referred to as *normalized fixed-point*.

In the remainder of this section, b denotes the bit width of the fixed-point integer representation. When the integer is one of the types defined by the API, b is the bit width of that type. When the integer comes from an image containing color or depth component texels, b is the number of bits allocated to that component in its specified image format.

The signed and unsigned fixed-point representations are assumed to be b-bit binary twos-complement integers and binary unsigned integers, respectively.

2.7.1 Conversion from Normalized Fixed-Point to Floating-Point

Unsigned normalized fixed-point integers represent numbers in the range $[0,1]$. The conversion from an unsigned normalized fixed-point value c to the corresponding floating-point value f is defined as

$$f = \frac{c}{2^b - 1}$$

Signed normalized fixed-point integers represent numbers in the range $[-1,1]$. The conversion from a signed normalized fixed-point value c to the corresponding floating-point value f is performed using

$$f = \max\left\{\frac{c}{2^{b-1} - 1}, -1.0\right\}$$

Only the range $[-2^{b-1}+1, 2^{b-1}-1]$ is used to represent signed fixed-point values in the range $[-1,1]$. For example, if $b = 8$, then the integer value -127 corresponds to -1.0 and the value 127 corresponds to 1.0. Note that while zero is exactly expressible in this representation, one value (-128 in the example) is outside the representable range, and must be clamped before use. This equation is used everywhere that signed normalized fixed-point values are converted to floating-point, including for all signed normalized fixed-point parameters in GL commands, such as vertex attribute values, as well as for specifying texture or framebuffer values using signed normalized fixed-point.

2.7.2 Conversion from Floating-Point to Normalized Fixed-Point

The conversion from a floating-point value f to the corresponding unsigned normalized fixed-point value c is defined by first clamping f to the range $[0,1]$, then computing

$$f' = \text{convertFloatToUint}(f \times (2^b - 1), b)$$

where convertFloatToUint(r,b) returns one of the two unsigned binary integer values with exactly b bits which are closest to the floating-point value r (where rounding to nearest is preferred).

The conversion from a floating-point value f to the corresponding signed normalized fixed-point value c is performed by clamping f to the range $[-1,1]$, then computing

$$f' = \text{convertFloatToInt}(f \times (2^{b-1} - 1), b)$$

where convertFloatToInt(r,b) returns one of the two signed two's-complement binary integer values with exactly b bits which are closest to the floating-point value r (where rounding to nearest is preferred).

This equation is used everywhere that floating-point values are converted to signed normalized fixed-point, including when querying floating-point state and returning integers, as well as for specifying signed normalized texture or framebuffer values using floating-point.

2.8 API Version Numbers and Semantics

The Vulkan version number is used in several places in the API. In each such use, the API *major version number*, *minor version number*, and *patch version number* are packed into a 32-bit integer as follows:

- The major version number is a 10-bit integer packed into bits 31-22.
- The minor version number is a 10-bit integer packed into bits 21-12.
- The patch version number is a 12-bit integer packed into bits 11-0.

All Vulkan releases with the same major number are binary backwards compatible, but not forward compatible.

All Vulkan releases with the same major and minor number are binary backwards and forwards compatible. If two Vulkan releases have identical major numbers but different minor numbers, the one with the larger minor number must be a functional superset of the one with the smaller minor number, and should be an effective substitute for the first.

If two Vulkan releases have identical major and minor numbers but different patch numbers, the one with the larger patch number may contain additional bug fixes, and otherwise should be an effective substitute for the first.

2.9 Common Object Types

Some types of Vulkan objects are used in many different structures and command parameters, and are described here. These types include *offsets*, *extents*, and *rectangles*.

2.9.1 Offsets

Offsets are used to describe a pixel location within an image or framebuffer, as an (x,y) location for two-dimensional images, or an (x,y,z) location for three-dimensional images. Two- and three-dimensional offsets are respectively defined by the structures

```
typedef struct VkOffset2D {
    int32_t                                      x;
    int32_t                                      y;
} VkOffset2D;
```

```
typedef struct VkOffset3D {
    int32_t                                         x;
    int32_t                                         y;
    int32_t                                         z;
} VkOffset3D;
```

2.9.2 Extents

Extents are used to describe the size of a block of pixels within an image or framebuffer, as (width,height) for two-dimensional images, or as (width,height,depth) for three-dimensional images. Two- and three-dimensional extents are respectively defined by the structures

```
typedef struct VkExtent2D {
    int32_t                                         width;
    int32_t                                         height;
} VkExtent2D;
```

```
typedef struct VkExtent3D {
    int32_t                                         width;
    int32_t                                         height;
    int32_t                                         depth;
} VkExtent3D;
```

2.9.3 Rectangles

Rectangles are used to describe a specified rectangular block of pixels within an image or framebuffer. Rectangles include both an offset and an extent of the same dimensionality, as described above. Two-dimensional rectangles are defined by the structure

```
typedef struct VkRect2D {
    VkOffset2D                                      offset;
    VkExtent2D                                      extent;
} VkRect2D;
```

Chapter 3

Initialization

Before using Vulkan, an application must initialize it by loading the Vulkan commands, and creating a `VkInstance` object.

3.1 Command Function Pointers

Vulkan commands are not necessarily exposed statically on a platform. Function pointers for all Vulkan commands can be obtained with the command:

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

`instance` is the instance that the function pointer will be compatible with, and *`pName`* is the name of the command to obtain.

Valid Usage

- If *`instance`* is not `NULL`, *`instance`* must be a valid VkInstance handle
- *`pName`* must be a null-terminated string
- If *`instance`* is `NULL`, *`pName`* must be one of: **`vkEnumerateInstanceExtensionProperties`**, **`vkEnumerateInstanceLayerProperties`** or **`vkCreateInstance`**
- If *`instance`* is not `NULL`, *`pName`* must be the name of a core command or a command from an enabled extension, other than: **`vkEnumerateInstanceExtensionProperties`**, **`vkEnumerateInstanceLayerProperties`** or **`vkCreateInstance`**

`vkGetInstanceProcAddr` itself is obtained in a platform- and loader- specific manner. Typically, the loader library will export this command as a function symbol, so applications can link against the loader library, or load it dynamically and look up the symbol using platform-specific APIs. Loaders are encouraged to export function symbols for all other core Vulkan commands as well; if this is done, then applications that use only the core Vulkan commands have no need to use **vkGetInstanceProcAddr**.

Function pointers to commands that don't operate on a specific instance can be obtained by using this command with *instance* equal to `NULL`. The following commands can be accessed this way:

- **vkEnumerateInstanceExtensionProperties**
- **vkEnumerateInstanceLayerProperties**
- **vkCreateInstance**

If *instance* is a valid VkInstance, function pointers to any commands that operate on *instance* or a child of *instance* can be obtained. The returned function pointer must only be called with a dispatchable object (the first parameter) that is a child of *instance*.

If *pName* is not the name of a core Vulkan command, or is an extension command for any extension not supported by any available layer or implementation, then **vkGetInstanceProcAddr** will return `NULL`.

In order to support systems with multiple Vulkan implementations comprising heterogenous collections of hardware and software, the function pointers returned by **vkGetInstanceProcAddr** may point to dispatch code, which calls a different real implementation for different VkDevice objects (and objects created from them). The overhead of this internal dispatch can be avoided by obtaining device-specific function pointers for any commands that use a device or device-child object as their dispatchable object. Such function pointers can be obtained with the command:

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

device is the device the function pointer will be compatible with, and *pName* is the name of any Vulkan command whose first parameter is one of

- VkDevice
- VkQueue
- VkCommandBuffer

If *pName* is not the name of one of these Vulkan commands, and is not the name of an extension command belonging to an extension enabled for *device*, then **vkGetDeviceProcAddr** will return `NULL`.

Valid Usage

- *device* must be a valid VkDevice handle
- *pName* must be a null-terminated string
- *pName* must be the name of a supported command that has a first parameter of type VkDevice, VkQueue or VkCommandBuffer, either in the core API or an enabled extension

3.2 Instances

There is no global state in Vulkan and all per-application state is stored in a VkInstance object. Creating a VkInstance object initializes the Vulkan library and allows the application to pass information about itself to the implementation.

To create an instance object, call:

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

pAllocator controls host memory allocation as described in the Memory Allocation chapter. *pInstance* is a pointer to a VkInstance handle where the command writes its result, if successful.

Valid Usage

- *pCreateInfo* must be a pointer to a valid VkInstanceCreateInfo structure
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pInstance* must be a pointer to a VkInstance handle

The definition of VkInstanceCreateInfo is:

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

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *pApplicationInfo* is `NULL` or a pointer to an instance of VkApplicationInfo. If not `NULL`, this information helps implementations recognize behavior inherent to classes of applications. `VkApplicationInfo` is defined in detail below.
- *enabledLayerCount* is the number of global layers to enable.
- *ppEnabledLayerNames* is a pointer to an array of *enabledLayerCount* null-terminated UTF-8 strings containing the names of layers to enable.

- *enabledExtensionCount* is the number of global extensions to enable.
- *ppEnabledExtensionNames* is a pointer to an array of *enabledExtensionCount* null-terminated UTF-8 strings containing the names of extensions to enable.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`
- If *pApplicationInfo* is not `NULL`, *pApplicationInfo* must be a pointer to a valid VkApplicationInfo structure
- If *enabledLayerNameCount* is not `0`, *ppEnabledLayerNames* must be a pointer to an array of *enabledLayerNameCount* null-terminated strings
- If *enabledExtensionNameCount* is not `0`, *ppEnabledExtensionNames* must be a pointer to an array of *enabledExtensionNameCount* null-terminated strings
- Any given element of *ppEnabledLayerNames* must be the name of a layer present on the system, exactly matching a string returned in the VkLayerProperties structure by **`vkEnumerateInstanceLayerProperties`**
- Any given element of *ppEnabledExtensionNames* must be the name of an extension present on the system, exactly matching a string returned in the VkExtensionProperties structure by **`vkEnumerateInstanceExtensionProperties`**
- If an extension listed in *ppEnabledExtensionNames* is provided as part of a layer, then both the layer and extension must be enabled to enable that extension

`vkCreateInstance` creates the instance, then enables and initializes global layers and extensions requested by the application. If an extension is provided by a layer, both the layer and extension must be specified at **`vkCreateInstance`** time.

The *pApplicationInfo* member of `VkInstanceCreateInfo` can point to an instance of VkApplicationInfo. This structure is defined as:

```
typedef struct VkApplicationInfo {
    VkStructureType                                      sType;
    const void*                                          pNext;
    const char*                                          pApplicationName;
    uint32_t                                             applicationVersion;
    const char*                                          pEngineName;
    uint32_t                                             engineVersion;
    uint32_t                                             apiVersion;
} VkApplicationInfo;
```

- *sType* is the type of this structure.

- *pNext* is NULL or a pointer to an extension-specific structure.
- *pApplicationName* is a pointer to a null-terminated UTF-8 string containing the name of the application.
- *applicationVersion* is an unsigned integer variable containing the developer-supplied version number of the application.
- *pEngineName* is a pointer to a null-terminated UTF-8 string containing the name of the engine (if any) used to create the application.
- *engineVersion* is an unsigned integer variable containing the developer-supplied version number of the engine used to create the application.
- *apiVersion* is the version of the Vulkan API against which the application expects to run, encoded as described in the API Version Numbers and Semantics section.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_APPLICATION_INFO
- *pNext* must be NULL
- If *pApplicationName* is not NULL, *pApplicationName* must be a null-terminated string
- If *pEngineName* is not NULL, *pEngineName* must be a null-terminated string
- *apiVersion* must be a version that the implementation supports, or supports an effective substitute for

If **vkCreateInstance** is not successful, one of the following error codes is returned:

- VK_ERROR_INITIALIZATION_FAILED: initialization could not be completed for implementation-specific reasons.
- VK_ERROR_LAYER_NOT_PRESENT: a requested layer is not present or could not be loaded.
- VK_ERROR_EXTENSION_NOT_PRESENT: a requested extension is not supported.
- VK_ERROR_INCOMPATIBLE_DRIVER: the requested version of the API is not supported by the driver or is otherwise incompatible for implementation-specific reasons.

To destroy an instance, call:

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

instance is the handle of the instance to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- If *`instance`* is not `NULL`, *`instance`* must be a valid VkInstance handle
- If *`pAllocator`* is not `NULL`, *`pAllocator`* must be a pointer to a valid VkAllocationCallbacks structure
- All child objects created using *`instance`* must have been destroyed prior to destroying *`instance`*
- If VkAllocationCallbacks were provided when *`instance`* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *`instance`* was created, *`pAllocator`* must be `NULL`

Host Synchronization

- Host access to *`instance`* must be externally synchronized

Chapter 4

Devices and Queues

Once Vulkan is initialized, devices and queues are the primary objects used to interact with a Vulkan implementation.

Vulkan separates the concept of *physical* and *logical* devices. A physical device usually represents a single device in a system (perhaps made up of several individual hardware devices working together), of which there are a finite number. A logical device represents an application's view of the device.

4.1 Physical Devices

To retrieve a list of physical device objects representing the physical devices installed in the system, call:

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

`instance` is a handle to a Vulkan instance previously created with **`vkCreateInstance`**. If *`pPhysicalDevices`* is `NULL`, the number of physical devices available is returned in *`pPhysicalDeviceCount`*. If *`pPhysicalDevices`* is not `NULL`, *`pPhysicalDeviceCount`* must point to a variable set by the user to the size of the array pointed to by *`pPhysicalDevices`*, and is overwritten with the number of physical devices actually written to *`pPhysicalDevices`*.

Valid Usage

- *`instance`* must be a valid VkInstance handle
- *`pPhysicalDeviceCount`* must be a pointer to a uint32_t value
- If the value referenced by *`pPhysicalDeviceCount`* is not `0`, and *`pPhysicalDevices`* is not `NULL`, *`pPhysicalDevices`* must be a pointer to an array of *`pPhysicalDeviceCount`* VkPhysicalDevice handles

Once enumerated, general properties of the physical devices are queried by calling:

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

physicalDevice is the handle to the physical device whose properties will be queried. *pProperties* points to an instance of the VkPhysicalDeviceProperties structure, that will be filled with returned information.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *pProperties* must be a pointer to a VkPhysicalDeviceProperties structure

The definition of VkPhysicalDeviceProperties is:

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

The members of VkPhysicalDeviceProperties have the following meanings:

- *apiVersion* is the version of Vulkan supported by the device, encoded as described in the API Version Numbers and Semantics section.
- *driverVersion* is the vendor-specified version of the driver.
- *deviceID* is the PCI device ID.
- *vendorID* is the PCI vendor ID.
- *deviceType* is a `VkPhysicalDeviceType` specifying the type of device.
- *deviceName* is a pointer to a null-terminated UTF-8 string containing the name of the device.
- *pipelineCacheUUID* is an array of size VK_UUID_SIZE, containing 8-bit values that represent a universally unique identifier for the device.
- *limits* is the VkPhysicalDeviceLimits structure which specifies device-specific limits of the physical device. See Limits for details.
- *sparseProperties* is the VkPhysicalDeviceSparseProperties structure which specifies various sparse related properties of the physical device. See Sparse Properties for details.

The physical devices types are:

```
typedef enum VkPhysicalDeviceType {
    VK_PHYSICAL_DEVICE_TYPE_OTHER = 0,
    VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU = 1,
    VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU = 2,
    VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU = 3,
    VK_PHYSICAL_DEVICE_TYPE_CPU = 4,
} VkPhysicalDeviceType;
```

- VK_PHYSICAL_DEVICE_TYPE_OTHER The device that does not match any other available types.
- VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU The device is typically one embedded in or tightly coupled with the host.
- VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU The device is typically a separate processor connected to the host via an interlink.
- VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU The device is typically a virtual node in a virtualization environment.
- VK_PHYSICAL_DEVICE_TYPE_CPU The device is typically running on the same processors as the host.

The physical device type is advertised for informational purposes only, and does not directly affect the operation of the system. However, the device type may correlate with other advertised properties or capabilities of the system, such as how many memory heaps there are.

Properties of queues available on a physical device are queried by calling:

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

physicalDevice is the handle to the physical device whose properties will be queried. If *pQueueFamilyProperties* is `NULL`, then the number of queue families available is returned in *pQueueFamilyPropertyCount*. If *pQueueFamilyProperties* is not `NULL`, then *pQueueFamilyPropertyCount* must point to a variable set by the user to the size of the array pointed to by *pQueueFamilyProperties*, and is overwritten with the number of VkQueueFamilyProperties structures actually written to *pQueueFamilyProperties*.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *pQueueFamilyPropertyCount* must be a pointer to a uint32_t value
- If the value referenced by *pQueueFamilyPropertyCount* is not `0`, and *pQueueFamilyProperties* is not `NULL`, *pQueueFamilyProperties* must be a pointer to an array of *pQueueFamilyPropertyCount* VkQueueFamilyProperties structures

The definition of VkQueueFamilyProperties is:

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

The members of VkQueueFamilyProperties have the following meanings:

- *`queueFlags`* contains flags indicating the capabilities of the queues in this queue family.
- *`queueCount`* is the unsigned integer count of queues in this queue family.
- *`timestampValidBits`* is the unsigned integer count of meaningful bits in the timestamps written via **`vkCmdWriteTimestamp`**. The valid range for the count is 36..64 bits, or a value of 0, indicating no support for timestamps. Bits outside the valid range are guaranteed to be zeros.
- *`minImageTransferGranularity`* is the minimum granularity supported for image transfer operations on the queues in this queue family.

The bits specified in *`queueFlags`* are:

```
typedef enum VkQueueFlagBits {
    VK_QUEUE_GRAPHICS_BIT = 0x00000001,
    VK_QUEUE_COMPUTE_BIT = 0x00000002,
    VK_QUEUE_TRANSFER_BIT = 0x00000004,
    VK_QUEUE_SPARSE_BINDING_BIT = 0x00000008,
} VkQueueFlagBits;
```

- if VK_QUEUE_GRAPHICS_BIT is set, then the queues in this queue family support graphics operations.
- if VK_QUEUE_COMPUTE_BIT is set, then the queues in this queue family support compute operations.
- if VK_QUEUE_TRANSFER_BIT is set, then the queues in this queue family support transfer operations.
- if VK_QUEUE_SPARSE_BINDING_BIT is set, then the queues in this queue family support sparse memory management operations (see Sparse Resources). If any of the sparse resource features are enabled, then at least one queue family must support this bit.

If an implementation exposes any queue family that supports graphics operations, at least one queue family of at least one physical device exposed by the implementation must support both graphics and compute operations.

For further details see Queues.

The value returned in *`minImageTransferGranularity`* has a unit of blocks for images having a block compressed format, and a unit of texels otherwise.

Possible values of *`minImageTransferGranularity`* are:

- $(0,0,0)$ which indicates that only whole mip levels must be transferred using the image transfer operations on the corresponding queues. In this case, the following restrictions apply to all offset and extent parameters of image transfer operations:
  - The *`x`*, *`y`*, and *`z`* members of a VkOffset3D parameter must always be zero.

  - The *width*, *height*, and *depth* members of a VkExtent3D parameter must always match the width, height, and depth of the image subresource corresponding to the parameter, respectively.

- (Ax, Ay, Az) where Ax, Ay, and Az are all integer powers of two. In this case the following restrictions apply to all image transfer operations:

  - *x*, *y*, and *z* of a VkOffset3D parameter must be integer multiples of Ax, Ay, and Az, respectively.
  - *width* of a VkExtent3D parameter must be an integer multiple of Ax, or else $(x + width)$ must equal the width of the image subresource corresponding to the parameter.
  - *height* of a VkExtent3D parameter must be an integer multiple of Ay, or else $(y + height)$ must equal the height of the image subresource corresponding to the parameter.
  - *depth* of a VkExtent3D parameter must be an integer multiple of Az, or else $(z + depth)$ must equal the depth of the image subresource corresponding to the parameter.
  - If the format of the image corresponding to the parameters is one of the block compressed formats then for the purposes of the above calculations the granularity must be scaled up by the block size.

Queues supporting graphics and/or compute operations must report $(1, 1, 1)$ in *minImageTransferGranularity*, meaning that there are no additional restrictions on the granularity of image transfer operations for these queues. Other queues supporting image transfer operations are only required to support whole mip level transfers, thus the value of *minImageTransferGranularity* for queues belonging to such queue families may be $(0, 0, 0)$.

The Device Memory section describes memory properties queried from the physical device.

For physical device feature queries see the Features chapter.

4.2 Devices

Device objects represent logical connections to physical devices. Each device exposes a number of *queue families* each having one or more *queues*. All queues in a queue family support the same operations.

As described in Physical Devices, a Vulkan application will first query for all physical devices in a system. Each physical device can then be queried for its capabilities, including its queue and queue family properties. Once an acceptable physical device is identified, an application will create a corresponding logical device. An application must create a separate logical device for each physical device it will use. The created logical device is then the primary interface to the physical device.

How to enumerate the physical devices in a system and query those physical devices for their queue family properties is described in the Physical Device Enumeration section above.

4.2.1 Device Creation

A logical device is created as a *connection* to a physical device. To create a logical device, call:

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

physicalDevice must be one of the device handles returned from a call to **vkEnumeratePhysicalDevices** (see Physical Device Enumeration). *pCreateInfo* is a pointer to a VkDeviceCreateInfo structure containing information about how to create the device. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *pCreateInfo* must be a pointer to a valid VkDeviceCreateInfo structure
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pDevice* must be a pointer to a VkDevice handle

The definition of VkDeviceCreateInfo is:

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

The members of VkDeviceCreateInfo have the following meanings:

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *queueCreateInfoCount* is the unsigned integer size of the *pQueueCreateInfos* array. Refer to the Queue Creation section below for further details.
- *pQueueCreateInfos* is a pointer to an array of VkDeviceQueueCreateInfo structures describing the queues that are requested to be created along with the logical device. Refer to the Queue Creation section below for further details.
- *enabledLayerNameCount* is the unsigned integer size of the *ppEnabledLayerNames* array. Refer to the Querying Layers and Extensions chapter for further details.
- *ppEnabledLayerNames* is an array of strings (char pointers) containing the exact names of the layers to be enabled for the created device. Refer to the Querying Layers and Extensions chapter for further details.

- *enabledExtensionNameCount* is the unsigned integer size of the ppEnabledExtensionNames array. Refer to the Querying Layers and Extensions chapter for further details.
- *ppEnabledExtensionNames* is an array of strings (char pointers) containing the exact names of the extensions to be enabled for the created device. Refer to the Querying Layers and Extensions chapter for further details.
- *pEnabledFeatures* is a pointer to a VkPhysicalDeviceFeatures structure that contains boolean indicators of all the features to be enabled. Refer to the Features section for further details.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`
- *pQueueCreateInfos* must be a pointer to an array of *queueCreateInfoCount* valid VkDeviceQueueCreateInfo structures
- If *enabledLayerNameCount* is not `0`, and *ppEnabledLayerNames* is not `NULL`, *ppEnabledLayerNames* must be a pointer to an array of *enabledLayerNameCount* null-terminated strings
- If *enabledExtensionNameCount* is not `0`, and *ppEnabledExtensionNames* is not `NULL`, *ppEnabledExtensionNames* must be a pointer to an array of *enabledExtensionNameCount* null-terminated strings
- *pEnabledFeatures* must be a pointer to a valid VkPhysicalDeviceFeatures structure
- The value of *queueCreateInfoCount* must be greater than `0`
- Any given element of *ppEnabledLayerNames* must be the name of a layer present on the system, exactly matching a string returned in the VkLayerProperties structure by **`vkEnumerateDeviceLayerProperties`**
- Any given element of *ppEnabledExtensionNames* must be the name of an extension present on the system, exactly matching a string returned in the VkExtensionProperties structure by **`vkEnumerateDeviceExtensionProperties`**
- If an extension listed in *ppEnabledExtensionNames* is provided as part of a layer, then both the layer and extension must be enabled to enable that extension
- The *queueFamilyIndex* member of any given element of *pQueueCreateInfos* must be unique within *pQueueCreateInfos*

If **`vkCreateDevice`** is not successful, one of the following error codes is returned:

- VK_ERROR_INITIALIZATION_FAILED: initialization could not be completed for implementation-specific reasons.
- VK_ERROR_LAYER_NOT_PRESENT: a requested layer is not present or could not be loaded.

- VK_ERROR_EXTENSION_NOT_PRESENT: a requested extension is not supported.
- VK_ERROR_FEATURE_NOT_PRESENT: a requested feature is not supported.

4.2.2 Device Use

The logical device is the primary Vulkan interface to the graphics hardware.

Note
This is illustrated in Vulkan by the fact that the majority of Vulkan commands include "VkDevice *device*" as the first parameter.

The following is a high-level list of VkDevice uses along with references on where to find more information:

- Creation of queues. See the Queues section below for further details.
- Creation and tracking of various synchronization constructs. See Synchronization and Cache Control for further details.
- Allocating, freeing, and managing memory. See Memory Allocation and Resource Creation for further details.
- Creation and destruction of command buffers and command buffer pools. See Command Buffers for further details.
- Creation, destruction, and management of graphics state. See Pipelines and Resource Descriptors, among others, for further details.

4.2.3 Device Idle

A device is active while any of its queues have work to process. Once all device queues are idle, the device is idle. The command to wait for this condition is **vkDeviceWaitIdle**:

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

This command will block until the *device* referenced in the parameter is idle. Once the device is idle, **vkDeviceWaitIdle** will return VK_SUCCESS.

Valid Usage

- *device* must be a valid VkDevice handle

Host Synchronization

- Host access to all VkQueue objects created from `device` must be externally synchronized

4.2.4 Lost Device

A logical device may become *lost* because of hardware errors, execution timeouts, power management events and/or platform-specific events. This may cause pending and future command execution to fail and cause hardware resources to be corrupted. When this happens, certain commands will return VK_ERROR_DEVICE_LOST (see Error Codes for a list of such commands). After any such event, the logical device is considered *lost*. It is not possible to reset the logical device to a non-lost state, however the lost state is specific to a logical device (VkDevice), and the corresponding physical device (VkPhysicalDevice) may be otherwise unaffected. In some cases, the physical device may also be lost, and attempting to create a new logical device will fail, returning VK_ERROR_DEVICE_LOST. This is usually indicative of a problem with the underlying hardware, or its connection to the host. If the physical device has not been lost, and a new logical device is successfully created from that physical device, it must be in the non-lost state.

Note
Whilst logical device loss may be recoverable, in the case of physical device loss, it is unlikely that an application will be able to recover unless additional, unaffected physical devices exist on the system. The error is largely informational and intended only to inform the user that their hardware has probably developed a fault or become physically disconnected, and should be investigated further. In many cases, physical device loss may cause other more serious issues such as the operating system crashing; in which case it may not be reported via the Vulkan API.

Note
Undefined behavior caused by an application error may cause a device to become lost. However, such undefined behavior may also cause unrecoverable damage to the process, and it is then not guaranteed that the API objects, including the VkPhysicalDevice or the VkInstance are still valid or that the error is recoverable.

When a device is lost, its child objects are not implicitly destroyed and their handles are still valid. Those objects must still be destroyed before their parents or the device can be destroyed (see Lifetime). The host address space corresponding to device memory mapped using `vkMapMemory` is still valid, and host memory accesses to these mapped regions are still valid, but the contents are undefined. It is still legal to call any API command on the device and child objects.

Once a device is lost, command execution may fail, and commands that return a VkResult may return VK_ERROR_DEVICE_LOST. Commands that do not allow run-time errors must still operate correctly for valid usage and, if applicable, return valid data.

Commands that wait indefinitely for device execution (namely `vkDeviceWaitIdle`, `vkQueueWaitIdle`, `vkWaitForFences` with a maximum *`timeout`*, and `vkGetQueryPoolResults` with the VK_QUERY_RESULT_WAIT_BIT bit set in *`flags`*) must return in finite time even in the case of a lost device, and return either VK_SUCCESS or VK_ERROR_DEVICE_LOST. For any command that may return VK_ERROR_DEVICE_LOST, for the purpose of determining whether a command buffer is pending execution, or whether resources are considered in-use by the device, a return value of VK_ERROR_DEVICE_LOST is equivalent to VK_SUCCESS.

4.2.5 Device Destruction

To destroy a device, call:

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

`device` is the handle of the device to destroy. `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- If `device` is not `NULL`, `device` must be a valid VkDevice handle
- If `pAllocator` is not `NULL`, `pAllocator` must be a pointer to a valid VkAllocationCallbacks structure
- All child objects created on `device` must have been destroyed prior to destroying `device`
- If VkAllocationCallbacks were provided when `device` was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when `device` was created, `pAllocator` must be `NULL`

Host Synchronization

- Host access to `device` must be externally synchronized

To ensure that no work is active on the device, `vkDeviceWaitIdle` can be used to gate the destruction of the device. Prior to destroying a device, an application is responsible for destroying/freeing any Vulkan objects that were created using that device as the first parameter of the corresponding **`vkCreate*`** or **`vkAllocate*`** command.

Note
The lifetime of each of these objects is bound by the lifetime of the VkDevice object. Therefore, to avoid resource leaks, it is critical that an application explicitly free all of these resources prior to calling **`vkDestroyDevice`**. In the event of resource leaks, the validation layer will trigger errors at the time of **`vkDestroyDevice`**.

4.3 Queues

4.3.1 Queue Family Properties

As discussed in the Physical Device Enumeration section above, the `vkGetPhysicalDeviceQueueFamilyProperties` command is used to retrieve details about the queue families and queues supported by a device.

Each index in the *`pQueueFamilyProperties`* array returned by `vkGetPhysicalDeviceQueueFamilyProperties` describes a unique queue family on that physical device. These indices are used when creating queues, and they correspond directly with the *`queueFamilyIndex`* that is passed to the `vkCreateDevice` command via the `VkDeviceQueueCreateInfo` structure as described in the Queue Creation section below.

Grouping of queue families within a physical device is implementation-dependent.

Note
The general expectation is that a physical device groups all queues of matching capabilities into a single family. However, this is a recommendation to implementations and it is possible that a physical device may return two separate queue families with the same capabilities.

Once an application has identified a physical device with the queue(s) that it desires to use, it will create those queues in conjunction with a logical device. This is described in the following section.

4.3.2 Queue Creation

Creating a logical device also creates the queues associated with that device. The queues to create are described by a set of VkDeviceQueueCreateInfo structures that are passed to `vkCreateDevice` in *`pQueueCreateInfos`*. The definition of VkDeviceQueueCreateInfo is:

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

The members of VkDeviceQueueCreateInfo have the following meanings:

- *`sType`* is the type of this structure.
- *`pNext`* is `NULL` or a pointer to an extension-specific structure.
- *`flags`* is reserved for future use.
- *`queueFamilyIndex`* is an unsigned integer indicating the index of the queue family to create on this device. The value of this index corresponds to the index of an element of the *`pQueueFamilyProperties`* array that was returned by **`vkGetPhysicalDeviceQueueFamilyProperties`**.

* *queueCount* is an unsigned integer specifying the number of queues to create in the queue family indicated by *queueFamilyIndex*.
* *pQueuePriorities* is an array of *queueCount* normalized floating point values, specifying priorities of work that will be submitted to each created queue. See Queue Priority for more information.

Valid Usage

* *sType* must be VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO
* *pNext* must be `NULL`
* *flags* must be `0`
* *pQueuePriorities* must be a pointer to an array of *queueCount* float values
* The value of *queueCount* must be greater than `0`
* *queueFamilyIndex* must be less than the value of *pQueueFamilyPropertyCount* returned by **vkGetPhysicalDeviceQueueFamilyProperties**
* *queueCount* must be less than or equal to the value of the *queueCount* member of the VkQueueFamilyProperties structure, as returned by **vkGetPhysicalDeviceQueueFamilyProperties** in the *pQueueFamilyProperties*[*queueFamilyIndex*]
* The value of any given element of *pQueuePriorities* must be between `0.0` and `1.0` inclusive

Queue instances are queried by calling:

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

device parameter is a handle to a Vulkan device that was created with **vkCreateDevice**. *queueFamilyIndex* is the index of the queue family to which the queue belongs. *queueIndex* is the index within this queue family of the queue to retrieve. *pQueue* is a pointer to a VkQueue object that will be filled with the handle for the requested queue.

Valid Usage

* *device* must be a valid VkDevice handle
* *pQueue* must be a pointer to a VkQueue handle

- *queueFamilyIndex* must be one of the queue family indexes specified when *device* was created, via the VkDeviceQueueCreateInfo structure
- *queueIndex* must be less than the number of queues created for the specified queue family index when *device* was created, via the *queueCount* member of the VkDeviceQueueCreateInfo structure

4.3.3 Queue Family Index

The queue family index is used in multiple places in Vulkan in order to tie operations to a specific family of queues.

When retrieving a handle to the queue via **vkGetDeviceQueue**, the queue family index is used to select which queue family to retrieve the VkQueue handle from as described in the previous section.

When creating a VkCommandPool object (see Command Pools), a queue family index is specified in the VkCommandPoolCreateInfo structure. Command buffers from this pool can only be submitted on queues corresponding to this queue family.

When creating VkImage (see Images) and VkBuffer (see Buffers), a set of queue families is included in the `VkImageCreateInfo` and `VkBufferCreateInfo` structures to specify the queue families that can reference the resource.

When inserting a `VkBufferMemoryBarrier` or `VkImageMemoryBarrier` (see Section 6.3) a source and destination queue family index is specified to allow the ownership of a buffer or image to be transferred from one queue family to another. See the Resource Sharing section for details.

4.3.4 Queue Priority

Each queue is assigned a priority, as set in the VkDeviceQueueCreateInfo structures when creating the device. The priority of each queue is a normalized floating point value between 0.0 and 1.0, which is then translated to a discrete priority level by the implementation. Higher values indicate a higher priority, with 0.0 being the lowest priority and 1.0 being the highest.

Within the same device, queues with higher priority may be allotted more processing time than queues with lower priority. The implementation makes no guarantees with regards to ordering or scheduling among queues with the same priority, other than the constraints defined by explicit scheduling primitives. The implementation make no guarantees with regards to queues across different devices.

An implementation may allow a higher-priority queue to starve a lower-priority queue on the same VkDevice until the higher-priority queue has no further commands to execute. The relationship of queue priorities must not cause queues on one VkDevice to starve queues on another VkDevice.

No specific guarantees are made about higher priority queues receiving more processing time or better quality of service than lower priority queues.

4.3.5 Queue Synchronization

Wait on the completion of all work within a single queue by calling:

```
VkResult vkQueueWaitIdle(
    VkQueue                                     queue);
```

The *queue* parameter is the queue on which to wait. This command will block until all command buffers and sparse binding operations in the queue have completed. Once the queue is empty, VK_SUCCESS is returned.

Valid Usage

- *queue* must be a valid VkQueue handle

Synchronization between queues is done using Vulkan semaphores as described in the Synchronization and Cache Control chapter.

4.3.6 Sparse Memory Binding

In Vulkan it is possible to sparsely bind memory to buffers and images as described in the Sparse Resource chapter. Sparse memory binding is a queue operation. A queue whose flags include the VK_QUEUE_SPARSE_BINDING_BIT must be able to support the mapping of a virtual address to a physical address on the device. This causes an update to the page table mappings on the device. This update must be synchronized on a queue to avoid corrupting page table mappings during execution of graphics commands. By binding the sparse memory resources on queues, all commands that are dependent on the updated bindings are synchronized to only execute after the binding is updated. See the Synchronization and Cache Control chapter for how this synchronization is accomplished.

4.3.7 Queue Destruction

Queues are created along with a logical device during **`vkCreateDevice`**. All queues associated with a logical device are destroyed when **`vkDestroyDevice`** is called on that device.

Chapter 5

Command Buffers

Command buffers are objects used to record commands which can be subsequently submitted to a device queue for execution. There are two levels of command buffers - primary command buffers (which can execute secondary command buffers, and which are submitted to queues) and secondary command buffers (which can be called from primary command buffers, and which are not directly submitted to queues).

Recorded commands include commands to bind pipelines and descriptor sets to the command buffer, commands to modify dynamic state, commands to draw (for graphics rendering), commands to dispatch (for compute), commands to execute secondary command buffers (for primary command buffers only), commands to copy buffers and images, and other commands.

Each command buffer manages state independently of other command buffers. There is no inheritance of state across primary and secondary command buffers, or between secondary command buffers. When a command buffer begins recording, all state in that command buffer is undefined. When secondary command buffer(s) are recorded to execute on a primary command buffer, the secondary command buffer inherits no state from the primary command buffer, and all state of the primary command buffer is undefined after an execute secondary command buffer command is recorded. There is one exception to this rule - if the primary command buffer is inside a render pass instance, then the render pass and subpass state is not disturbed by executing secondary command buffers. Whenever the state of a command buffer is undefined, the application must set all relevant state on the command buffer before any state dependent commands such as draws and dispatches are recorded, otherwise the behavior of executing that command buffer is undefined.

Command buffers are referenced using VkCommandBuffer object handles.

Unless otherwise specified, and without explicit synchronization, the various commands submitted to a queue via command buffers may execute in arbitrary order relative to each other, and/or concurrently. Also, the memory side-effects of those commands may not be directly visible to other commands without memory barriers. This is true within a command buffer, and across command buffers submitted to a given queue. See Section 6.3, Section 6.5 and Section 6.5.3 about synchronization primitives suitable to guarantee execution order and side-effect visibility between commands on a given queue.

5.1 Command Pools

Command pools are opaque objects that command buffer memory is allocated from, and which allow the implementation to amortize the cost of resource creation across multiple command buffers. Command pools are application-synchronized, meaning that a command pool must not be used simultaneously in multiple threads at the same time. That includes use via recording commands on any command buffers allocated from the pool, as well as operations that allocate, free, and reset command buffers or the pool itself.

An application creates a command pool by calling:

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

`device` parameter specifies the logical device that the command pool is created on. `pCreateInfo` contains information used to create the command pool. The created pool is returned in `pCommandPool`. `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- `device` must be a valid VkDevice handle
- `pCreateInfo` must be a pointer to a valid VkCommandPoolCreateInfo structure
- If `pAllocator` is not `NULL`, `pAllocator` must be a pointer to a valid VkAllocationCallbacks structure
- `pCommandPool` must be a pointer to a VkCommandPool handle

The VkCommandPoolCreateInfo structure is defined as follows:

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

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is a combination of bitfield flags indicating usage behavior for the pool and command buffers allocated from it. Possible values include:

  ```
  typedef enum VkCommandPoolCreateFlagBits {
      VK_COMMAND_POOL_CREATE_TRANSIENT_BIT = 0x00000001,
      VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT = 0x00000002,
  } VkCommandPoolCreateFlagBits;
  ```

  - VK_COMMAND_POOL_CREATE_TRANSIENT_BIT indicates that command buffers allocated from the pool will be short-lived, meaning that they will be reset or freed in a relatively short timeframe. This flag may be used by the implementation to control memory allocation behavior within the pool.
  - VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT controls whether command buffers allocated from the pool can be individually reset. If this flag is set, individual command buffers allocated from the pool can be reset either explicitly, by calling **`vkResetCommandBuffer`**, or implicitly, by calling

  vkBeginCommandBuffer on a recorded command buffer. If this flag is not set, then **vkResetCommandBuffer** and **vkBeginCommandBuffer** (after recording) must not be called on the command buffers allocated from the pool, and they can only be reset in bulk by calling **vkResetCommandPool**.

- *queueFamilyIndex* designates a queue family as described in section Queue Family Properties. All command buffers created from this command pool must be submitted on queues from the same queue family.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO
- *pNext* must be NULL
- *flags* must be a valid combination of VkCommandPoolCreateFlagBits values
- *queueFamilyIndex* must be the index of a queue family available in the calling command's *device* parameter

Reset a command pool by calling:

```
VkResult vkResetCommandPool(
    VkDevice                                    device,
    VkCommandPool                               commandPool,
    VkCommandPoolResetFlags                     flags);
```

commandPool is the command pool to reset, and *device* is the device the command pool was created from. *flags* contains additional flags controlling the behavior of the reset.

Valid Usage

- *device* must be a valid VkDevice handle
- *commandPool* must be a valid VkCommandPool handle
- *flags* must be a valid combination of VkCommandPoolResetFlagBits values
- *commandPool* must have been created, allocated or retrieved from *device*
- Each of *device* and *commandPool* must have been created, allocated or retrieved from the same VkPhysicalDevice
- All VkCommandBuffer objects allocated from *commandPool* must not currently be pending execution

Host Synchronization

- Host access to *commandPool* must be externally synchronized

Resetting a command pool recycles all of the resources from all of the command buffers allocated from the command pool back to the command pool. All command buffers that have been allocated from the command pool are put in a state where it is legal to start recording commands by calling **vkBeginCommandBuffer**.

flags is of type VkCommandPoolResetFlags, which is defined as:

```
typedef enum VkCommandPoolResetFlagBits {
    VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT = 0x00000001,
} VkCommandPoolResetFlagBits;
```

If *flags* includes VK_COMMAND_POOL_RESET_RELEASE_RESOURCES_BIT, resetting a command pool recycles all of the resources from the command pool back to the system.

To destroy a command pool, call:

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

device is the device to be used to destroy the command pool. *commandPool* is the handle of the command pool to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *commandPool* is not VK_NULL_HANDLE, *commandPool* must be a valid VkCommandPool handle
- If *pAllocator* is not NULL, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *commandPool* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *commandPool* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All VkCommandBuffer objects allocated from *commandPool* must not be pending execution
- If VkAllocationCallbacks were provided when *commandPool* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *commandPool* was created, *pAllocator* must be NULL

Host Synchronization

- Host access to *`commandPool`* must be externally synchronized

When a pool is destroyed, all command buffers allocated from the pool are implicitly freed and become invalid. Command buffers allocated from a given pool do not need to be freed before destroying that command pool.

5.2 Command Buffer Lifetime

Command buffers are allocated by calling:

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

`device` is the logical device that the command buffers are allocated on. *`pAllocateInfo`* is an instance of the VkCommandBufferAllocateInfo structure which defines additional information about creating the pool. The allocated command buffers are returned in the *`pCommandBuffers`* array.

Valid Usage

- *`device`* must be a valid VkDevice handle
- *`pAllocateInfo`* must be a pointer to a valid VkCommandBufferAllocateInfo structure
- *`pCommandBuffers`* must be a pointer to an array of *`pAllocateInfo`*→bufferCount VkCommandBuffer handles

Host Synchronization

- Host access to *`pAllocateInfo`*→commandPool must be externally synchronized

The VkCommandBufferAllocateInfo structure is defined as:

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

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *commandPool* is the name of the command pool that the command buffers allocate their memory from.
- *level* determines whether the command buffers are primary or secondary command buffers. Possible values include:

  ```
  typedef enum VkCommandBufferLevel {
      VK_COMMAND_BUFFER_LEVEL_PRIMARY = 0,
      VK_COMMAND_BUFFER_LEVEL_SECONDARY = 1,
  } VkCommandBufferLevel;
  ```

- *bufferCount* is the number of command buffers to allocate from the pool.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO
- *pNext* must be `NULL`
- *commandPool* must be a valid VkCommandPool handle
- *level* must be a valid `VkCommandBufferLevel` value

Command buffers are reset by calling:

```
VkResult vkResetCommandBuffer(
    VkCommandBuffer                                 commandBuffer,
    VkCommandBufferResetFlags                       flags);
```

commandBuffer is the command buffer to be reset. *flags* is of type `VkCommandBufferResetFlags`:

```
typedef enum VkCommandBufferResetFlagBits {
    VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT = 0x00000001,
} VkCommandBufferResetFlagBits;
```

If *flags* includes VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT, then most or all memory resources currently owned by the command buffer should be returned to the parent command pool. If this flag is not set, then the command buffer may hold onto memory resources and reuse them when recording commands.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *flags* must be a valid combination of VkCommandBufferResetFlagBits values
- *commandBuffer* must not currently be pending execution
- *commandBuffer* must have been allocated from a pool that was created with the VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Command buffers are freed by calling:

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

where *device* is the logical device the command buffers were allocated from. *commandPool* is the handle of the command pool that the command buffers were allocated from. *pCommandBuffers* is an array of handles of command buffers to free.

Valid Usage

- *device* must be a valid VkDevice handle
- *commandPool* must be a valid VkCommandPool handle
- The value of *commandBufferCount* must be greater than 0
- *commandPool* must have been created, allocated or retrieved from *device*
- Each element of *pCommandBuffers* that is a valid handle must have been created, allocated or retrieved from *commandPool*
- Each of *device*, *commandPool* and the elements of *pCommandBuffers* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice

- All elements of *pCommandBuffers* must not be pending execution
- *pCommandBuffers* must be a pointer to an array of *commandBufferCount* VkCommandBuffer handles, each element of which must either be a valid handle or VK_NULL_HANDLE

Host Synchronization

- Host access to *commandPool* must be externally synchronized
- Host access to each member of *pCommandBuffers* must be externally synchronized

5.3 Command Buffer Recording

To begin recording a command buffer, an application calls:

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

where *commandBuffer* is the handle of the command buffer which is to be put in the recording state. *pBeginInfo* is an instance of the VkCommandBufferBeginInfo structure, which defines additional information about how the command buffer begins recording.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *pBeginInfo* must be a pointer to a valid VkCommandBufferBeginInfo structure
- *commandBuffer* must not be in the recording state
- If *commandBuffer* was allocated from a VkCommandPool which didn't have the VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT flag set, and has been previously recorded, the pool must be reset via **vkResetCommandPool** before calling **vkBeginCommandBuffer**
- If *commandBuffer* is a secondary command buffer and either the *occlusionQueryEnable* member of *pBeginInfo* is VK_FALSE, or the precise occlusion queries feature is not enabled, the *queryFlags* member of *pBeginInfo* must not contain VK_QUERY_CONTROL_PRECISE_BIT

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

The VkCommandBufferBeginInfo structure is defined as:

```
typedef struct VkCommandBufferBeginInfo {
    VkStructureType                          sType;
    const void*                              pNext;
    VkCommandBufferUsageFlags                flags;
    VkRenderPass                             renderPass;
    uint32_t                                 subpass;
    VkFramebuffer                            framebuffer;
    VkBool32                                 occlusionQueryEnable;
    VkQueryControlFlags                      queryFlags;
    VkQueryPipelineStatisticFlags            pipelineStatistics;
} VkCommandBufferBeginInfo;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is a combination of bitfield flags indicating usage behavior for the command buffer. Possible values include:

  ```
  typedef enum VkCommandBufferUsageFlagBits {
      VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT = 0x00000001,
      VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT = 0x00000002,
      VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT = 0x00000004,
  } VkCommandBufferUsageFlagBits;
  ```

  – VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT indicates that each recording of the command buffer will only be submitted once, and the command buffer will be reset and recorded again before submitting again.
  – If *flags* has VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT set, the command buffer is considered to be entirely inside a render pass.
  – Setting VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT allows the command buffer to be pending execution in multiple places at the same time.

- *renderPass* is a VkRenderPass object that the VkCommandBuffer will be rendering against if this is a secondary command buffer that was allocated with the VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT set.
- *subpass* is the index of the subpass within *renderPass* that the VkCommandBuffer will be rendering against if this is a secondary command buffer that was allocated with the VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT set.
- *framebuffer* refers to the VkFramebuffer object that the VkCommandBuffer will be rendering to if this was allocated with the VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT set. It can be VK_NULL_HANDLE if the framebuffer is not known.

Note
Specifying the exact framebuffer that the secondary command buffer will be executed with may result in better performance at command buffer execution time.

- `occlusionQueryEnable` indicates whether this secondary command buffer can be executed while an occlusion query is active in the primary command buffer. If this is VK_TRUE, then this command buffer can be executed whether the primary command buffer has an occlusion query active or not. If this is VK_FALSE, then the primary command buffer must not have an occlusion query active. If this is a primary command buffer, then this value is ignored.
- `queryFlags` indicates the query flags that can be used by an active occlusion query in the primary command buffer when this secondary command buffer is executed. If this value includes the VK_QUERY_CONTROL_PRECISE_BIT bit, then the active query can be precise or imprecise. If this bit is not set, then the active query must not use the VK_QUERY_CONTROL_PRECISE_BIT bit. If this is a primary command buffer, then this value is ignored.
- `pipelineStatistics` indicates the set of pipeline statistics that can be counted by an active query in the primary command buffer when this secondary command buffer is executed. If this value includes a given bit, then this command buffer can be executed whether the primary command buffer has a pipeline statistics query active that includes this bit or not. If this value excludes a given bit, then the active pipeline statistics query must not be from a query pool that counts that statistic. If this is a primary command buffer, then this value is ignored.

Valid Usage

- `sType` must be VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO
- `pNext` must be `NULL`
- `flags` must be a valid combination of `VkCommandBufferUsageFlagBits` values
- If `renderPass` is not VK_NULL_HANDLE, `renderPass` must be a valid VkRenderPass handle
- If `framebuffer` is not VK_NULL_HANDLE, `framebuffer` must be a valid VkFramebuffer handle
- Each of `renderPass` and `framebuffer` that are valid handles must have been created, allocated or retrieved from the same VkDevice
- If the inherited queries feature is not enabled, `occlusionQueryEnable` must be VK_FALSE
- If the inherited queries feature is enabled, `queryFlags` must be a valid combination of `VkQueryControlFlagBits` values
- If `flags` contains VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT, `renderpass` must not be VK_NULL_HANDLE, and must be compatible with the render pass for the render pass instance which this secondary command buffer will be executed in - see Section 7.2
- If `flags` contains VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT, and `framebuffer` is not VK_NULL_HANDLE, `framebuffer` must match the VkFramebuffer that is specified by **`vkCmdBeginRenderPass`** for the render pass instance which this secondary command buffer will be executed in

- If *flags* contains VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT, and *framebuffer* is not VK_NULL_HANDLE, *framebuffer* must have been created with a render pass that is compatible with *renderPass*
- If *renderPass* is not VK_NULL_HANDLE, subpass must refer to a valid subpass index within *renderPass*, specifically the index of the subpass which this secondary command buffer will be executed in

A primary command buffer is considered to be pending execution from the time it is submitted via **vkQueueSubmit** until that submission completes.

A secondary command buffer is considered to be pending execution from the time its execution is recorded into a primary buffer (via **vkCmdExecuteCommands**) until the final time that primary buffer's submission to a queue completes. If, after the primary buffer completes, the secondary command buffer is recorded to execute on a different primary buffer, the first primary buffer must not be resubmitted until after it is reset with vkResetCommandBuffer.

If VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT is not set on a secondary command buffer, that command buffer must not be used more than once in a given primary command buffer. Furthermore, if a secondary command buffer without VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT set is recorded to execute in a primary command buffer with VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT set, the primary command buffer must not be pending execution more than once at a time.

Note
On some implementations, not using the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT bit enables command buffers to be patched in-place if needed, rather than creating a copy of the command buffer.

If a command buffer has already been recorded and the command buffer was allocated from a command pool with the VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT flag set, then **vkBeginCommandBuffer** implicitly resets the command buffer, behaving as if **vkResetCommandBuffer** had been called with VK_COMMAND_BUFFER_RESET_RELEASE_RESOURCES_BIT not set.

Once recording starts, an application records a sequence of commands (**vkCmd***) to set state in the command buffer, draw, dispatch, and other commands.

An application completes recording by calling:

```
VkResult vkEndCommandBuffer(
    VkCommandBuffer                                 commandBuffer);
```

commandBuffer is the command buffer to complete recording.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *commandBuffer* must currently be in the recording state

- **`vkEndCommandBuffer`** must not be called inside a render pass instance
- All queries made active during the recording of *`commandBuffer`* must have been made inactive

Host Synchronization

- Host access to *`commandBuffer`* must be externally synchronized

If there was an error during recording, the application will be notified by an unsuccessful return code returned by **`vkEndCommandBuffer`**. If the application wishes to further use the command buffer, the command buffer must be reset.

After a command buffer completes recording, it can be submitted to a queue for execution.

5.4 Command Buffer Submission

Command buffers are submitted to a queue by calling:

```
VkResult vkQueueSubmit(
    VkQueue                                     queue,
    uint32_t                                    submitCount,
    const VkSubmitInfo*                         pSubmits,
    VkFence                                     fence);
```

`queue` is the handle of the queue that the command buffers will be submitted to. *`submitCount`* is the number of elements in the *`pSubmits`* array. *`pSubmits`* is a pointer to an array of `VkSubmitInfo` structures which describe the work to submit. All work described by *`pSubmits`* must be submitted to the queue before the command returns.

Valid Usage

- *`queue`* must be a valid VkQueue handle
- If *`submitCount`* is not `0`, *`pSubmits`* must be a pointer to an array of *`submitCount`* valid VkSubmitInfo structures
- If *`fence`* is not VK_NULL_HANDLE, *`fence`* must be a valid VkFence handle
- Each of *`queue`* and *`fence`* that are valid handles must have been created, allocated or retrieved from the same VkDevice
- *`fence`* must be unsignalled
- *`fence`* must not be associated with any other queue command that has not yet completed execution on that queue

Host Synchronization

- Host access to *queue* must be externally synchronized
- Host access to *pSubmits*[].pWaitSemaphores[] must be externally synchronized
- Host access to *pSubmits*[].pSignalSemaphores[] must be externally synchronized
- Host access to *fence* must be externally synchronized

Each submission of work is represented by a sequence of command buffers, each preceded by a list of semaphores upon which to wait before beginning execution of specific stages of commands in the command buffers, and followed by a second list of semaphores to signal upon completion of the work contained in the command buffers.

Note
The exact definition of a submission is platform-specific, but is considered a relatively expensive operation. In general, applications should attempt to batch work together into as few calls to **vkQueueSubmit** as possible.

Each call to **vkQueueSubmit** submits zero or more *batches* of work to the queue for execution. *submitCount* is used to specify the number of batches to submit. Each batch can reference a number of semaphores, and a corresponding set of stages that will wait for the semaphore to be signalled before executing any work, followed by a number of command buffers that will be executed, and finally, a number of semaphores that will be signaled after command buffer execution completes. Each batch is represented as an instance of the `VkSubmitInfo` structure stored in an array, the address of which is passed in *pSubmitInfo*. The definition of VkSubmitInfo is:

```
typedef struct VkSubmitInfo {
    VkStructureType                               sType;
    const void*                                   pNext;
    uint32_t                                      waitSemaphoreCount;
    const VkSemaphore*                            pWaitSemaphores;
    const VkPipelineStageFlags*                   pWaitDstStageMask;
    uint32_t                                      commandBufferCount;
    const VkCommandBuffer*                        pCommandBuffers;
    uint32_t                                      signalSemaphoreCount;
    const VkSemaphore*                            pSignalSemaphores;
} VkSubmitInfo;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *waitSemaphoreCount* is the number of semaphores upon which to wait before executing the command buffers for the batch.
- *pWaitSemaphores* is a pointer to an array of semaphores upon which to wait before executing the command buffers in the batch.

- `pWaitDstStageMask` is a pointer to an array of pipeline stages at which each corresponding semaphore wait will occur.
- `commandBufferCount` contains the number of command buffers to execute in the batch.
- `pCommandBuffers` is a pointer to an array of command buffers to execute in the batch. The command buffers submitted in a batch begin execution in the order they appear in `pCommandBuffers`, but may complete out of order.
- `signalSemaphoreCount` is the number of semaphores to be signaled once the commands specified in `pCommandBuffers` have completed execution.
- `pSignalSemaphores` is a pointer to an array of semaphores which will be signaled when the command buffers for this batch have completed execution.

Valid Usage

- `sType` must be VK_STRUCTURE_TYPE_SUBMIT_INFO
- `pNext` must be `NULL`
- If `waitSemaphoreCount` is not `0`, `pWaitSemaphores` must be a pointer to an array of `waitSemaphoreCount` valid VkSemaphore handles
- If `waitSemaphoreCount` is not `0`, `pWaitDstStageMask` must be a pointer to an array of `waitSemaphoreCount` valid combinations of `VkPipelineStageFlagBits` values
- Each element of `pWaitDstStageMask` must not be `0`
- If `commandBufferCount` is not `0`, `pCommandBuffers` must be a pointer to an array of `commandBufferCount` valid VkCommandBuffer handles
- If `signalSemaphoreCount` is not `0`, `pSignalSemaphores` must be a pointer to an array of `signalSemaphoreCount` valid VkSemaphore handles
- Each of the elements of `pWaitSemaphores`, the elements of `pCommandBuffers` and the elements of `pSignalSemaphores` that are valid handles must have been created, allocated or retrieved from the same VkDevice
- Any given element of `pSignalSemaphores` must currently be unsignalled
- Any given element of `pCommandBuffers` must either have been recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, or not currently be executing on the device
- Any given element of `pCommandBuffers` must not be in the recording state
- If any given element of `pCommandBuffers` contains references to secondary command buffers, those secondary command buffers must have been recorded with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT, or not currently be executing on the device
- If any given element of `pCommandBuffers` was created with VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT, it must not have been previously submitted without re-recording that command buffer

- Any given element of *pCommandBuffers* must not contain references to a secondary command buffer, if that secondary command buffer has been recorded in another primary command buffer after it was recorded into this VkCommandBuffer
- Any given element of *pCommandBuffers* must have been created on a VkCommandPool that was created for the same queue family that the calling command's *queue* belongs to
- Any given element of VkSemaphore in *pWaitSemaphores* must refer to a prior signal of that VkSemaphore that won't be consumed by any other wait on that semaphore
- If the geometry shaders feature is not enabled, any given element of *pWaitDstStageMask* must not contain *VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT*
- If the tessellation shaders feature is not enabled, any given element of *pWaitDstStageMask* must not contain *VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT* or *VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT*

If *fence* is provided, it must be in the unsignaled state (see Fences) and a fence must only be associated with a single submission until that submission completes, and the fence is subsequently reset. When all command buffers in *pCommandBuffers* have completed execution, the status of *fence* is set to signaled, providing certain implicit ordering guarantees.

5.5 Queue Forward Progress

The application must ensure that command buffer submissions will be able to complete without any subsequent operations by the application on any queue. After any call to **vkQueueSubmit**, for every queued wait on a semaphore there must be a prior signal of that semaphore that won't be consumed by a different wait on the semaphore.

Command buffers in the submission can include vkCmdWaitEvents commands that wait on events that won't be signaled by earlier commands in the queue. Such events must be signaled by the application using vkSetEvent, and the **vkCmdWaitEvents** commands that reference them must not be inside a render pass instance. Implementations may have limits on how long the command buffer will wait, in order to avoid interfering with progress of other clients of the device. If the event isn't signaled within these limits, results are undefined and may include device loss.

5.6 Secondary Command Buffer Execution

A secondary command buffer must not be directly submitted to a queue. Instead, secondary command buffers are recorded to execute as part of a primary command buffer with the command:

```
void vkCmdExecuteCommands(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    commandBuffersCount,
    const VkCommandBuffer*                      pCommandBuffers);
```

commandBuffer is a handle to a primary command buffer that the secondary command buffers are submitted to, and must be in the recording state. *pCommandBuffers* is an array of *commandBuffersCount* secondary command buffer handles, which are recorded to execute in the primary command buffer in the order they are listed in the array.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *pCommandBuffers* must be a pointer to an array of *commandBuffersCount* valid VkCommandBuffer handles
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support transfer, graphics or compute operations
- *commandBuffer* must be a primary VkCommandBuffer
- The value of *commandBuffersCount* must be greater than `0`
- Each of *commandBuffer* and the elements of *pCommandBuffers* must have been created, allocated or retrieved from the same VkDevice
- *commandBuffer* must have been created with a *level* value of VK_COMMAND_BUFFER_LEVEL_PRIMARY
- Any given element of *pCommandBuffers* must have been created with a *level* value of VK_COMMAND_BUFFER_LEVEL_SECONDARY
- Any given element of *pCommandBuffers* must not be already pending execution in *commandBuffer*, or appear twice in *pCommandBuffers*, unless it was created with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT flag
- Any given element of *pCommandBuffers* must not be already pending execution in any other VkCommandBuffer, unless it was created with the VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT flag
- Any given element of *pCommandBuffers* must not be in the recording state
- If **`vkCmdExecuteCommands`** is being called within a render pass instance, any given element of *pCommandBuffers* must have been recorded with the VK_COMMAND_BUFFER_USAGE_RENDER_PASS_CONTINUE_BIT
- If the inherited queries feature is not enabled, *commandBuffer* must not have any queries active
- If *commandBuffer* has a VK_QUERY_TYPE_OCCLUSION query active, then each element of *pCommandBuffers* must have been recorded with VkCommandBufferBeginInfo::*occlusionQueryEnable* set to VK_TRUE
- If *commandBuffer* has a VK_QUERY_TYPE_OCCLUSION query active, then each element of *pCommandBuffers* must have been recorded with VkCommandBufferBeginInfo::*queryFlags* having all bits set that are set for the query
- If *commandBuffer* has a VK_QUERY_TYPE_PIPELINE_STATISTICS query active, then each element of *pCommandBuffers* must have been recorded with VkCommandBufferBeginInfo::*pipelineStatistics* having all bits set that are set in the VkQueryPool the query uses
- Any given element of *pCommandBuffers* must not begin any query types that are active in *commandBuffer*

Host Synchronization

- Host access to *`commandBuffer`* must be externally synchronized

Once **`vkCmdExecuteCommands`** has been called, any references to the secondary command buffers specified by *`pCommandBuffers`* in any other primary command buffer becomes invalidated, unless those secondary command buffers were recorded with VK_COMMAND_BUFFER_USAGE_SIMULTANEOUS_USE_BIT.

5.7 Commands Allowed Inside Command Buffers

This section summarizes commands which record commands in command buffers. They are grouped in several different categories, corresponding to the different tables below.

Table 5.1: Commands that set state in the command buffer

Command	**Command buffer level**	**Render pass instance?**
`vkCmdBindPipeline`	Primary and secondary	Both
`vkCmdSetViewport`	Primary and secondary	Both
`vkCmdSetScissor`	Primary and secondary	Both
`vkCmdSetLineWidth`	Primary and secondary	Both
`vkCmdSetDepthBias`	Primary and secondary	Both
`vkCmdSetBlendConstants`	Primary and secondary	Both
`vkCmdSetDepthBounds`	Primary and secondary	Both
`vkCmdSetStencilCompareMask`	Primary and secondary	Both
`vkCmdSetStencilWriteMask`	Primary and secondary	Both
`vkCmdSetStencilReference`	Primary and secondary	Both
`vkCmdBindDescriptorSets`	Primary and secondary	Both
`vkCmdBindIndexBuffer`	Primary and secondary	Both
`vkCmdBindVertexBuffers`	Primary and secondary	Both

Table 5.2: Commands that cause the device to perform processing

Command	**Command buffer level**	**Render pass instance?**
`vkCmdDraw`	Primary and secondary	Inside
`vkCmdDrawIndexed`	Primary and secondary	Inside
`vkCmdDrawIndirect`	Primary and secondary	Inside
`vkCmdDrawIndexedIndirect`	Primary and secondary	Inside

Table 5.3: Commands that dispatch compute

Command	Command buffer level	Render pass instance?
`vkCmdDispatch`	Primary and secondary	Outside
`vkCmdDispatchIndirect`	Primary and secondary	Outside

Table 5.4: Commands that update and modify images and buffers

Command	Command buffer level	Render pass instance?
`vkCmdCopyBuffer`	Primary and secondary	Outside
`vkCmdCopyImage`	Primary and secondary	Outside
`vkCmdBlitImage`	Primary and secondary	Outside
`vkCmdCopyBufferToImage`	Primary and secondary	Outside
`vkCmdCopyImageToBuffer`	Primary and secondary	Outside
`vkCmdUpdateBuffer`	Primary and secondary	Outside
`vkCmdFillBuffer`	Primary and secondary	Outside
`vkCmdClearColorImage`	Primary and secondary	Outside
`vkCmdClearDepthStencilImage`	Primary and secondary	Outside
`vkCmdResolveImage`	Primary and secondary	Outside

Table 5.5: Commands that update and modify the currently bound frame-buffer

Command	Command buffer level	Render pass instance?
`vkCmdClearAttachments`	Primary and secondary	Inside

Table 5.6: Synchronization

Command	Command buffer level	Render pass instance?
`vkCmdSetEvent`	Primary and secondary	Outside
`vkCmdResetEvent`	Primary and secondary	Outside
`vkCmdWaitEvents`	Primary and secondary	Both
`vkCmdPipelineBarrier`	Primary and secondary	Both

Table 5.7: Queries

Command	**Command buffer level**	**Render pass instance?**
`vkCmdBeginQuery`	Primary and secondary	Both
`vkCmdEndQuery`	Primary and secondary	Both
`vkCmdResetQueryPool`	Primary and secondary	Outside
`vkCmdCopyQueryPoolResults`	Primary and secondary	Outside
`vkCmdWriteTimestamp`	Primary and secondary	Both*

Table 5.8: Push constants

Command	**Command buffer level**	**Render pass instance?**
`vkCmdPushConstants`	Primary and secondary	Both

Table 5.9: Render passes

Command	**Command buffer level**	**Render pass instance?**
`vkCmdBeginRenderPass`	Primary	Outside
`vkCmdNextSubpass`	Primary	Inside
`vkCmdEndRenderPass`	Primary	Inside

Table 5.10: Execute commands

Command	**Command buffer level**	**Render pass instance?**
`vkCmdExecuteCommands`	Primary	Both

Chapter 6

Synchronization and Cache Control

Synchronization of access to resources is primarily the responsibility of the application. In Vulkan, there are four forms of concurrency during execution: between the host and device, between the queues, between queue submissions, and between commands within a command buffer. Vulkan provides the application with a set of synchronization primitives for these purposes. Further, memory caches and other optimizations mean that the normal flow of command execution does not guarantee that all memory transactions from a command are immediately visible to other agents with views into a given block of memory. Vulkan also provides barrier operations to ensure this type of synchronization.

Four synchronization primitive types are exposed by Vulkan. These are:

- Fences
- Semaphores
- Events
- Barriers

Each is covered in detail in its own subsection of this chapter. Fences are used to communicate completion of execution of command buffer submissions to queues back to the application. Fences can therefore be used as a coarse-grained synchronization mechanism. Semaphores are generally associated with resources or groups of resources and can be used to marshal ownership of shared data. Their status is not visible to the host. Events provide a finer-grained synchronization primitive which can be signaled at command level granularity by both device and host, and can be waited upon by either. Barriers provide execution and memory synchronization between sets of commands.

6.1 Fences

Fences can be used by the host to determine completion of execution of submissions to queues performed with `vkQueueSubmit` and `vkQueueBindSparse`.

A fence's status is always either *signaled* or *unsignaled*. The host can poll the status of a single fence, or wait for any or all of a group of fences to become signaled.

To create a fence, use the command

```
VkResult vkCreateFence(
    VkDevice                                    device,
    const VkFenceCreateInfo*                    pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkFence*                                    pFence);
```

`vkCreateFence` creates a new fence object using the device specified by *`device`*. *`pCreateInfo`* points to a `VkFenceCreateInfo` structure specifying the state of the fence object. A handle to the resulting fence object is returned in the variable pointed to by *`pFence`*. *`pAllocator`* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *`device`* must be a valid VkDevice handle
- *`pCreateInfo`* must be a pointer to a valid VkFenceCreateInfo structure
- If *`pAllocator`* is not `NULL`, *`pAllocator`* must be a pointer to a valid VkAllocationCallbacks structure
- *`pFence`* must be a pointer to a VkFence handle

The definition of VkFenceCreateInfo is:

```
typedef struct VkFenceCreateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkFenceCreateFlags                          flags;
} VkFenceCreateInfo;
```

The *`flags`* member of the VkFenceCreateInfo structure pointed to by *`pCreateInfo`* contains flags defining the initial state and behavior of the fence. The flags are:

```
typedef enum VkFenceCreateFlagBits {
    VK_FENCE_CREATE_SIGNALED_BIT = 0x00000001,
} VkFenceCreateFlagBits;
```

If *`flags`* contains VK_FENCE_CREATE_SIGNALED_BIT then the fence object is created in the signaled state. Otherwise it is created in the unsignaled state.

Valid Usage

- *`sType`* must be VK_STRUCTURE_TYPE_FENCE_CREATE_INFO
- *`pNext`* must be `NULL`
- *`flags`* must be a valid combination of `VkFenceCreateFlagBits` values

A fence can be passed as a parameter to the queue submission commands, and when the associated queue submissions all complete execution the fence will transition from the unsignaled to the signaled state. See Command Buffer Submission and Binding Resource Memory.

To destroy a fence, call:

```
void vkDestroyFence(
    VkDevice                                    device,
    VkFence                                     fence,
    const VkAllocationCallbacks*                pAllocator);
```

device is the device to be used to destroy the fence. *fence* is the handle of the fence to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *fence* is not VK_NULL_HANDLE, *fence* must be a valid VkFence handle
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *fence* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *fence* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- *fence* must not be associated with any queue command that has not yet completed execution on that queue
- If VkAllocationCallbacks were provided when *fence* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *fence* was created, *pAllocator* must be `NULL`

Host Synchronization

- Host access to *fence* must be externally synchronized

To query the status of a fence from the host, use the command

```
VkResult vkGetFenceStatus(
    VkDevice                                    device,
    VkFence                                     fence);
```

device is the device on which the fence was created, and *fence* is the handle of the fence to query.

Valid Usage

- *device* must be a valid VkDevice handle
- *fence* must be a valid VkFence handle
- *fence* must have been created, allocated or retrieved from *device*
- Each of *device* and *fence* must have been created, allocated or retrieved from the same VkPhysicalDevice

Upon success, **vkGetFenceStatus** returns the status of the fence, which is one of:

- VK_SUCCESS indicates that the fence is signaled.
- VK_NOT_READY indicates that the fence is unsignaled.

To reset the status of one or more fences to unsignaled, so that they can be reused after a queue submission completes, use the command:

```
VkResult vkResetFences(
    VkDevice                                    device,
    uint32_t                                    fenceCount,
    const VkFence*                              pFences);
```

device is the device on which the fences were created, *fenceCount* is the number of fences to reset, and *pFences* is a pointer to an array of *fenceCount* fence handles. Each fence is reset to the unsignaled state.

Valid Usage

- *device* must be a valid VkDevice handle
- *pFences* must be a pointer to an array of *fenceCount* valid VkFence handles
- The value of *fenceCount* must be greater than 0
- Each element of *pFences* must have been created, allocated or retrieved from *device*
- Each of *device* and the elements of *pFences* must have been created, allocated or retrieved from the same VkPhysicalDevice
- Any given element of *pFences* must not currently be associated with any queue command that has not yet completed execution on that queue

Host Synchronization

- Host access to each member of *pFences* must be externally synchronized

To cause the host to wait until any one or all of a group of fences is signaled, use the command:

```
VkResult vkWaitForFences(
    VkDevice                                    device,
    uint32_t                                    fenceCount,
    const VkFence*                              pFences,
    VkBool32                                    waitAll,
    uint64_t                                    timeout);
```

device is the device on which the fences were created. *fenceCount* is the number of fences to wait on, and *pFences* is a pointer to an array of *fenceCount* fence handles.

The condition that must be satisfied to successfully unblock the wait is determined by the value of *waitAll*. If *waitAll* is VK_TRUE, then the condition is that all fences in *pFences* are signaled. Otherwise, the condition is that at least one fence in *pFences* is signaled.

timeout is the timeout period in units of nanoseconds. The value of *timeout* is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which may be substantially longer than one nanosecond, and may be longer than the requested period.

Valid Usage

- *device* must be a valid VkDevice handle
- *pFences* must be a pointer to an array of *fenceCount* valid VkFence handles
- The value of *fenceCount* must be greater than `0`
- Each element of *pFences* must have been created, allocated or retrieved from *device*
- Each of *device* and the elements of *pFences* must have been created, allocated or retrieved from the same VkPhysicalDevice

If the condition is satisfied when **vkWaitForFences** is called, then **vkWaitForFences** returns immediately. If the condition is not satisfied at the time **vkWaitForFences** is called, then **vkWaitForFences** will block and wait up to *timeout* nanoseconds for the condition to become satisfied.

If the value of *timeout* is zero, then **vkWaitForFences** does not wait, but simply returns the current state of the fences. VK_TIMEOUT will be returned in this case if the condition is not satisfied, even though no actual wait was performed.

If the specified timeout period expires before the condition is satisfied, **vkWaitForFences** returns VK_ TIMEOUT. If the condition is satisfied before *timeout* nanoseconds has expired, **vkWaitForFences** returns VK_SUCCESS.

Fences become signaled when the device completes executing the work that was submitted to a queue accompanied by the fence. But this alone is not sufficient for the host to be guaranteed to see the results of device writes to memory. To provide that guarantee, the application must insert a memory barrier between the device writes and the end of the submission that will signal the fence, with *dstAccessMask* having the VK_ACCESS_HOST_READ_BIT bit set, with *dstStageMask* having the VK_PIPELINE_STAGE_HOST_BIT bit set, and with the appropriate *srcStageMask* and *srcAccessMask* members set to guarantee completion of the writes. If the memory was allocated without the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT set, then **vkInvalidateMappedMemoryRanges** must be called after the fence is signaled in order to ensure the writes are visible to the host, as described in Host Access to Device Memory Objects.

6.2 Semaphores

Semaphores are used to coordinate operations between queues and between queue submissions within a single queue. An application might associate semaphores with resources or groups of resources to marshal ownership of shared data. A semaphore's status is always either *signaled* or *unsignaled*. Semaphores are signaled by queues and can also be waited on in the same or different queues until they are signaled.

To create a semaphore, use the command

```
VkResult vkCreateSemaphore(
    VkDevice                                    device,
    const VkSemaphoreCreateInfo*                pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkSemaphore*                                pSemaphore);
```

vkCreateSemaphore creates a new semaphore object using the device specified by *device*. *pCreateInfo* points to a VkSemaphoreCreateInfo structure specifying the state of the semaphore object. A handle to the resulting semaphore object is returned in the variable pointed to by *pSemaphore*. The semaphore is created in the unsignaled state. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- *pCreateInfo* must be a pointer to a valid VkSemaphoreCreateInfo structure
- If *pAllocator* is not NULL, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pSemaphore* must be a pointer to a VkSemaphore handle

The definition of VkSemaphoreCreateInfo is:

```
typedef struct VkSemaphoreCreateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkSemaphoreCreateFlags                      flags;
} VkSemaphoreCreateInfo;
```

The members of VkSemaphoreCreateInfo have the following meanings:

- *sType* is the type of this structure.

- *pNext* is NULL or a pointer to an extension-specific structure.
- *flags* is reserved for future use.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO
- *pNext* must be NULL
- *flags* must be 0

To destroy a semaphore, call:

```
void vkDestroySemaphore(
    VkDevice                                    device,
    VkSemaphore                                 semaphore,
    const VkAllocationCallbacks*                pAllocator);
```

device is the device to be used to destroy the semaphore. *semaphore* is the handle of the semaphore to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *semaphore* is not VK_NULL_HANDLE, *semaphore* must be a valid VkSemaphore handle
- If *pAllocator* is not NULL, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *semaphore* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *semaphore* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- *semaphore* must not be associated with any queue command that has not yet completed execution on that queue
- If VkAllocationCallbacks were provided when *semaphore* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *semaphore* was created, *pAllocator* must be NULL

Host Synchronization

- Host access to *`semaphore`* must be externally synchronized

To signal a semaphore from a queue, include it in an element of the array of `VkSubmitInfo` structures passed through the *`pSubmitInfo`* parameter to a call to `vkQueueSubmit`, or in an element of the array of `VkBindSparseInfo` structures passed through the *`pBindInfo`* parameter to a call to `vkQueueBindSparse`.

Semaphores included in the *`pSignalSemaphores`* array of one of the elements of a queue submission are signaled once queue execution reaches the signal operation, and all previous work in the queue completes. Any operations waiting on that semaphore in other queues will be released once it is signaled.

Similarly, to wait on a semaphore from a queue, include it in the *`pWaitSemaphores`* array of one of the elements of a batch in a queue submission. When queue execution reaches the wait operation, will stall execution of subsequently submitted operations until the semaphore reaches the signaled state due to a signaling operation. Once the semaphore is signaled, the subsequent operations will be permitted to execute and the status of the semaphore will be reset to the unsignaled state.

In the case of `VkSubmitInfo`, command buffers wait at specific pipeline stages, rather than delaying the entire command buffer's execution, with the pipeline stages determined by the value of the corresponding element of the *`pWaitDstStageMask`* member of VkSubmitInfo. Execution of work by those stages in subsequent commands is stalled until the corresponding semaphore reaches the signaled state. Subsequent sparse binding operations wait for the semaphore to become signaled, regardless of the values of *`pWaitDstStageMask`*.

Note
A common scenario for using *pWaitDstStageMask* with values other than VK_PIPELINE_STAGE_ALL_COMMANDS_BIT is when synchronizing a window system presentation operation against subsequent command buffers which render the next frame. In this case, an image that was being presented must not be overwritten until the presentation operation completes, but other pipeline stages can execute without waiting. A mask of VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT prevents subsequent color attachment writes from executing until the semaphore signals. Some implementations may be able to execute transfer operations and/or vertex processing work before the semaphore is signaled.
If an image layout transition needs to be performed on a swapchain image before it is used in a framebuffer, that can be performed as the first operation submitted to the queue after acquiring the image, and should not prevent other work from overlapping with the presentation operation. For example, a VkImageMemoryBarrier could use:

- *srcStageMask* = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- *srcAccessMask* = VK_ACCESS_MEMORY_READ_BIT
- *dstStageMask* = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- *dstAccessMask* = VK_ACCESS_COLOR_ATTACHMENT_READ_BIT | VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT.
- *oldLayout* = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
- *newLayout* = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL

Alternately, *oldLayout* can be VK_IMAGE_LAYOUT_UNDEFINED, if the image's contents need not be preserved.
This barrier accomplishes a dependency chain between previous presentation operations and subsequent color attachment output operations, with the layout transition performed in between, and does not introduce a dependency between previous work and any vertex processing stages. More precisely, the semaphore signals after the presentation operation completes, then the semaphore wait stalls the VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT stage, then there is a dependency from that same stage to itself with the layout transition performed in between.

When a queue signals or waits upon a semaphore, certain implicit ordering guarantees are provided.

Semaphore operations may not make the side effects of commands visible to the host.

6.3 Events

Events represent a fine-grained synchronization primitive that can be used to gauge progress through a sequence of commands executed on a queue by Vulkan. An event is initially in the unsignaled state. It can be signaled by a device, using commands inserted into the command buffer, or by the host. It can also be reset to the unsignaled state by a device or the host. The host can query the state of an event. A device can wait for one or more events to become signaled.

To create an event, call:

```
VkResult vkCreateEvent(
    VkDevice                                    device,
    const VkEventCreateInfo*                    pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkEvent*                                    pEvent);
```

device is the device that is to be used to create the event object and *pCreateInfo* is a pointer to an instance of the VkEventCreateInfo structure which contains information about how the event is to be created. A handle to the resulting event object is written into the variable whose address is given in *pEvent*. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- *pCreateInfo* must be a pointer to a valid VkEventCreateInfo structure
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pEvent* must be a pointer to a VkEvent handle

The definition of VkEventCreateInfo is:

```
typedef struct VkEventCreateInfo {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkEventCreateFlags                              flags;
} VkEventCreateInfo;
```

The *flags* member of the VkEventCreateInfo structure pointed to by *pCreateInfo* contains flags defining the behavior of the event. Currently, no flags are defined. When created, the event object is in the unsignaled state.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_EVENT_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`

To destroy an event, call:

```
void vkDestroyEvent(
    VkDevice                                        device,
    VkEvent                                         event,
    const VkAllocationCallbacks*                    pAllocator);
```

device is the device to be used to destroy the event. *event* is the handle of the event to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *event* is not VK_NULL_HANDLE, *event* must be a valid VkEvent handle
- If *pAllocator* is not NULL, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *event* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *event* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to *event* must have completed execution
- If VkAllocationCallbacks were provided when *event* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *event* was created, *pAllocator* must be NULL

Host Synchronization

- Host access to *event* must be externally synchronized

To query the state of an event from the host, call:

```
VkResult vkGetEventStatus(
    VkDevice                                    device,
    VkEvent                                     event);
```

vkGetEventStatus returns the current state of the event object specified in *event*. *device* is used solely to determine ownership of the event.

Valid Usage

- *device* must be a valid VkDevice handle
- *event* must be a valid VkEvent handle
- *event* must have been created, allocated or retrieved from *device*
- Each of *device* and *event* must have been created, allocated or retrieved from the same VkPhysicalDevice

Upon success, **`vkGetEventStatus`** returns the state of the event object with the following return codes:

Status	Meaning
VK_EVENT_SET	The event specified by `event` is signaled.
VK_EVENT_RESET	The event specified by `event` is unsignaled.

The state of an event can be updated by the host. The state of the event is immediately changed, and subsequent calls to **`vkGetEventStatus`** will return the new state.

To set the state of an event to signaled from the host, call:

```
VkResult vkSetEvent(
    VkDevice                                    device,
    VkEvent                                     event);
```

`device` is the device that was used to create the *`event`* object that is being signaled.

Valid Usage

- *`device`* must be a valid VkDevice handle
- *`event`* must be a valid VkEvent handle
- *`event`* must have been created, allocated or retrieved from *`device`*
- Each of *`device`* and *`event`* must have been created, allocated or retrieved from the same VkPhysicalDevice

Host Synchronization

- Host access to *`event`* must be externally synchronized

To set the state of an event to unsignaled from the host, call:

```
VkResult vkResetEvent(
    VkDevice                                    device,
    VkEvent                                     event);
```

`device` is the device that was used to create the *`event`* object that is being reset.

Valid Usage

- *device* must be a valid VkDevice handle
- *event* must be a valid VkEvent handle
- *event* must have been created, allocated or retrieved from *device*
- Each of *device* and *event* must have been created, allocated or retrieved from the same VkPhysicalDevice

Host Synchronization

- Host access to *event* must be externally synchronized

The state of an event can also be updated on the device by commands inserted in command buffers. To set the state of an event to signaled from a device, call:

```
void vkCmdSetEvent(
    VkCommandBuffer                                 commandBuffer,
    VkEvent                                         event,
    VkPipelineStageFlags                            stageMask);
```

commandBuffer is the command buffer into which the command is recorded. *event* is the event that will be signaled. *stageMask* specifies the pipeline stage at which the state of *event* is updated as described below.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *event* must be a valid VkEvent handle
- *stageMask* must be a valid combination of `VkPipelineStageFlagBits` values
- *stageMask* must not be `0`
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics or compute operations
- This command must only be called outside of a render pass instance
- Each of *commandBuffer* and *event* must have been created, allocated or retrieved from the same VkDevice
- If the geometry shaders feature is not enabled, *stageMask* must not contain *VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT*
- If the tessellation shaders feature is not enabled, *stageMask* must not contain *VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT* or *VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT*

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

To set the state of an event to unsignaled from a device, call:

```
void vkCmdResetEvent(
    VkCommandBuffer                             commandBuffer,
    VkEvent                                     event,
    VkPipelineStageFlags                        stageMask);
```

commandBuffer is the command buffer into which the command is recorded. *event* is the event that will be reset. *stageMask* specifies the pipeline stage at which the state of *event* is updated as described below.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *event* must be a valid VkEvent handle
- *stageMask* must be a valid combination of `VkPipelineStageFlagBits` values
- *stageMask* must not be `0`
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics or compute operations
- This command must only be called outside of a render pass instance
- Each of *commandBuffer* and *event* must have been created, allocated or retrieved from the same VkDevice
- If the geometry shaders feature is not enabled, *stageMask* must not contain *VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT*
- If the tessellation shaders feature is not enabled, *stageMask* must not contain *VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT* or *VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT*

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

For both **vkCmdSetEvent** and **vkCmdResetEvent**, the status of *event* is updated once the pipeline stages specified by *stageMask* (see Section 6.5.2) have completed executing prior commands. The command modifying the event is passed through the pipeline bound to the command buffer at time of execution.

To wait for one or more events to enter the signaled state on a device, call:

```
void vkCmdWaitEvents(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    eventCount,
    const VkEvent*                              pEvents,
    VkPipelineStageFlags                        srcStageMask,
    VkPipelineStageFlags                        dstStageMask,
    uint32_t                                    memoryBarrierCount,
    const void* const*                          ppMemoryBarriers);
```

commandBuffer is the command buffer into which the command is recorded. *pEvents* is an array of *eventCount* event object handles to wait on. *srcStageMask* (see Section 6.5.2) is the bitwise OR of the pipeline stages used to signal the event object handles in *pEvents*. *dstStageMask* is the pipeline stages at which the wait will occur. *ppMemoryBarriers* is an array of *memoryBarrierCount* pointers to memory barrier structures. Vulkan provides three types of memory barriers: global memory, buffer memory, and image memory (see Section 6.5.3 for more details).

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *pEvents* must be a pointer to an array of *eventCount* valid VkEvent handles
- *srcStageMask* must be a valid combination of `VkPipelineStageFlagBits` values
- *srcStageMask* must not be `0`
- *dstStageMask* must be a valid combination of `VkPipelineStageFlagBits` values
- *dstStageMask* must not be `0`
- If *memoryBarrierCount* is not `0`, *ppMemoryBarriers* must be a pointer to an array of *memoryBarrierCount* pointers
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics or compute operations
- The value of *eventCount* must be greater than `0`
- Each of *commandBuffer* and the elements of *pEvents* must have been created, allocated or retrieved from the same VkDevice
- *srcStageMask* must be the bitwise OR of the *stageMask* parameter used in previous calls to **vkCmdSetEvent** with any of the members of *pEvents*
- If **vkSetEvent** was used to signal any of the events in *pEvents*, *srcStageMask* must include the VK_ PIPELINE_STAGE_HOST_BIT flag

- If the geometry shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the geometry shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`
- If the tessellation shaders feature is not enabled, `srcStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- If the tessellation shaders feature is not enabled, `dstStageMask` must not contain `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT` or `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`
- If the value of `memoryBarrierCount` is not `0`, any given element of `ppMemoryBarriers` must point to a valid VkMemoryBarrier, VkBufferMemoryBarrier or VkImageMemoryBarrier structure
- If `pEvents` includes one or more events that will be signaled by **`vkSetEvent`** after `commandBuffer` has been submitted to a queue, then **`vkCmdWaitEvents`** must not be called inside a render pass instance

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

`vkCmdWaitEvents` waits for events set by either **`vkSetEvent`** or **`vkCmdSetEvent`** to become signaled. Logically, it has three phases:

1. Wait at the pipeline stages specified by `dstStageMask` (see Section 6.5.2) until the `eventCount` event objects specified by `pEvents` become signaled. Implementations may wait for each event object to become signaled in sequence (starting with the first event object in `pEvents`, and ending with the last), or wait for all of the event objects to become signaled at the same time.
2. Execute the `memoryBarrierCount` memory barriers specified by `ppMemoryBarriers` (see Section 6.5.3).
3. Resume execution of pipeline stages specified by `dstStageMask`

Implementations may not execute commands in a pipelined manner, so **`vkCmdWaitEvents`** may not observe the results of a subsequent **`vkCmdSetEvent`** or **`vkCmdResetEvent`** command, even if the stages in `dstStageMask` occur after the stages in `srcStageMask`.

Commands that update the state of events in different pipeline stages may execute out of order, unless the ordering is enforced by execution dependencies.

Note
Applications should be careful to avoid race conditions when using events. For example, an event should only be reset if no **`vkCmdWaitEvents`** command is executing that references that event.

An act of setting or resetting an event in one queue may not affect or be visible to other queues. For cross-queue synchronization, semaphores can be used.

6.4 Execution And Memory Dependencies

Synchronization commands introduce explicit execution and memory dependencies between two sets of action commands, where the second set of commands depends on the first set of commands. The two sets can be:

- First set: commands before a `vkCmdSetEvent` command.

  Second set: commands after a `vkCmdWaitEvents` command in the same queue, using the same event.
- First set: commands in a lower numbered subpass (or before a render pass instance).

  Second set: commands in a higher numbered subpass (or after a render pass instance), where there is a subpass dependency between the two subpasses (or between a subpass and VK_SUBPASS_EXTERNAL).
- First set: commands before a pipeline barrier.

  Second set: commands after that pipeline barrier in the same queue (possibly limited to within the same subpass).

An *execution dependency* is a single dependency between a set of source and destination pipeline stages, which guarantees that all work performed by the set of pipeline stages included in *`srcStageMask`* (see Pipeline Stage Flags) of the first set of commands completes before any work performed by the set of pipeline stages included in *`dstStageMask`* of the second set of commands begins.

An *execution dependency chain* from a set of source pipeline stages *A* to a set of destination pipeline stages *B* is a sequence of execution dependencies submitted to a queue in order between a first set of commands and a second set of commands, satisfying the following conditions:

- the first dependency includes *A* or VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT or VK_PIPELINE_STAGE_ALL_COMMANDS_BIT in the *`srcStageMask`*. And,
- the final dependency includes *B* or VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT or VK_PIPELINE_STAGE_ALL_COMMANDS_BIT in the *`dstStageMask`*. And,
- for each dependency in the sequence (except the first) at least one of the following conditions is true:
  - *`srcStageMask`* of the current dependency includes at least one bit *C* that is present in the *`dstStageMask`* of the previous dependency. Or,
  - *`srcStageMask`* of the current dependency includes VK_PIPELINE_STAGE_ALL_COMMANDS_BIT or VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT. Or,
  - *`dstStageMask`* of the previous dependency includes VK_PIPELINE_STAGE_ALL_COMMANDS_BIT or VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT. Or,
  - *`srcStageMask`* of the current dependency includes VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT, and *`dstStageMask`* of the previous dependency includes at least one graphics pipeline stage. Or,
  - *`dstStageMask`* of the previous dependency includes VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT, and *`srcStageMask`* of the current dependency includes at least one graphics pipeline stage.

A pair of consecutive execution dependencies in an execution dependency chain accomplishes a dependency between the stages *A* and *B* via intermediate stages *C*, even if no work is executed between them that uses the pipeline stages included in *C*.

An execution dependency chain guarantees that the work performed by the pipeline stages *A* in the first set of commands completes before the work performed by pipeline stages *B* in the second set of commands begins.

An execution dependency is *by-region* if its `dependencyFlags` parameter includes VK_DEPENDENCY_BY_REGION_BIT. Such a barrier describes a per-region (x,y,layer) dependency. That is, for each region, the implementation must ensure that the source stages for the first set of commands complete execution before any destination stages begin execution in the second set of commands for the same region. Since fragment shader invocations are not specified to run in any particular groupings, the size of a region is implementation-dependent, not known to the application, and must be assumed to be no larger than a single pixel. If `dependencyFlags` does not include VK_DEPENDENCY_BY_REGION_BIT, it describes a global dependency, that is for all pixel regions, the source stages must have completed for preceding commands before any destination stages starts for subsequent commands.

Memory dependencies synchronize accesses to memory between two sets of commands. They operate according to two "halves" of a dependency to synchronize two sets of commands, the commands that execute first vs the commands that execute second, as described above. The first half of the dependency makes memory accesses using the set of access types in `srcAccessMask` performed in pipeline stages in `srcStageMask` by the first set of commands complete and writes be *available* for subsequent commands. The second half of the dependency makes any available writes from previous commands *visible* to pipeline stages in `dstStageMask` using the set of access types in `dstAccessMask` for the second set of commands, if those writes have been made available with the first half of the same or a previous dependency. The two halves of a memory dependency can either be expressed as part of a single command, or can be part of separate barriers as long as there is an execution dependency chain between them. The application must use memory dependencies to make writes visible before subsequent reads can rely on them, and before subsequent writes can overwrite them. Failure to do so causes the result of the reads to be undefined, and the order of writes to be undefined.

Global memory barriers apply to all resources owned by the device. Buffer and image memory barriers apply to the buffer range(s) or image subresource(s) included in the command. For accesses to a byte of a buffer or subresource of an image to be synchronized between two sets of commands, the byte or subresource must be included in both the first and second halves of the dependencies described above, but need not be included in each step of the execution dependency chain between them.

An execution dependency chain is *by-region* if all stages in all dependencies in the chain are framebuffer-space pipeline stages, and if the VK_DEPENDENCY_BY_REGION_BIT bit is included in all dependencies in the chain. Otherwise, the execution dependency chain is not by-region. The two halves of a memory dependency form a by-region dependency if **all** execution dependency chains between them are by-region. In other words, if there is any execution dependency between two sets of commands that is not by-region, then the memory dependency is not by-region.

When an image memory barrier includes a layout transition, the barrier first makes writes via `srcStageMask` and `srcAccessMask` available, then performs the layout transition, then makes the contents of the image subresource(s) in the new layout visible to memory accesses in `dstStageMask` and `dstAccessMask`, as if there is an execution and memory dependency between the source masks and the transition, as well as between the transition and the destination masks. Any writes that have previously been made available are included in the layout transition, but any previous writes that have not been made available may become lost or corrupt the image.

All dependencies must include at least one bit in each of the `srcStageMask` and `dstStageMask`.

Memory dependencies are used to solve data hazards, e.g. to ensure that write operations are visible to subsequent read operations (read-after-write hazard), as well as write-after-write hazards. Write-after-read and read-after-read hazards only require execution dependencies to synchronize.

6.5 Pipeline Barriers

A *pipeline barrier* inserts an execution dependency and a set of memory dependencies between a set of commands earlier in the command buffer and a set of commands later in the command buffer. A pipeline barrier is recorded by calling:

```
void vkCmdPipelineBarrier(
    VkCommandBuffer                             commandBuffer,
    VkPipelineStageFlags                        srcStageMask,
    VkPipelineStageFlags                        dstStageMask,
    VkDependencyFlags                           dependencyFlags,
    uint32_t                                    memoryBarrierCount,
    const void* const*                          ppMemoryBarriers);
```

commandBuffer is the command buffer into which the command is recorded. The pipeline barrier specifies an execution dependency such that all work performed by the set of pipeline stages included in *srcStageMask* (see Section 6.5.2) of the first set of commands completes before any work performed by the set of pipeline stages included in *dstStageMask* of the second set of commands begins. The execution dependency is by-region if *dependencyFlags* includes VK_DEPENDENCY_BY_REGION_BIT.

If **vkCmdPipelineBarrier** is called outside a render pass instance, then the first set of commands is all prior commands submitted to the queue and recorded in the command buffer and the second set of commands is all subsequent commands recorded in the command buffer and submitted to the queue. If **vkCmdPipelineBarrier** is called inside a render pass instance, then the first set of commands is all prior commands in the same subpass and the second set of commands is all subsequent commands in the same subpass.

ppMemoryBarriers is an array of *memoryBarrierCount* pointers to memory barrier structures. Each element specifies two halves of a memory dependency, as defined above. Specifics of each type of memory barrier and the memory access types are defined further in Memory Barriers.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *srcStageMask* must be a valid combination of `VkPipelineStageFlagBits` values
- *srcStageMask* must not be `0`
- *dstStageMask* must be a valid combination of `VkPipelineStageFlagBits` values
- *dstStageMask* must not be `0`
- *dependencyFlags* must be a valid combination of `VkDependencyFlagBits` values
- If *memoryBarrierCount* is not `0`, *ppMemoryBarriers* must be a pointer to an array of *memoryBarrierCount* pointers
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support transfer, graphics or compute operations

- If the geometry shaders feature is not enabled, *srcStageMask* must not contain *VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT*
- If the geometry shaders feature is not enabled, *dstStageMask* must not contain *VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT*
- If the tessellation shaders feature is not enabled, *srcStageMask* must not contain *VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT* or *VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT*
- If the tessellation shaders feature is not enabled, *dstStageMask* must not contain *VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT* or *VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT*
- If the value of *memoryBarrierCount* is not `0`, any given element of *ppMemoryBarriers* must point to a valid VkMemoryBarrier, VkBufferMemoryBarrier or VkImageMemoryBarrier structure
- If **`vkCmdPipelineBarrier`** is called within a render pass instance, the render pass must declare at least one self-dependency from the current subpass to itself - see Subpass Self-dependency

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

6.5.1 Subpass Self-dependency

If **`vkCmdPipelineBarrier`** is called inside a render pass instance, the following restrictions apply. For a given subpass to allow a pipeline barrier, the render pass must declare a *self-dependency* from that subpass to itself. That is, there must exist a VkSubpassDependency in the subpass dependency list for the render pass with *srcSubpass* and *dstSubpass* equal to that subpass index. More than one self-dependency can be declared for each subpass. Self-dependencies must only include pipeline stage bits that are graphics stages. Self-dependencies must not have any earlier pipeline stages depend on any later pipeline stages. More precisely, this means that whatever is the last pipeline stage in *srcStageMask* must be no later than whatever is the first pipeline stage in *dstStageMask* (the latest source stage can be equal to the earliest destination stage). If the source and destination stage masks both include framebuffer-space stages, then *dependencyFlags* must include VK_DEPENDENCY_BY_REGION_BIT.

A **`vkCmdPipelineBarrier`** command inside a render pass instance must be a *subset* of one of the self-dependencies of the subpass it is used in, meaning that the stage masks and access masks must each include only a subset of the bits of the corresponding mask in that self-dependency. If the self-dependency has VK_DEPENDENCY_BY_REGION_BIT set, then so must the pipeline barrier. Pipeline barriers within a render pass instance can only be types VkMemoryBarrier or VkImageMemoryBarrier. If a VkImageMemoryBarrier is used, the image and subresource range specified in the barrier must be a subset of one of the image views used by the framebuffer in the current subpass. Additionally, *oldLayout* must be equal to *newLayout*, and both the *srcQueueFamilyIndex* and *dstQueueFamilyIndex* must be VK_QUEUE_FAMILY_IGNORED.

6.5.2 Pipeline Stage Flags

Several of the event commands, **vkCmdPipelineBarrier**, and VkSubpassDependency depend on being able to specify where in the logical pipeline events can be signaled or the source and destination of an execution dependency. These pipeline stages are specified with the bitfield:

```
typedef enum VkPipelineStageFlagBits {
    VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT = 0x00000001,
    VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT = 0x00000002,
    VK_PIPELINE_STAGE_VERTEX_INPUT_BIT = 0x00000004,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT = 0x00000008,
    VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT = 0x00000010,
    VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT = 0x00000020,
    VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT = 0x00000040,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT = 0x00000080,
    VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT = 0x00000100,
    VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT = 0x00000200,
    VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT = 0x00000400,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT = 0x00000800,
    VK_PIPELINE_STAGE_TRANSFER_BIT = 0x00001000,
    VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT = 0x00002000,
    VK_PIPELINE_STAGE_HOST_BIT = 0x00004000,
    VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT = 0x00008000,
    VK_PIPELINE_STAGE_ALL_COMMANDS_BIT = 0x00010000,
} VkPipelineStageFlagBits;
```

The meaning of each bit is:

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT: Stage of the pipeline where commands are initially received by the queue.
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT: Stage of the pipeline where Draw/DispatchIndirect data structures are consumed.
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT: Stage of the pipeline where vertex and index buffers are consumed.
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT: Vertex shader stage.
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT: Tessellation control shader stage.
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT: Tessellation evaluation shader stage.
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT: Geometry shader stage.
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT: Fragment shader stage.
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT: Stage of the pipeline where early fragment tests (depth/stencil test before fragment shading) are performed.
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT: Stage of the pipeline where late fragment tests (depth/stencil test after fragment shading) are performed.
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT: Stage of the pipeline after blending where the final color values are output from the pipeline. This stage also includes resolve operations that occur at the end of a subpass. Note that this does not necessarily indicate that the values have been committed to memory.

- VK_PIPELINE_STAGE_TRANSFER_BIT: Execution of copy commands. This includes the operations resulting from all transfer commands. The set of transfer commands comprises **`vkCmdCopyBuffer`**, **`vkCmdCopyImage`**, **`vkCmdBlitImage`**, **`vkCmdCopyBufferToImage`**, **`vkCmdCopyImageToBuffer`**, **`vkCmdUpdateBuffer`**, **`vkCmdFillBuffer`**, **`vkCmdClearColorImage`**, **`vkCmdClearDepthStencilImage`**, **`vkCmdResolveImage`**, and **`vkCmdCopyQueryPoolResults`**.
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT: Execution of a compute shader.
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT: Final stage in the pipeline where commands complete execution.
- VK_PIPELINE_STAGE_HOST_BIT: A pseudo-stage indicating execution on the host of reads/writes of device memory.
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT: Execution of all graphics pipeline stages.
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT: Execution of all stages supported on the queue.

Note
The VK_PIPELINE_STAGE_ALL_COMMANDS_BIT and VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT differ from VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT in that they correspond to all (or all graphics) stages, rather than to a specific stage at the end of the pipeline. An execution dependency with only VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT in *`dstStageMask`* will not delay subsequent commands, while including either of the other two bits will. Similarly, when defining a memory dependency, if the stage mask(s) refer to all stages, then the indicated access types from all stages will be made available and/or visible, but using only VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT would not make any accesses available and/or visible because this stage doesn't access memory. The VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT is useful for accomplishing memory barriers and layout transitions when the next accesses will be done in a different queue or by a presentation engine; in these cases subsequent commands in the same queue do not need to wait, but the barrier or transition must complete before semaphores associated with the batch signal.

Note
If an implementation is unable to update the state of an event at any specific stage of the pipeline, it may instead update the event at any logically later stage. For example, if an implementation is unable to signal an event immediately after vertex shader execution is complete, it may instead signal the event after color attachment output has completed. In the limit, an event may be signaled after all graphics stages complete. If an implementation is unable to wait on an event at any specific stage of the pipeline, it may instead wait on it at any logically earlier stage.
Similarly, if an implementation is unable to implement an execution dependency at specific stages of the pipeline, it may implement the dependency in a way where additional source pipeline stages complete and/or where additional destination pipeline stages' execution is blocked to satisfy the dependency.
If an implementation makes such a substitution, it must not affect the semantics of execution or memory dependencies or image and buffer memory barriers.

Certain pipeline stages are only available on queues that support a particular set of operations. The following table lists, for each pipeline stage flag, which queue capability flag must be supported by the queue. When multiple flags are enumerated in the second column of the table, it means that the pipeline stage is supported on the queue if it supports any of the listed capability flags. For further details on queue capabilities see Physical Device Enumeration and Queues.

Table 6.1: Supported pipeline stage flags

Pipeline stage flag	**Required queue capability flag**
VK_PIPELINE_STAGE_TOP_OF_ PIPE_BIT	None
VK_PIPELINE_STAGE_DRAW_ INDIRECT_BIT	VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT
VK_PIPELINE_STAGE_VERTEX_ INPUT_BIT	VK_QUEUE_GRAPHICS_BIT
VK_PIPELINE_STAGE_VERTEX_ SHADER_BIT	VK_QUEUE_GRAPHICS_BIT
VK_PIPELINE_STAGE_ TESSELLATION_CONTROL_ SHADER_BIT	VK_QUEUE_GRAPHICS_BIT
VK_PIPELINE_STAGE_ TESSELLATION_EVALUATION_ SHADER_BIT	VK_QUEUE_GRAPHICS_BIT
VK_PIPELINE_STAGE_ GEOMETRY_SHADER_BIT	VK_QUEUE_GRAPHICS_BIT
VK_PIPELINE_STAGE_ FRAGMENT_SHADER_BIT	VK_QUEUE_GRAPHICS_BIT
VK_PIPELINE_STAGE_EARLY_ FRAGMENT_TESTS_BIT	VK_QUEUE_GRAPHICS_BIT
VK_PIPELINE_STAGE_LATE_ FRAGMENT_TESTS_BIT	VK_QUEUE_GRAPHICS_BIT
VK_PIPELINE_STAGE_COLOR_ ATTACHMENT_OUTPUT_BIT	VK_QUEUE_GRAPHICS_BIT
VK_PIPELINE_STAGE_COMPUTE_ SHADER_BIT	VK_QUEUE_COMPUTE_BIT
VK_PIPELINE_STAGE_ TRANSFER_BIT	VK_QUEUE_GRAPHICS_BIT, VK_ QUEUE_COMPUTE_BIT, or VK_ QUEUE_TRANSFER_BIT
VK_PIPELINE_STAGE_BOTTOM_ OF_PIPE_BIT	None
VK_PIPELINE_STAGE_HOST_BIT	None
VK_PIPELINE_STAGE_ALL_ GRAPHICS_BIT	VK_QUEUE_GRAPHICS_BIT
VK_PIPELINE_STAGE_ALL_ COMMANDS_BIT	None

6.5.3 Memory Barriers

Memory barriers express the two halves of a memory dependency between an earlier set of memory accesses against a later set of memory accesses. Vulkan provides three types of memory barriers: global memory, buffer memory, and image memory.

6.5.4 Global Memory Barriers

The global memory barrier type is specified with an instance of the VkMemoryBarrier structure. This type of barrier applies to memory accesses involving all memory objects that exist at the time of its execution. The definition of VkMemoryBarrier is:

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

The members of VkMemoryBarrier have the following meanings:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `srcAccessMask` is a mask of the classes of memory accesses performed by the first set of commands that will participate in the dependency.
- `dstAccessMask` is a mask of the classes of memory accesses performed by the second set of commands that will participate in the dependency.

Valid Usage

- `sType` must be VK_STRUCTURE_TYPE_MEMORY_BARRIER
- `pNext` must be `NULL`
- `srcAccessMask` must be a valid combination of `VkAccessFlagBits` values
- `dstAccessMask` must be a valid combination of `VkAccessFlagBits` values

`srcAccessMask` and `dstAccessMask`, along with `srcStageMask` and `dstStageMask` from `vkCmdPipelineBarrier`, define the two halves of a memory dependency and an execution dependency. Memory accesses using the set of access types in `srcAccessMask` performed in pipeline stages in `srcStageMask` by the first set of commands must complete and be available to later commands. The side effects of the first set of commands will be visible to memory accesses using the set of access types in `dstAccessMask` performed in pipeline stages in `dstStageMask` by the second set of commands. If the barrier is by-region, these requirements only apply to invocations within the same framebuffer-space region, for pipeline stages that perform framebuffer-space work. The execution dependency guarantees that execution of work by the destination stages of the second set of commands will not begin until execution of work by the source stages of the first set of commands has completed.

A common type of memory dependency is to avoid a read-after-write hazard. In this case, the source access mask and stages will include writes from a particular stage, and the destination access mask and stages will indicate how those writes will be read in subsequent commands. However, barriers can also express write-after-read dependencies and write-after-write dependencies, and are even useful to express read-after-read dependencies across an image layout change.

`srcAccessMask` and `dstAccessMask` are each masks of the following bitfield:

```
typedef enum VkAccessFlagBits {
    VK_ACCESS_INDIRECT_COMMAND_READ_BIT = 0x00000001,
    VK_ACCESS_INDEX_READ_BIT = 0x00000002,
    VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT = 0x00000004,
    VK_ACCESS_UNIFORM_READ_BIT = 0x00000008,
    VK_ACCESS_INPUT_ATTACHMENT_READ_BIT = 0x00000010,
    VK_ACCESS_SHADER_READ_BIT = 0x00000020,
    VK_ACCESS_SHADER_WRITE_BIT = 0x00000040,
    VK_ACCESS_COLOR_ATTACHMENT_READ_BIT = 0x00000080,
    VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT = 0x00000100,
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT = 0x00000200,
    VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT = 0x00000400,
    VK_ACCESS_TRANSFER_READ_BIT = 0x00000800,
    VK_ACCESS_TRANSFER_WRITE_BIT = 0x00001000,
    VK_ACCESS_HOST_READ_BIT = 0x00002000,
    VK_ACCESS_HOST_WRITE_BIT = 0x00004000,
    VK_ACCESS_MEMORY_READ_BIT = 0x00008000,
    VK_ACCESS_MEMORY_WRITE_BIT = 0x00010000,
} VkAccessFlagBits;
```

`VkAccessFlagBits` has the following meanings:

- VK_ACCESS_INDIRECT_COMMAND_READ_BIT indicates that the access is an indirect command structure read as part of an indirect drawing command.
- VK_ACCESS_INDEX_READ_BIT indicates that the access is an index buffer read.
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT indicates that the access is a read via the vertex input bindings.
- VK_ACCESS_UNIFORM_READ_BIT indicates that the access is a read via a uniform buffer or dynamic uniform buffer descriptor.
- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT indicates that the access is a read via an input attachment descriptor.
- VK_ACCESS_SHADER_READ_BIT indicates that the access is a read from a shader via any other descriptor type.
- VK_ACCESS_SHADER_WRITE_BIT indicates that the access is a write or atomic from a shader via the same descriptor types as in VK_ACCESS_SHADER_READ_BIT.
- VK_ACCESS_COLOR_ATTACHMENT_READ_BIT indicates that the access is a read via a color attachment.
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT indicates that the access is a write via a color or resolve attachment.
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT indicates that the access is a read via a depth/stencil attachment.
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT indicates that the access is a write via a depth/stencil attachment.
- VK_ACCESS_TRANSFER_READ_BIT indicates that the access is a read from a transfer (copy, blit, resolve, etc.) operation. For the complete set of transfer operations, see VK_PIPELINE_STAGE_TRANSFER_BIT.

- VK_ACCESS_TRANSFER_WRITE_BIT indicates that the access is a write from a transfer (copy, blit, resolve, etc.) operation. For the complete set of transfer operations, see VK_PIPELINE_STAGE_TRANSFER_BIT.
- VK_ACCESS_HOST_READ_BIT indicates that the access is a read via the host.
- VK_ACCESS_HOST_WRITE_BIT indicates that the access is a write via the host.
- VK_ACCESS_MEMORY_READ_BIT indicates that the access is a read via a non-specific unit attached to the memory. This unit may be external to the Vulkan device or otherwise not part of the core Vulkan pipeline. When included in *dstAccessMask*, all writes using access types in *srcAccessMask* performed by pipeline stages in *srcStageMask* must be visible in memory.
- VK_ACCESS_MEMORY_WRITE_BIT indicates that the access is a write via a non-specific unit attached to the memory. This unit may be external to the Vulkan device or otherwise not part of the core Vulkan pipeline. When included in *srcAccessMask*, all access types in *dstAccessMask* from pipeline stages in *dstStageMask* will observe the side effects of commands that executed before the barrier. When included in *dstAccessMask* all writes using access types in *srcAccessMask* performed by pipeline stages in *srcStageMask* must be visible in memory.

Color attachment reads and writes are automatically (without memory or execution dependencies) coherent and ordered against themselves and each other for a given sample within a subpass of a render pass instance, executing in API order. Similarly, depth/stencil attachment reads and writes are automatically coherent and ordered against themselves and each other in the same circumstances.

Shader reads and/or writes through two variables (in the same or different shader invocations) decorated with **Coherent** and which use the same image view or buffer view are automatically coherent with each other, but require execution dependencies if a specific order is desired. Similarly, shader atomic operations are coherent with each other and with **Coherent** variables. Non-**Coherent** shader memory accesses require memory dependencies for writes to be available and reads to be visible.

Certain memory access types are only supported on queues that support a particular set of operations. The following table lists, for each access flag, which queue capability flag must be supported by the queue. When multiple flags are enumerated in the second column of the table it means that the access type is supported on the queue if it supports any of the listed capability flags. For further details on queue capabilities see Physical Device Enumeration and Queues.

Table 6.2: Supported access flags

Access flag	**Required queue capability flag**
VK_ACCESS_INDIRECT_ COMMAND_READ_BIT	VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT
VK_ACCESS_INDEX_READ_BIT	VK_QUEUE_GRAPHICS_BIT
VK_ACCESS_VERTEX_ ATTRIBUTE_READ_BIT	VK_QUEUE_GRAPHICS_BIT
VK_ACCESS_UNIFORM_READ_ BIT	VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT
VK_ACCESS_INPUT_ ATTACHMENT_READ_BIT	VK_QUEUE_GRAPHICS_BIT
VK_ACCESS_SHADER_READ_BIT	VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT
VK_ACCESS_SHADER_WRITE_ BIT	VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT
VK_ACCESS_COLOR_ ATTACHMENT_READ_BIT	VK_QUEUE_GRAPHICS_BIT

Table 6.2: (continued)

Access flag	Required queue capability flag
VK_ACCESS_COLOR_ ATTACHMENT_WRITE_BIT	VK_QUEUE_GRAPHICS_BIT
VK_ACCESS_DEPTH_STENCIL_ ATTACHMENT_READ_BIT	VK_QUEUE_GRAPHICS_BIT
VK_ACCESS_DEPTH_STENCIL_ ATTACHMENT_WRITE_BIT	VK_QUEUE_GRAPHICS_BIT
VK_ACCESS_TRANSFER_READ_ BIT	VK_QUEUE_GRAPHICS_BIT, VK_ QUEUE_COMPUTE_BIT, or VK_ QUEUE_TRANSFER_BIT
VK_ACCESS_TRANSFER_WRITE_ BIT	VK_QUEUE_GRAPHICS_BIT, VK_ QUEUE_COMPUTE_BIT, or VK_ QUEUE_TRANSFER_BIT
VK_ACCESS_HOST_READ_BIT	None
VK_ACCESS_HOST_WRITE_BIT	None
VK_ACCESS_MEMORY_READ_ BIT	None
VK_ACCESS_MEMORY_WRITE_ BIT	None

6.5.5 Buffer Memory Barriers

The buffer memory barrier type is specified with an instance of the VkBufferMemoryBarrier structure. This type of barrier only applies to memory accesses involving a specific range of the specified buffer object. That is, a memory dependency formed from a buffer memory barrier is scoped to the specified range of the buffer. It is also used to transfer ownership of a buffer range from one queue family to another, as described in the Resource Sharing section.

VkBufferMemoryBarrier has the following definition:

```
typedef struct VkBufferMemoryBarrier {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkAccessFlags                                   srcAccessMask;
    VkAccessFlags                                   dstAccessMask;
    uint32_t                                        srcQueueFamilyIndex;
    uint32_t                                        dstQueueFamilyIndex;
    VkBuffer                                        buffer;
    VkDeviceSize                                    offset;
    VkDeviceSize                                    size;
} VkBufferMemoryBarrier;
```

The members of VkBufferMemoryBarrier have the following meanings:

- *`sType`* is the type of this structure.
- *`pNext`* is `NULL` or a pointer to an extension-specific structure.
- *`srcAccessMask`* is a mask of the classes of memory accesses performed by the first set of commands that will participate in the dependency.

- *dstAccessMask* is a mask of the classes of memory accesses performed by the second set of commands that will participate in the dependency.
- *srcQueueFamilyIndex* is the queue family that is relinquishing ownership of the range of *buffer* to another queue, or VK_QUEUE_FAMILY_IGNORED if there is no transfer of ownership.
- *dstQueueFamilyIndex* is the queue family that is acquiring ownership of the range of *buffer* from another queue, or VK_QUEUE_FAMILY_IGNORED if there is no transfer of ownership.
- *buffer* is a handle to the buffer whose backing memory is affected by the barrier.
- *offset* is an offset in bytes into the backing memory for *buffer*; this is relative to the base offset as bound to the buffer (see `vkBindBufferMemory`).
- *size* is a size in bytes of the affected area of backing memory for *buffer*, or VK_WHOLE_SIZE to use the range from *offset* to the end of the buffer.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER
- *pNext* must be `NULL`
- *srcAccessMask* must be a valid combination of `VkAccessFlagBits` values
- *dstAccessMask* must be a valid combination of `VkAccessFlagBits` values
- *buffer* must be a valid VkBuffer handle
- The value of *offset* must be less than the size of *buffer*
- The sum of *offset* and *size* must be less than or equal to than the size of *buffer*
- If *buffer* was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, *srcQueueFamilyIndex* and *dstQueueFamilyIndex* must both be VK_QUEUE_FAMILY_IGNORED
- If *buffer* was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, *srcQueueFamilyIndex* and *dstQueueFamilyIndex* must either both be VK_QUEUE_FAMILY_IGNORED, or both be a valid queue family (see Section 4.3.1)
- If *buffer* was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and *srcQueueFamilyIndex* and *dstQueueFamilyIndex* are valid queue families, at least one of them must be the same as the family of the queue that will execute this barrier

6.5.6 Image Memory Barriers

The image memory barrier type is specified with an instance of the VkImageMemoryBarrier structure. This type of barrier only applies to memory accesses involving a specific subresource range of the specified image object. That is, a memory dependency formed from a image memory barrier is scoped to the specified subresources of the image. It is also used to perform a layout transition for an image subresource range, or to transfer ownership of an image subresource range from one queue family to another as described in the Resource Sharing section.

VkImageMemoryBarrier has the following definition:

```
typedef struct VkImageMemoryBarrier {
    VkStructureType                            sType;
    const void*                                pNext;
    VkAccessFlags                              srcAccessMask;
    VkAccessFlags                              dstAccessMask;
    VkImageLayout                              oldLayout;
    VkImageLayout                              newLayout;
    uint32_t                                   srcQueueFamilyIndex;
    uint32_t                                   dstQueueFamilyIndex;
    VkImage                                    image;
    VkImageSubresourceRange                    subresourceRange;
} VkImageMemoryBarrier;
```

The members of VkImageMemoryBarrier have the following meanings:

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to an extension-specific structure.
- *srcAccessMask* is a mask of the classes of memory accesses performed by the first set of commands that will participate in the dependency.
- *dstAccessMask* is a mask of the classes of memory accesses performed by the second set of commands that will participate in the dependency.
- *oldLayout* describes the current layout of the image subresource(s).
- *newLayout* describes the new layout of the image subresource(s).
- *srcQueueFamilyIndex* is the queue family that is relinquishing ownership of the image subresource(s) to another queue, or VK_QUEUE_FAMILY_IGNORED if there is no transfer of ownership).
- *dstQueueFamilyIndex* is the queue family that is acquiring ownership of the image subresource(s) from another queue, or VK_QUEUE_FAMILY_IGNORED if there is no transfer of ownership).
- *image* is a handle to the image whose backing memory is affected by the barrier.
- *subresourceRange* describes an area of the backing memory for *image* (see Section 11.5 for the description of VkImageSubresourceRange), as well as the set of subresources whose image layouts are modified.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER
- *pNext* must be NULL
- *srcAccessMask* must be a valid combination of VkAccessFlagBits values
- *dstAccessMask* must be a valid combination of VkAccessFlagBits values
- *oldLayout* must be a valid VkImageLayout value

- `newLayout` must be a valid `VkImageLayout` value
- `image` must be a valid VkImage handle
- `subresourceRange` must be a valid VkImageSubresourceRange structure
- `oldLayout` must be VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_PREINITIALIZED or the current layout of the image region affected by the barrier
- `newLayout` must not be VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED
- If `image` was created with a sharing mode of VK_SHARING_MODE_CONCURRENT, `srcQueueFamilyIndex` and `dstQueueFamilyIndex` must both be VK_QUEUE_FAMILY_IGNORED
- If `image` was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, `srcQueueFamilyIndex` and `dstQueueFamilyIndex` must either both be VK_QUEUE_FAMILY_IGNORED, or both be a valid queue family (see Section 4.3.1)
- If `image` was created with a sharing mode of VK_SHARING_MODE_EXCLUSIVE, and `srcQueueFamilyIndex` and `dstQueueFamilyIndex` are valid queue families, at least one of them must be the same as the family of the queue that will execute this barrier
- `subresourceRange` must be a valid subresource range for the image (see Section 11.5)
- If `image` has a combined depth/stencil format, then `aspectMask` must include both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT

If `oldLayout` differs from `newLayout`, a layout transition occurs as part of the image memory barrier, affecting the data contained in the region of the image defined by the `subresourceRange`. If `oldLayout` is VK_IMAGE_LAYOUT_UNDEFINED, then the data is undefined after the layout transition. This may allow a more efficient transition, since the data may be discarded. The layout transition must occur after all operations using the old layout are completed and before all operations using the new layout are started. This is achieved by ensuring that there is a memory dependency between previous accesses and the layout transition, as well as between the layout transition and subsequent accesses, where the layout transition occurs between the two halves of a memory dependency in an image memory barrier.

Layout transitions that are performed via image memory barriers are automatically ordered against other layout transitions, including those that occur as part of a render pass instance.

Note
See Section 11.4 for details on available image layouts and their usages.

6.6 Implicit Ordering Guarantees

Submitting command buffers and sparse memory operations, signaling fences, and signaling and waiting on semaphores each perform implicit memory barriers. The following guarantees are made:

After a fence or semaphore is signaled, it is guaranteed that:

- All commands in any command buffer submitted to the queue before and including the submission that signals the fence, or the batch that signals the semaphore, have completed execution.
- The side effects of these commands are available to any commands or sparse binding operations (on any queue) that follow a semaphore wait, if the semaphore they wait upon was signaled at a later time than this fence or semaphore, or that are submitted to any queue after the fence is signaled. Those side effects are also visible to the same sparse binding operations that follow the semaphore wait. If the semaphore wait is part of a `VkSubmitInfo` structure passed to `vkQueueSubmit`, they are also visible to the pipeline stages specified in the *`pWaitDstStageMask`* element corresponding to the semaphore wait, for the same commands that follow the semaphore wait. If the semaphore wait is part of a `VkSubmitInfo` structure passed to `vkQueueBindSparse`, they are visible to all stages for the same commands.
- All sparse binding operations submitted to the queue before and including the submission that signals the fence, or the batch that signals the semaphore, have completed.
- The bindings performed by these operations are available to any commands or sparse binding operations (on any queue) that follow a semaphore wait, if the semaphore they wait upon was signaled at a later time than this fence or semaphore, or that are submitted to any queue after the fence is signaled. Those bindings are also visible to the same sparse binding operations that follow the semaphore wait. If the semaphore wait is part of a `VkSubmitInfo` structure passed to `vkQueueSubmit`, they are also visible to the pipeline stages specified in the *`pWaitDstStageMask`* element corresponding to the semaphore wait, for the same commands that follows the semaphore wait. If the semaphore wait is part of a `VkSubmitInfo` structure passed to `vkQueueBindSparse`, they are visible to all stages for the same commands.
- Objects that were used in previous command buffers in this queue before the fence was signaled, or in another queue that has signaled a semaphore after using the objects and before this fence or semaphore was signaled, and which are not used in any subsequent command buffers, can be freed or destroyed, including the command buffers themselves.
- The fence can be reset or destroyed.
- The semaphore can be destroyed.

These rules define how a signal and wait operation combine to form the two halves of an implicit dependency. Signaling a fence or semaphore guarantees that previous work is complete and the effects are available to later operations. Waiting on a semaphore, waiting on a fence before submitting further work, or some combination of the two (e.g. waiting on a fence in a different queue, after using semaphores to synchronize between two queues) guarantees that the effects of the work that came before the synchronization primitive is visible to subsequent work that executes in the specified *`pWaitDstStageMask`* stages (in the case of commands following a semaphore wait as part of a `vkQueueSubmit` submission), or any stage (for all the other cases).

The rules are phrased in terms of wall clock time (*before*, *at a later time*, etc.). However, for these rules to apply, the order in wall clock time of two operations must be enforced either by:

- signaling a semaphore after the first operation and waiting on the semaphore before the second operation
- signaling a fence after the first operation, waiting on the host for the fence to be signaled, and then submitting command buffers or sparse binding operations to perform the second operation
- a combination of two or more uses of these ordering rules applied transitively.

`vkQueueWaitIdle` provides implicit ordering equivalent to having used a fence in the most recent submission on the queue and then waiting on that fence. `vkDeviceWaitIdle` provides implicit ordering equivalent to using `vkQueueWaitIdle` on all queues owned by the device.

Signaling a semaphore or fence does not guarantee that device writes are visible to the host.

When submitting batches of command buffers to a queue via `vkQueueSubmit`, it is guaranteed that:

- Host writes to mappable device memory that occured before the call to **`vkQueueSubmit`** are visible to the command buffers in that submission, if the device memory is coherent or if the memory range was flushed with `vkFlushMappedMemoryRanges`.

Chapter 7

Render Pass

A *render pass* represents a collection of attachments, subpasses, and dependencies between the subpasses, and describes how the attachments are used over the course of the subpasses. The use of a render pass in a command buffer is a *render pass instance*.

An *attachment description* describes the properties of an attachment including its format, sample count, and how its contents are treated at the beginning and end of each render pass instance.

A *subpass* represents a phase of rendering that reads and writes a subset of the attachments in a render pass. Rendering commands are recorded into a particular subpass of a render pass instance.

A *subpass description* describes the subset of attachments that is referenced by a subpass. Each subpass can read from some attachments as *input attachments*, write to some as *color attachments* or *depth/stencil attachments*, and do resolve operations to others as *resolve attachments*. A subpass can also reference a set of *preserve attachments*, which are attachments that are not read or written by the subpass but whose contents must be preserved throughout the subpass.

A subpass *uses* an attachment if it references it as a color, depth/stencil, resolve, or input attachment. A subpass does not use an attachment if it references it only as a preserve attachment. The first use of an attachment is in the lowest numbered subpass that uses that attachment. Similarly, the last use of an attachment is in the highest numbered subpass that uses that attachment.

The subpasses in a render pass all render to the same dimensions, and fragments for pixel (x,y,layer) in one subpass can only read attachment contents written by previous subpasses at that same (x,y,layer) location.

Note
By describing a complete set of subpasses a priori, render passes provide the implementation an opportunity to optimize the storage and transfer of attachment data between subpasses.
In practice, this means that subpasses with a simple framebuffer-space dependency may be merged into a single tiled rendering pass, keeping the attachment data on-chip for the duration of a render pass instance. However, it is also quite common for a render pass to only contain a single subpass.

Subpass dependencies describe ordering restrictions between pairs of subpasses. If no dependencies are specified, implementations may reorder or overlap portions (e.g., certain shader stages) of the execution of subpasses. Dependencies limit the extent of overlap or reordering, and are defined using masks of pipeline stages and memory access types. Each dependency acts as an execution and memory dependency, similarly to how pipeline barriers are defined. Dependencies are needed if two subpasses operate on attachments with overlapping ranges of the same VkDeviceMemory object and at least one subpass writes to that range.

A *subpass dependency chain* is a sequence of subpass dependencies in a render pass, where the source subpass of each subpass dependency (after the first) equals the destination subpass of the previous dependency.

A render pass describes the structure of subpasses and attachments without reference to specific image views for the attachments. The specific image views that will be used for the attachments, and their dimensions, are specified in VkFramebuffer objects. Framebuffers are created with respect to a specific render pass that the framebuffer is compatible with (see Render Pass Compatibility). Collectively, a render pass and a framebuffer define the complete render target state for one or more subpasses as well as the algorithmic dependencies between the subpasses.

The various pipeline stages of the drawing commands for a given subpass may execute concurrently and/or out of order, both within and across drawing commands. However for a given (x,y,layer,sample) sample location, certain per-sample operations are performed in API order.

7.1 Render Pass Creation

A render pass is created by calling:

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

where `device` is the device to be used to create the render pass. The resulting render pass object handle is returned in `pRenderPass`. `pCreateInfo` is a pointer to an instance of the VkRenderPassCreateInfo structure that describes the parameters of the render pass. `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- `device` must be a valid VkDevice handle
- `pCreateInfo` must be a pointer to a valid VkRenderPassCreateInfo structure
- If `pAllocator` is not NULL, `pAllocator` must be a pointer to a valid VkAllocationCallbacks structure
- `pRenderPass` must be a pointer to a VkRenderPass handle

The VkRenderPassCreateInfo structure is defined as:

```
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;
```

- *`sType`* is the type of this structure.
- *`pNext`* is `NULL` or a pointer to an extension-specific structure.
- *`flags`* is reserved for future use.
- *`attachmentCount`* is the number of attachments used by this render pass, or zero indicating no attachments. Attachments are referred to by zero-based indices in the range [0,*`attachmentCount`*).
- *`pAttachments`* points to an array of *`attachmentCount`* number of `VkAttachmentDescription` structures describing properties of the attachments, or `NULL` if *`attachmentCount`* is zero.
- *`subpassCount`* is the number of subpasses to create for this render pass. Subpasses are referred to by zero-based indices in the range [0,*`subpassCount`*). A render pass must have at least one subpass.
- *`pSubpasses`* points to an array of *`subpassCount`* number of `VkSubpassDescription` structures describing properties of the subpasses.
- *`dependencyCount`* is the number of dependencies between pairs of subpasses, or zero indicating no dependencies.
- *`pDependencies`* points to an array of *`dependencyCount`* number of `VkSubpassDependency` structures describing dependencies between pairs of subpasses, or `NULL` if *`dependencyCount`* is zero.

Valid Usage

- *`sType`* must be VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO
- *`pNext`* must be `NULL`
- *`flags`* must be `0`
- If *`attachmentCount`* is not `0`, *`pAttachments`* must be a pointer to an array of *`attachmentCount`* valid VkAttachmentDescription structures
- *`pSubpasses`* must be a pointer to an array of *`subpassCount`* valid VkSubpassDescription structures
- If *`dependencyCount`* is not `0`, *`pDependencies`* must be a pointer to an array of *`dependencyCount`* valid VkSubpassDependency structures
- The value of *`subpassCount`* must be greater than `0`
- If any two subpasses operate on attachments with overlapping ranges of the same VkDeviceMemory object, and at least one subpass writes to that area of VkDeviceMemory, a subpass dependency must be included (either directly or via some intermediate subpasses) between them
- If the *`attachment`* member of any element of *`pInputAttachments`*, *`pColorAttachments`*, *`pResolveAttachments`*, *`pDepthStencilAttachment`*, or *`pPreserveAttachments`* in any given element of *`pSubpasses`* is bound to a range of a VkDeviceMemory object that overlaps with any other attachment in any subpass (including the same subpass), the VkAttachmentReference structures describing them must include VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT in *`flags`*
- If the *`attachment`* member of any element of *`pInputAttachments`*, *`pColorAttachments`*, *`pResolveAttachments`*, *`pDepthStencilAttachment`*, or *`pPreserveAttachments`* in any given element of *`pSubpasses`* is not VK_ATTACHMENT_UNUSED, it must be less than the value of *`attachmentCount`*

- The `attachment` member of any element of the `pPreserveAttachments` member in any given element of `pSubpasses` must not be VK_ATTACHMENT_UNUSED

VkAttachmentDescription is defined as:

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

- *`format`* is a `VkFormat` value specifying the format of the image that will be used for the attachment.
- *`samples`* is the number of samples of the image as defined in `VkSampleCountFlagBits`.
- *`loadOp`* specifies how the contents of color and depth components of the attachment are treated at the beginning of the subpass where it is first used:

```
typedef enum VkAttachmentLoadOp {
    VK_ATTACHMENT_LOAD_OP_LOAD = 0,
    VK_ATTACHMENT_LOAD_OP_CLEAR = 1,
    VK_ATTACHMENT_LOAD_OP_DONT_CARE = 2,
} VkAttachmentLoadOp;
```

  – VK_ATTACHMENT_LOAD_OP_LOAD means the contents within the render area will be preserved.
  – VK_ATTACHMENT_LOAD_OP_CLEAR means the contents within the render area will be cleared to a uniform value, which is specified when a render pass instance is begun.
  – VK_ATTACHMENT_LOAD_OP_DONT_CARE means the contents within the area need not be preserved; the contents of the attachment will be undefined inside the render area.

- *`storeOp`* specifies how the contents of color and depth components of the attachment are treated at the end of the subpass where it is last used:

```
typedef enum VkAttachmentStoreOp {
    VK_ATTACHMENT_STORE_OP_STORE = 0,
    VK_ATTACHMENT_STORE_OP_DONT_CARE = 1,
} VkAttachmentStoreOp;
```

  – VK_ATTACHMENT_STORE_OP_STORE means the contents within the render area are written to memory and will be available for reading after the render pass instance completes once the writes have been synchronized with VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT (for color attachments) or VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT (for depth/stencil attachments).
  – VK_ATTACHMENT_STORE_OP_DONT_CARE means the contents within the render area are not needed after rendering, and may be discarded; the contents of the attachment will be undefined inside the render area.

- *stencilLoadOp* specifies how the contents of stencil components of the attachment are treated at the beginning of the subpass where it is first used, and must be one of the same values allowed for *loadOp* above.
- *stencilStoreOp* specifies how the contents of stencil components of the attachment are treated at the end of the last subpass where it is used, and must be one of the same values allowed for *storeOp* above.
- *initialLayout* is the layout the attachment image subresource will be in when a render pass instance begins.
- *finalLayout* is the layout the attachment image subresource will be transitioned to when a render pass instance ends. During a render pass instance, an attachment can use a different layout in each subpass, if desired.
- *flags* is a bitfield of `VkAttachmentDescriptionFlagBits` describing additional properties of the attachment:

```
typedef enum VkAttachmentDescriptionFlagBits {
    VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT = 0x00000001,
} VkAttachmentDescriptionFlagBits;
```

Valid Usage

- *flags* must be a valid combination of `VkAttachmentDescriptionFlagBits` values
- *format* must be a valid `VkFormat` value
- *samples* must be a valid `VkSampleCountFlagBits` value
- *loadOp* must be a valid `VkAttachmentLoadOp` value
- *storeOp* must be a valid `VkAttachmentStoreOp` value
- *stencilLoadOp* must be a valid `VkAttachmentLoadOp` value
- *stencilStoreOp* must be a valid `VkAttachmentStoreOp` value
- *initialLayout* must be a valid `VkImageLayout` value
- *finalLayout* must be a valid `VkImageLayout` value

If the attachment uses a color format, then *loadOp* and *storeOp* are used, and *stencilLoadOp* and *stencilStoreOp* are ignored. If the format has depth and/or stencil components, *loadOp* and *storeOp* apply only to the depth data, while *stencilLoadOp* and *stencilStoreOp* define how the stencil data is handled.

If *flags* includes VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT, then the attachment is treated as if it shares physical memory with another attachment in the same render pass. This information limits the ability of the implementation to reorder certain operations (like layout transitions and the *loadOp*) such that it is not improperly reordered against other uses of the same physical memory via a different attachment. This is described in more detail below.

If a render pass uses multiple attachments that alias the same device memory, those attachments must each include the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT bit in their attachment description flags. Attachments aliasing the same memory occurs in multiple ways:

- Multiple attachments being assigned the same image view as part of framebuffer creation.
- Attachments using distinct image views that correspond to the same subresource of an image.
- Attachments using views of distinct image subresources which are bound to overlapping memory.

Render passes must include subpass dependencies (either directly or via a subpass dependency chain) between any two subpasses that operate on the same attachment or aliasing attachments and those subpass dependencies must include execution and memory dependencies separating uses of the aliases, if at least one of those subpasses writes to one of the aliases. Those dependencies must not include the VK_DEPENDENCY_BY_REGION_BIT if the aliases are views of distinct image subresources which overlap in memory.

Multiple attachments that alias the same memory must not be used in a single subpass. A given attachment index must not be used multiple times in a single subpass, with one exception: two subpass attachments can use the same attachment index if at least one use is as an input attachment and neither use is as a resolve or preserve attachment. In other words, the same view can be used simultaneously as an input and color or depth/stencil attachment, but must not be used as multiple color or depth/stencil attachments nor as resolve or preserve attachments. This valid scenario is described in more detail below.

If a set of attachments alias each other, then all except the first to be used in the render pass must use an *initialLayout* of VK_IMAGE_LAYOUT_UNDEFINED, since the earlier uses of the other aliases make their contents undefined. Once an alias has been used and a different alias has been used after it, the first alias must not be used in any later subpasses. However, an application can assign the same image view to multiple aliasing attachment indices, which allows that image view to be used multiple times even if other aliases are used in between. Once an attachment needs the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT bit, there should be no additional cost of introducing additional aliases, and using these additional aliases may allow more efficient clearing of the attachments on multiple uses via VK_ATTACHMENT_LOAD_OP_CLEAR.

Note
The exact set of attachment indices that alias with each other is not known until a framebuffer is created using the render pass, so the above conditions cannot be validated at render pass creation time.

VkSubpassDescription is defined as:

```
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 VkAttachmentReference*                 pPreserveAttachments;
} VkSubpassDescription;
```

- *flags* is reserved for future use.
- *pipelineBindPoint* is a `VkPipelineBindPoint` value specifying whether this is a compute or graphics subpass. Currently, only graphics subpasses are supported.
- *inputAttachmentCount* is the number of input attachments.

- *pInputAttachments* is an array of `VkAttachmentReference` structures (defined below) that lists which of the render pass's attachments can be read in the shader during the subpass, and what layout the attachment images will be in during the subpass. Each element of the array corresponds to an input attachment unit number in the shader, i.e. if the shader declares an input variable "layout(input_attachment_index=X, set=Y, binding=Z)" then it uses the attachment provided in *pInputAttachments*[X]. Input attachments must also be bound to the pipeline with a descriptor set, with the input attachment descriptor written in the location (set=Y, binding=Z).
- *colorAttachmentCount* is the number of color attachments.
- *pColorAttachments* is an array of *colorAttachmentCount* `VkAttachmentReference` structures that lists which of the render pass's attachments will be used as color attachments in the subpass, and what layout the attachment images will be in during the subpass. Each element of the array correponds to a fragment shader output location, i.e. if the shader declared an output variable "layout(location=X)" then it uses the attachment provided in *pColorAttachments*[X].
- *pResolveAttachments* is `NULL` or a pointer to an array of VkAttachmentReference structures. If *pResolveAttachments* is not `NULL`, each of its elements corresponds to a color attachment (the element in *pColorAttachments* at the same index). At the end of each subpass, the subpass's color attachments are resolved to corresponding resolve attachments, unless the resolve attachment index is VK_ATTACHMENT_UNUSED or *pResolveAttachments* is `NULL`. If the first use of an attachment in a render pass is as a resolve attachment, then the *loadOp* is effectively ignored as the resolve is guaranteed to overwrite all pixels in the render area.
- *pDepthStencilAttachment* is a pointer to a `VkAttachmentReference` specifying which attachment will be used for depth/stencil data and the layout it will be in during the subpass. Setting the attachment index to VK_ ATTACHMENT_UNUSED or leaving this pointer as `NULL` indicates that no depth/stencil attachment will be used in the subpass.
- *preserveAttachmentCount* is the number of preserved attachments.
- *pPreserveAttachments* is an array of *preserveAttachmentCount* VkAttachmentReference structures describing the attachments that are not used by a subpass, but whose contents must be preserved throughout the subpass.

The contents of an attachment within the render area become undefined at the start of a subpass S if all of the following conditions are true:

- The attachment is used as a color, depth/stencil, or resolve attachment in any subpass in the render pass.
- There is a subpass S1 that uses or preserves the attachment, and a subpass dependency from S1 to S.
- The attachment is not used or preserved in subpass S.

Once the contents of an attachment become undefined in subpass S, they remain undefined for subpasses in subpass dependency chains starting with subpass S until they are written again. However, they remain valid for subpasses in other subpass dependency chains starting with subpass S1 if those subpasses use or preserve the attachment.

Valid Usage

- *flags* must be `0`
- *pipelineBindPoint* must be a valid `VkPipelineBindPoint` value

- If *inputAttachmentCount* is not `0`, *pInputAttachments* must be a pointer to an array of *inputAttachmentCount* valid VkAttachmentReference structures
- If *colorAttachmentCount* is not `0`, *pColorAttachments* must be a pointer to an array of *colorAttachmentCount* valid VkAttachmentReference structures
- If *colorAttachmentCount* is not `0`, and *pResolveAttachments* is not `NULL`, *pResolveAttachments* must be a pointer to an array of *colorAttachmentCount* valid VkAttachmentReference structures
- If *pDepthStencilAttachment* is not `NULL`, *pDepthStencilAttachment* must be a pointer to a valid VkAttachmentReference structure
- If *preserveAttachmentCount* is not `0`, *pPreserveAttachments* must be a pointer to an array of *preserveAttachmentCount* valid VkAttachmentReference structures
- *pipelineBindPoint* must be VK_PIPELINE_BIND_POINT_GRAPHICS
- The value of *colorCount* must be less than or equal to VkPhysicalDeviceLimits::*maxColorAttachments*
- If no depth/stencil attachment is used in the subpass, *pDepthStencil* must be `NULL`, or the *attachment* member of a given element of *pDepthStencil* must be VK_ATTACHMENT_UNUSED
- If the first reference to an attachment in this render pass is as an input attachment, and the attachment is not also referenced as a color or depth/stencil attachment in the same subpass, then *loadOp* must not be VK_ATTACHMENT_LOAD_OP_CLEAR
- If *pResolveAttachments* is not `NULL`, for each resolve attachment that does not have the value VK_ATTACHMENT_UNUSED, the corresponding color attachment must not have the value VK_ATTACHMENT_UNUSED
- If *pResolveAttachments* is not `NULL`, the sample count of each element of *pColorAttachments* must be anything other than VK_SAMPLE_COUNT_1_BIT
- Any given element of *pResolveAttachments* must have a sample count of VK_SAMPLE_COUNT_1_BIT
- Any given element of *pResolveAttachments* must have the same `VkFormat` as its corresponding color attachment
- All attachments referenced by *pColorAttachments* and *pDepthStencilAttachment* that are not VK_ATTACHMENT_UNUSED, must have the same sample count
- If any input attachments are VK_ATTACHMENT_UNUSED, then any pipelines bound during the subpass must not reference those input attachments

The VkAttachmentReference structure is defined as:

```
typedef struct VkAttachmentReference {
    uint32_t                                    attachment;
    VkImageLayout                               layout;
} VkAttachmentReference;
```

- *attachment* is the index of the attachment of the render pass, and corresponds to the index of the corresponding element in the *pAttachments* array of the VkRenderPassCreateInfo structure. If any color or depth/stencil attachments are VK_ATTACHMENT_UNUSED, then no writes occur for those attachments.

- *layout* is a VkImageLayout value specifying the layout the attachment uses during the subpass. The implementation will automatically perform layout transitions as needed between subpasses to make each subpass use the requested layouts.

Valid Usage

- *layout* must be a valid VkImageLayout value

The VkSubpassDependency structure is defined as:

```
typedef struct VkSubpassDependency {
    uint32_t                                        srcSubpass;
    uint32_t                                        dstSubpass;
    VkPipelineStageFlags                            srcStageMask;
    VkPipelineStageFlags                            dstStageMask;
    VkAccessFlags                                   srcAccessMask;
    VkAccessFlags                                   dstAccessMask;
    VkDependencyFlags                               dependencyFlags;
} VkSubpassDependency;
```

- *srcSubpass* and *dstSubpass* are the subpass indexes of the producer and consumer subpasses, respectively. *srcSubpass* and *dstSubpass* can also have the special value VK_SUBPASS_EXTERNAL. The source subpass must always be a lower numbered subpass than the destination subpass (excluding external subpasses), so that the order of subpass descriptions is a valid execution ordering, avoiding cycles in the dependency graph (excluding self-dependencies).
- *srcStageMask*, *dstStageMask*, *srcAccessMask*, *dstAccessMask*, and *dependencyFlags* describe a memory and execution barrier between subpasses.

Each subpass dependency defines an execution and memory dependency between two sets of commands, with the second set depending on the first set. The first set of commands is:

- All commands in the subpass indicated by *srcSubpass*, if *srcSubpass* is not VK_SUBPASS_EXTERNAL.
- All commands before the render pass instance, if *srcSubpass* is VK_SUBPASS_EXTERNAL.

The second set of commands is:

- All commands in the subpass indicated by *dstSubpass*, if *dstSubpass* is not VK_SUBPASS_EXTERNAL.
- All commands after the render pass instance, if *dstSubpass* is VK_SUBPASS_EXTERNAL.

The *srcStageMask*, *dstStageMask*, *srcAccessMask*, *dstAccessMask*, and *dependencyFlags* parameters of the dependency are interpreted the same way as for other dependencies, as described in Synchronization and Cache Control.

Valid Usage

- *srcStageMask* must be a valid combination of `VkPipelineStageFlagBits` values
- *srcStageMask* must not be `0`
- *dstStageMask* must be a valid combination of `VkPipelineStageFlagBits` values
- *dstStageMask* must not be `0`
- *srcAccessMask* must be a valid combination of `VkAccessFlagBits` values
- *dstAccessMask* must be a valid combination of `VkAccessFlagBits` values
- *dependencyFlags* must be a valid combination of `VkDependencyFlagBits` values
- If the geometry shaders feature is not enabled, *srcStageMask* must not contain *VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT*
- If the geometry shaders feature is not enabled, *dstStageMask* must not contain *VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT*
- If the tessellation shaders feature is not enabled, *srcStageMask* must not contain *VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT* or *VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT*
- If the tessellation shaders feature is not enabled, *dstStageMask* must not contain *VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT* or *VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT*
- The value of *srcSubpass* must be less than or equal to *dstSubpass*, unless one of them is VK_SUBPASS_EXTERNAL, to avoid cyclic dependencies and ensure a valid execution order
- The values of *srcSubpass* and *dstSubpass* must not both be equal to VK_SUBPASS_EXTERNAL

Automatic image layout transitions between subpasses also interact with the subpass dependencies. If two subpasses are connected by a dependency and those two subpasses use the same attachment in a different layout, then the layout transition will occur after the memory accesses via *srcAccessMask* have completed in all pipeline stages included in *srcStageMask* in the source subpass, and before any memory accesses via *dstAccessMask* occur in any pipeline stages included in *dstStageMask* in the destination subpass.

The automatic image layout transitions from *initialLayout* to the first used layout (if it is different) are performed according to the following rules:

- If the attachment does not include the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT bit and there is no subpass dependency from VK_SUBPASS_EXTERNAL to the first subpass that uses the attachment, then it is as if there were such a dependency with *srcStageMask* = *srcAccessMask* = 0 and *dstStageMask* and *dstAccessMask* including all relevant bits (all graphics pipeline stages and all access types that use image resources), with the transition executing as part of that dependency. In other words, it may overlap work before the render pass instance and is complete before the subpass begins.
- If the attachment does not include the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT bit and there is a subpass dependency from VK_SUBPASS_EXTERNAL to the first subpass that uses the attachment, then the transition executes as part of that dependency and according to its stage and access masks. It must not overlap work

that came before the render pass instance that is included in the source masks, but it may overlap work in previous subpasses.

- If the attachment includes the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT bit, then the transition executes according to all the subpass dependencies with `dstSubpass` equal to the first subpass index that the attachment is used in. That is, it occurs after all memory accesses in the source stages and masks from all the source subpasses have completed and are available, and before the union of all the destination stages begin, and the new layout is visible to the union of all the destination access types. If there are no incoming subpass dependencies, then this case follows the first rule.

Similar rules apply for the transition to the `finalLayout`, using dependencies with `dstSubpass` equal to VK_SUBPASS_EXTERNAL

If an attachment specifies the VK_ATTACHMENT_LOAD_OP_CLEAR load operation, then it will logically be cleared at the start of the first subpass where it is used.

Note
Implementations may move clears earlier as long as it does not affect the operation of a render pass instance. For example, an implementation may choose to clear all attachments at the start of the render pass instance. If an attachment has the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT flag set, then the clear must occur at the start of subpass where the attachment is first used, in order to preserve the operation of the render pass instance.

The first use of an attachment must not specify a layout equal to VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL or VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL if the attachment specifies that the `loadOp` is VK_ATTACHMENT_LOAD_OP_CLEAR. If a subpass uses the same attachment as both an input attachment and either a color attachment or a depth/stencil attachment, then both uses must observe the result of the clear.

Similarly, if an attachment specifies that the `storeOp` is VK_ATTACHMENT_STORE_OP_STORE, then it will logically be stored at the end of the last subpass where it is used.

Note
Implementations may move stores later as long as it does not affect the operation of a render pass instance. If an attachment has the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT flag set, then the store must occur at the end of the highest numbered subpass that uses the attachment.

If an attachment is not used by any subpass, then the `loadOp` and the `storeOp` are ignored and the attachment's memory contents will not be modified by execution of a render pass instance.

It will be common for a render pass to consist of a simple linear graph of dependencies, where subpass N depends on subpass N-1 for all N, and the operation of the memory barriers and layout transitions is fairly straightforward to reason about for those simple cases. But for more complex graphs, there are some rules that govern when there must be dependencies between subpasses.

As stated earlier, render passes must include subpass dependencies which (either directly or via a subpass dependency chain) separate any two subpasses that operate on the same attachment or aliasing attachments, if at least one of those subpasses writes to the attachment. If an image layout changes between those two subpasses, the implementation uses the stageMasks and accessMasks indicated by the subpass dependency as the masks that control when the layout transition must occur. If there is not a layout change on the attachment, or if an implementation treats the two layouts identically, then it may treat the dependency as a simple execution/memory barrier.

If two subpasses use the same attachment in different layouts but both uses are read-only (i.e. input attachment, or read-only depth/stencil), the application does not need to express a dependency between the two subpasses. Implementations that treat the two layouts differently may deduce and insert a dependency between the subpasses, with the implementation choosing the appropriate stage masks and access masks based on whether the attachment is used as an input or depth/stencil attachment, and may insert the appropriate layout transition along with the execution/memory barrier. Implementations that treat the two layouts identically need not insert a barrier, and the two subpasses may execute simultaneously. The stage masks and access masks are chosen as follows:

- for input attachments, stage mask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT, access mask = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT.
- for depth/stencil attachments, stage mask = VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT | VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT, access mask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT

where *srcStageMask* and *srcAccessMask* are taken based on usage in the source subpass and *dstStageMask* and *dstAccessMask* are taken based on usage in the destination subpass.

If a subpass uses the same attachment as both an input attachment and either a color attachment or a depth/stencil attachment, reads from the input attachment are not automatically coherent with writes through the color or depth/stencil attachment. In order to achieve well-defined results, one of two criteria must be satisfied. First, if the color channels or depth/stencil aspects read by the input attachment are mutually exclusive with the channels or aspects written by the color or depth/stencil attachment then there is no *feedback loop* and the reads and writes both function normally, with the reads observing values from the previous subpass(es) or from memory. This option requires the graphics pipelines used by the subpass to disable writes to color channels that are read as inputs via the *channelWriteMask*, and to disable writes to depth/stencil aspects that are read as inputs via *depthWriteEnable* or *stencilTestEnable*.

Second, if the input attachment reads channels or aspects that are written by the color or depth/stencil attachment, then there is a feedback loop and a pipeline barrier must be used between when the attachment is written and when it is subsequently read by later fragments. This pipeline barrier must follow the rules of a self-dependency as described in Subpass Self-dependency, where the barrier's flags include:

- *dstStageMask* = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
- *dstAccessMask* = VK_ACCESS_INPUT_ATTACHMENT_READ_BIT, and
- *srcAccessMask* = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT (for color attachments) or *srcAccessMask* = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT (for depth/stencil attachments).
- *srcStageMask* = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT (for color attachments) or *srcStageMask* = VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT | VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT (for depth/stencil attachments).
- *dependencyFlags* = VK_DEPENDENCY_BY_REGION_BIT.

A pipeline barrier is needed each time a fragment will read a particular (x,y,layer,sample) location if that location has been written since the most recent pipeline barrier, or since the start of the subpass if there have been no pipeline barriers since the start of the subpass.

An attachment used as both an input attachment and color attachment must be in the VK_IMAGE_LAYOUT_GENERAL layout. An attachment used as both an input attachment and depth/stencil attachment must be in either the VK_IMAGE_LAYOUT_GENERAL or VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL layout. Since an attachment in the VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL layout is read-only, this situation is not a feedback loop.

To destroy a render pass, call:

```
void vkDestroyRenderPass(
    VkDevice                                    device,
    VkRenderPass                                renderPass,
    const VkAllocationCallbacks*                pAllocator);
```

device is the device to be used to destroy the render pass. *renderPass* is the handle of the render pass to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *renderPass* is not VK_NULL_HANDLE, *renderPass* must be a valid VkRenderPass handle
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *renderPass* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *renderPass* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to *renderPass* must have completed execution
- If VkAllocationCallbacks were provided when *renderPass* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *renderPass* was created, *pAllocator* must be `NULL`

Host Synchronization

- Host access to *renderPass* must be externally synchronized

7.2 Render Pass Compatibility

Framebuffers and graphics pipelines are created with reference to a specific render pass object. They must only be used with that render pass object, or one compatible with it.

Two attachment references are compatible if they have matching format and sample count, or are both VK_ATTACHMENT_UNUSED or the pointer that would contain the reference is `NULL`.

Two arrays of attachment references are compatible if all corresponding pairs of attachments are compatible. If the arrays are of different lengths, attachment references not present in the smaller array are treated as VK_ATTACHMENT_UNUSED.

Two render passes that contain only a single subpass are compatible if their corresponding color, input, resolve, and depth/stencil attachment references are compatible.

If two render passes contain more than one subpass, they are compatible if they are identical except for:

- Initial and final image layout in attachment descriptions
- Load and store operations in attachment references
- Image layout in attachment references

A framebuffer is compatible with a render pass if it was created using the same render pass or a compatible render pass.

7.3 Framebuffers

Render passes operate in conjunction with framebuffers, which represent a collection of specific memory attachments that a render pass instance uses.

An application creates a framebuffer by calling:

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

device is the device in which the framebuffer is created. *pFramebuffer* points to the variable that will receive the VkFramebuffer handle if the call is successful. *pCreateInfo* points to a VkFramebufferCreateInfo structure which describes additional information about framebuffer creation. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- *pCreateInfo* must be a pointer to a valid VkFramebufferCreateInfo structure
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pFramebuffer* must be a pointer to a VkFramebuffer handle

The VkFramebufferCreateInfo structure is defined as:

```
typedef struct VkFramebufferCreateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkFramebufferCreateFlags                    flags;
    VkRenderPass                                renderPass;
    uint32_t                                     attachmentCount;
    const VkImageView*                           pAttachments;
    uint32_t                                     width;
    uint32_t                                     height;
    uint32_t                                     layers;
} VkFramebufferCreateInfo;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *renderPass* is a render pass that defines what render passes the framebuffer will be compatible with. See Render Pass Compatibility for details.
- *attachmentCount* is the number of attachments.
- *pAttachments* is an array of VkImageView structures, each of which will be used as the corresponding attachment in a render pass instance.
- *width*, *height* and *layers* define the dimensions of the framebuffer.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO
- *pNext* must be NULL
- *flags* must be 0
- *renderPass* must be a valid VkRenderPass handle
- If *attachmentCount* is not 0, *pAttachments* must be a pointer to an array of *attachmentCount* valid VkImageView handles
- Each of *renderPass* and the elements of *pAttachments* that are valid handles must have been created, allocated or retrieved from the same VkDevice
- The value of *attachmentCount* must be equal to the attachment count specified in *renderPass*
- Any given element of *pAttachments* that is used as a color attachment or resolve attachment by *renderPass* must have been created with a *usage* value including VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
- Any given element of *pAttachments* that is used as a depth or stencil attachment by *renderPass* must have been created with a *usage* value including VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
- Any given element of *pAttachments* that is used as an input attachment by *renderPass* must have been created with a *usage* value including VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT
- Any given element of *pAttachments* must have been created with an VkFormat value that matches the VkFormat specified by the corresponding VkAttachmentDescription in *renderPass*

- Any given element of *pAttachments* must have been created with a *samples* value that matches the *samples* value specified by the corresponding VkAttachmentDescription in *renderPass*
- Any given element of *pAttachments* must have dimensions at least as large as the corresponding framebuffer dimension
- Any given element of *pAttachments* must only specify a single mip-level
- Any given element of *pAttachments* must have been created with identity swizzle
- The value of *width* must be less than or equal to VkPhysicalDeviceLimits::*maxFramebufferWidth*
- The value of *height* must be less than or equal to VkPhysicalDeviceLimits::*maxFramebufferHeight*
- The value of *layers* must be less than or equal to VkPhysicalDeviceLimits::*maxFramebufferLayers*

Image subresources used as attachments must not be used via any non-attachment usage for the duration of a render pass instance.

Note
This restriction means that the render pass has full knowledge of all uses of all of the attachments, so that the implementation is able to make correct decisions about when and how to perform layout transitions, when to overlap execution of subpasses, etc.

It is legal for a subpass to use no color or depth/stencil attachments, and rather use shader side effects such as image stores and atomics to produce an output. In this case, the subpass continues to use the *width*, *height*, and *layers* of the framebuffer to define the dimensions of the rendering area, and the *rasterizationSamples* from each pipeline's VkPipelineMultisampleStateCreateInfo to define the number of samples used in rasterization; however, if VkPhysicalDeviceFeatures::*variableMultisampleRate* is **VK_FALSE**, then all pipelines to be bound with a given zero-attachment subpass must have the same value for VkPipelineMultisampleStateCreateInfo::*rasterizationSamples*.

To destroy a framebuffer, call:

```
void vkDestroyFramebuffer(
    VkDevice                                    device,
    VkFramebuffer                               framebuffer,
    const VkAllocationCallbacks*                pAllocator);
```

device is the device to be used to destroy the framebuffer. *framebuffer* is the handle of the framebuffer to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *framebuffer* is not VK_NULL_HANDLE, *framebuffer* must be a valid VkFramebuffer handle

- If `pAllocator` is not `NULL`, `pAllocator` must be a pointer to a valid VkAllocationCallbacks structure
- If `framebuffer` is a valid handle, it must have been created, allocated or retrieved from `device`
- Each of `device` and `framebuffer` that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to `framebuffer` must have completed execution
- If VkAllocationCallbacks were provided when `framebuffer` was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when `framebuffer` was created, `pAllocator` must be `NULL`

Host Synchronization

- Host access to `framebuffer` must be externally synchronized

7.4 Render Pass Commands

An application records the commands for a render pass instance one subpass at a time, by beginning a render pass instance, iterating over the subpasses to record commands for that subpass, and then ending the render pass instance.

To begin a render pass instance, call:

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

`commandBuffer` is the command buffer in which to record the command. `pRenderPassBegin` is a pointer to a VkRenderPassBeginInfo structure (defined below) which indicates the render pass to begin an instance of, and the framebuffer the instance uses. `contents` describes how the commands in the first subpass will be provided, and is one of the values:

```
typedef enum VkSubpassContents {
    VK_SUBPASS_CONTENTS_INLINE = 0,
    VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS = 1,
} VkSubpassContents;
```

If `contents` is VK_SUBPASS_CONTENTS_INLINE, the contents of the subpass will be recorded inline in the primary command buffer, and secondary command buffers must not be executed within the subpass. If `contents` is VK_SUBPASS_CONTENTS_SECONDARY_COMMAND_BUFFERS, the contents are recorded in secondary command buffers that will be called from the primary command buffer, and **`vkCmdExecuteCommands`** is the only valid command on the command buffer until **`vkCmdNextSubpass`** or **`vkCmdEndRenderPass`**.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *pRenderPassBegin* must be a pointer to a valid VkRenderPassBeginInfo structure
- *contents* must be a valid `VkSubpassContents` value
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- This command must only be called outside of a render pass instance
- *commandBuffer* must be a primary VkCommandBuffer

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

After beginning a render pass instance, the command buffer is ready to record the commands for the first subpass of that render pass.

The VkRenderPassBeginInfo structure is defined as:

```
typedef struct VkRenderPassBeginInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    VkRenderPass                               renderPass;
    VkFramebuffer                              framebuffer;
    VkRect2D                                   renderArea;
    uint32_t                                   clearValueCount;
    const VkClearValue*                        pClearValues;
} VkRenderPassBeginInfo;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *renderPass* is the render pass to begin an instance of.
- *framebuffer* is the framebuffer containing the attachments that are used with the render pass.
- *renderArea* is the render area that is affected by the render pass instance, and is described in more detail below.

- `clearValueCount` is the number of elements in `pClearValues`.
- `pClearValues` is an array of VkClearValue structures that contains clear values for each attachment, if the attachment uses a `loadOp` value of VK_ATTACHMENT_LOAD_OP_CLEAR. The array is indexed by attachment number, with elements corresponding to uncleared attachments being unused.

Valid Usage

- `sType` must be VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO
- `pNext` must be `NULL`
- `renderPass` must be a valid VkRenderPass handle
- `framebuffer` must be a valid VkFramebuffer handle
- If `clearValueCount` is not `0`, `pClearValues` must be a pointer to an array of `clearValueCount` VkClearValue unions
- Each of `renderPass` and `framebuffer` must have been created, allocated or retrieved from the same VkDevice
- The value of `clearValueCount` must be greater than or equal to the number of attachments in `renderPass` that specify a `loadOp` of VK_ATTACHMENT_LOAD_OP_CLEAR

`renderArea` is the render area that is affected by the render pass instance. The effects of attachment load, store and resolve operations are restricted to the pixels whose x and y coordinates fall within the render area on all attachments. The render area extends to all layers of `framebuffer`. The application must ensure (using scissor if necessary) that all rendering is contained within the render area, otherwise the pixels outside of the render area become undefined and shader side effects may or may not occur for fragments outside the render area. The render area must be contained within the framebuffer dimensions.

Note
There may be a performance cost for using a render area smaller than the framebuffer, unless it matches the render area granularity for the render pass.

The render area granularity is queried by calling:

```
void vkGetRenderAreaGranularity(
    VkDevice                                    device,
    VkRenderPass                                renderPass,
    VkExtent2D*                                 pGranularity);
```

`renderPass` is a handle to a render pass. `pGranularity` points to a VkExtent2D structure that will be filled if the call is successful.

Valid Usage

- *device* must be a valid VkDevice handle
- *renderPass* must be a valid VkRenderPass handle
- *pGranularity* must be a pointer to a VkExtent2D structure
- *renderPass* must have been created, allocated or retrieved from *device*
- Each of *device* and *renderPass* must have been created, allocated or retrieved from the same VkPhysicalDevice

The conditions leading to an optimal *renderArea* are:

- the *offset.x* member in *renderArea* is a multiple of the *width* member of the returned VkExtent2D (the horizontal granularity).
- the *offset.y* member in *renderArea* is a multiple of the *height* of the returned VkExtent2D (the vertical granularity).
- either the *offset.width* member in *renderArea* is a multiple of the horizontal granularity or *offset.x*+*offset.width* is equal to the *width* of the *framebuffer* in the VkRenderPassBeginInfo.
- either the *offset.height* member in *renderArea* is a multiple of the vertical granularity or *offset.y*+*offset.height* is equal to the *height* of the *framebuffer* in the VkRenderPassBeginInfo.

After recording the commands for a subpass, an application transitions to the next subpass in the render pass instance by calling:

```
void vkCmdNextSubpass(
    VkCommandBuffer                             commandBuffer,
    VkSubpassContents                           contents);
```

commandBuffer is the command buffer in which to record the command. The command buffer must be in the middle of recording a render pass instance, and the current subpass index must be less than the number of subpasses in the render pass minus one. The *contents* parameter has the same meaning as for **`vkCmdBeginRenderPass`**.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *contents* must be a valid `VkSubpassContents` value
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- This command must only be called inside of a render pass instance

- *commandBuffer* must be a primary VkCommandBuffer
- The current render pass instance must not be in the last subpass, that is - since the call to **vkCmdBeginRenderPass**, **vkCmdNextSubpass** must have been called a number of times at least two less than the number of subpasses in the render pass

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

The subpasses indices for a render pass begin at zero when **vkCmdBeginRenderPass** is recorded, and increments each time **vkCmdNextSubpass** is recorded.

Moving to the next subpass automatically performs any multisample resolve operations in the subpass being ended. End-of-subpass multisample resolves are treated as color attachment writes for the purposes of synchronization. That is, they are considered to execute in the VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT pipeline stage and their writes are synchronized with VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT. Synchronization between rendering within a subpass and any resolve operations at the end of the subpass occurs automatically, without need for explicit dependencies or pipeline barriers. However, if the resolve attachment is also used in a different subpass, an explicit dependency is needed.

After transitioning to the next subpass, the application can record the commands for that subpass.

After recording the commands for the last subpass, an application records a command to end a render pass instance by calling:

```
void vkCmdEndRenderPass(
    VkCommandBuffer                                 commandBuffer);
```

commandBuffer is the command buffer in which to end the current render pass instance.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- This command must only be called inside of a render pass instance
- *commandBuffer* must be a primary VkCommandBuffer
- The render pass instance that is being ended must be in the last subpass, that is - since the call to **vkCmdBeginRenderPass**, **vkCmdNextSubpass** must have been called a number of times equal to one less than the number of subpasses in the render pass

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Ending a render pass instance performs any multisample resolve operations on the final subpass.

Chapter 8

Shaders

A shader specifies programmable operations that execute for each vertex, control point, tessellated vertex, primitive, fragment, or workgroup in the corresponding stage(s) of the graphics and compute pipelines.

Graphics pipelines include vertex shader execution as a result of primitive assembly, followed, if enabled, by tessellation control and evaluation shaders operating on patches, geometry shaders, if enabled, operating on primitives, and fragment shaders, if present, operating on fragments generated by Rasterization. In this specification, vertex, tessellation control, tessellation evaluation and geometry shaders are collectively referred to as vertex processing stages and occur in the logical pipeline before rasterization. The fragment shader occurs logically after rasterization.

Only the compute shader stage is included in a compute pipeline. Compute shaders operate on compute invocations in a workgroup.

Shaders can read from input variables, and read from and write to output variables. Input and output variables can be used to transfer data between shader stages, or to allow the shader to interact with values that exist in the execution environment. Similarly, the execution environment provides constants that describe capabilities.

Shader variables are associated with execution environment-provided inputs and outputs using *built-in* decorations in the shader. The available decorations for each stage are documented in the following subsections.

8.1 Shader Modules

Shader modules contain *shader code* and one or more entry points. Shaders are selected from a shader module by specifying an entry point as part of pipeline creation. The stages of a pipeline can use shaders that come from different modules. The shader code defining a shader module must be in the SPIR-V format, as described by the Vulkan Environment for SPIR-V appendix.

A shader module is created by calling:

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

The `pCreateInfo` parameter is a pointer to an instance of the VkShaderModuleCreateInfo structure. `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- *pCreateInfo* must be a pointer to a valid VkShaderModuleCreateInfo structure
- If *pAllocator* is not NULL, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pShaderModule* must be a pointer to a VkShaderModule handle

The VkShaderModuleCreateInfo structure is defined as:

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

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *codeSize* is the size, in bytes, of the code pointed to by *pCode*.
- *pCode* points to code that is used to create the shader module. The type and format of the code is determined from the content of the memory addressed by *pCode*.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO
- *pNext* must be NULL
- *flags* must be 0
- *pCode* must be a pointer to an array of *codeSize*/4 uint32_t values
- *codeSize* must be greater than 0
- *codeSize* must be a multiple of 4
- *pCode* must point to valid SPIR-V code, formatted and packed as described by the SPIR-V Specification v1.0

- *pCode* must adhere to the validation rules described by the Validation Rules within a Module section of the SPIR-V Environment appendix
- *pCode* must declare the **Shader** capability
- *pCode* must not declare any capability that is not supported by the API, as described by the Capabilities section of the SPIR-V Environment appendix
- If *pCode* declares any of the capabilities that are listed as not required by the implementation, the relevant feature must be enabled, as listed in the SPIR-V Environment appendix

On success, the handle to the new shader module created by `vkCreateShaderModule` is placed in the variable pointed to by *pShaderModule*.

Once a shader module has been created, any entry points it contains can be used in pipeline shader stages as described in Compute Pipelines and Graphics Pipelines.

To destroy a shader module, call:

```
void vkDestroyShaderModule(
    VkDevice                                    device,
    VkShaderModule                              shaderModule,
    const VkAllocationCallbacks*                pAllocator);
```

device is the device to be used to destroy the shader module. *shaderModule* is the handle of the shader module to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

A shader module can be destroyed while pipelines created using its shaders are still in use.

Valid Usage

- *device* must be a valid VkDevice handle
- If *shaderModule* is not VK_NULL_HANDLE, *shaderModule* must be a valid VkShaderModule handle
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *shaderModule* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *shaderModule* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- If VkAllocationCallbacks were provided when *shaderModule* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *shaderModule* was created, *pAllocator* must be `NULL`

Host Synchronization

- Host access to *`shaderModule`* must be externally synchronized

8.2 Shader Execution

At each stage of the pipeline, multiple invocations of a shader may execute simultaneously. Further, invocations of a single shader produced as the result of different commands may execute simultaneously. The relative execution order of invocations of the same shader type is undefined. Shader invocations may complete in a different order than that in which the primitives they originated from were drawn or dispatched by the application. However, fragment shader outputs are written to attachments in API order.

The relative order of invocations of different shader types is largely undefined. However, when invoking a shader whose inputs are generated from a previous pipeline stage, the shader invocations from the previous stage are guaranteed to have executed far enough to generate input values for all required inputs.

8.3 Shader Memory Access Ordering

The order in which image or buffer memory is read or written by shaders is largely undefined. For some shader types (vertex, tessellation evaluation, and in some cases, fragment), even the number of shader invocations that may perform loads and stores is undefined.

In particular, the following rules apply:

- Vertex and tessellation evaluation shaders will be invoked at least once for each unique vertex, as defined in those sections.
- Fragment shaders will be invoked zero or more times, as defined in that section.
- The relative order of invocations of the same shader type are undefined. A store issued by a shader when working on primitive B might complete prior to a store for primitive A, even if primitive A is specified prior to primitive B. This applies even to fragment shaders; while fragment shader outputs are always written to the framebuffer in primitive order, stores executed by fragment shader invocations are not.
- The relative order of invocations of different shader types is largely undefined.

Note
The above limitations on shader invocation order make some forms of synchronization between shader invocations within a single set of primitives unimplementable. For example, having one invocation poll memory written by another invocation assumes that the other invocation has been launched and will complete its writes in finite time.

Stores issued to different memory locations within a single shader invocation may not be visible to other invocations in the order they were performed. The OpMemoryBarrier instruction can be used to provide stronger ordering of reads and writes performed by a single invocation. OpMemoryBarrier guarantees that any memory transactions issued by the shader invocation prior to the instruction complete prior to the memory transactions issued after the instruction. Memory barriers are needed for algorithms that require multiple invocations to access the same memory and require the operations to be performed in a partially-defined relative order. For example, if one shader invocation does a series of writes, followed by an OpMemoryBarrier instruction, followed by another write, then another invocation that sees the results of the final write will also see the previous writes. Without the memory barrier, the final write may be visible before the previous writes.

The built-in atomic memory transaction instructions can be used to read and write a given memory address atomically. While built-in atomic functions issued by multiple shader invocations are executed in undefined order relative to each other, these functions perform both a read and a write of a memory address and guarantee that no other memory transaction will write to the underlying memory between the read and write.

Note
Atomics allow shaders to use shared global addresses for mutual exclusion or as counters, among other uses.

8.4 Shader Inputs and Outputs

Data is passed into and out of shaders using variables with input or output storage class, respectively. User-defined inputs and outputs are connected between stages by matching their **`Location`** decorations. Additionally, data can be provided by or communicated to special functions provided by the execution environment using **`BuiltIn`** decorations.

In many cases, the same **`BuiltIn`** decoration can be used in multiple shader stages with similar meaning. The specific behavior of variables decorated as **`BuiltIn`** is documented in the following sections.

8.5 Vertex Shaders

Each vertex shader invocation operates on one vertex and its associated vertex attribute data, and outputs one vertex and associated data. Graphics pipelines must include a vertex shader, and the vertex shader stage is always the first shader stage in the graphics pipeline.

8.5.1 Vertex Shader Execution

A vertex shader must be executed at least once for each vertex specified by a draw command. During execution, the shader is presented with the index of the vertex and instance for which it has been invoked. Input variables declared in the vertex shader are filled by the implementation with the values of vertex attributes associated with the invocation being executed.

If a vertex is referenced by more than one input primitive, for example by including the same index value multiple times in an index buffer, the vertex shader may be invoked only once and the results shared amongst the resulting primitives. This is known as *vertex reuse*.

Implementor's Note

If a vertex is repeated in a draw command (i.e. the same index is repeated in an indexed draw), the shader may be executed anywhere from one to the number of repetitions times for that vertex, depending on the implementation's ability to reuse shader results.

8.6 Tessellation Control Shaders

The tessellation control shader is used to read an input patch provided by the application and to produce an output patch. Each tessellation control shader invocation operates on an input patch (after all control points in the patch are processed by a vertex shader) and its associated data, and outputs a single control point of the output patch and its associated data, and can also output additional per-patch data. The input patch is sized according to the *`patchControlPoints`* member of `VkPipelineTessellationStateCreateInfo`, as part of input assembly. The size of the output patch is controlled by the **`OpExecutionMode`** **`OutputVertices`** specified in the tessellation control or tessellation evaluation shaders, which must be specified in at least one of the shaders. The size of the input and output patches must each be greater than zero and less than or equal to VkPhysicalDeviceLimits::*`maxTessellationPatchSize`*.

8.6.1 Tessellation Control Shader Execution

A tessellation control shader is invoked at least once for each *output* vertex in a patch.

Inputs to the tessellation control shader are generated by the vertex shader. Each invocation of the tessellation control shader can read the attributes of any incoming vertices and their associated data. The invocations corresponding to a given patch execute logically in parallel, with undefined relative execution order. However, the **`OpControlBarrier`** instruction can be used to provide limited control of the execution order by synchronizing invocations within a patch, effectively dividing tessellation control shader execution into a set of phases. Tessellation control shaders will read undefined values if one invocation reads a per-vertex or per-patch attribute written by another invocation at any point during the same phase, or if two invocations attempt to write different values to the same per-patch output in a single phase.

8.7 Tessellation Evaluation Shaders

The Tessellation Evaluation Shader operates on an input patch of control points and their associated data, and a single input barycentric coordinate indicating the invocation's relative position within the subdivided patch, and outputs a single vertex and its associated data.

8.7.1 Tessellation Evaluation Shader Execution

A tessellation evaluation shader is invoked at least once for each unique vertex generated by the tessellator.

8.8 Geometry Shaders

The geometry shader operates on a group of vertices and their associated data assembled from a single input primitive, and emits zero or more output primitives and the group of vertices and their associated data required for each output primitive.

8.8.1 Geometry Shader Execution

A geometry shader is invoked at least once for each primitive produced by the tessellation stages, or at least once for each primitive generated by primitive assembly when tessellation is not in use. The number of geometry shader invocations per input primitive is determined from the invocation count of the geometry shader specified by the **`OpExecutionMode Invocations`** in the geometry shader. If the invocation count is not specified, then a default of one invocation is executed.

8.9 Fragment Shaders

Fragment shaders are invoked as the result of rasterization in a graphics pipeline. Each fragment shader invocation operates on a single fragment and its associated data. With few exceptions, fragment shaders do not have access to any data associated with other fragments and is considered to execute in isolation of fragment shader invocations associated with other fragments.

8.9.1 Fragment Shader Execution

For each fragment generated by rasterization, a fragment shader may or may not be invoked. A fragment shader must not be invoked if the Early Per-Fragment Tests cause it to have no coverage.

Furthermore, if it is determined that a fragment generated as the result of rasterizing a first primitive will have its outputs entirely overwritten by a fragment generated as the result of rasterizing a second primitive in the same subpass, and the fragment shader used for the fragment has no other side effects, then the fragment shader may not be executed for the fragment from the first primitive.

Relative ordering of execution of different fragment shader invocations is not defined.

The number of fragment shader invocations produced per-pixel is determined as follows:

- If per-sample shading is enabled, the fragment shader is invoked once per covered sample.
- Otherwise, the fragment shader is invoked at least once per fragment but no more than once per covered sample.

In addition to the conditions outlined above for the invocation of a fragment shader, a fragment shader invocation may be produced as a *helper invocation*. A helper invocation is a fragment shader invocation that is created solely for the purposes of evaluating derivatives for use in non-helper fragment shader invocations. Stores and atomics performed by helper invocations must not have any effect on memory, and values returned by atomic instructions in helper invocations are undefined.

8.9.2 Early Fragment Tests

An explicit control is provided to allow fragment shaders to enable early fragment tests. If the fragment shader specifies the **`EarlyFragmentTests OpExecutionMode`**, the per-fragment tests described in Early Fragment Test Mode are performed prior to fragment shader execution. Otherwise, they are performed after fragment shader execution.

8.10 Compute Shaders

Compute shaders are invoked via `vkCmdDispatch` and `vkCmdDispatchIndirect` commands. In general, they have access to similar resources as shader stages executing as part of a graphics pipeline.

Compute workloads are formed from groups of work items called *workgroups* and processed by the compute shader in the current compute pipeline. A workgroup is a collection of shader invocations that execute the same shader, potentially in parallel. Compute shaders execute in global workgroups which are divided into a number of *local workgroups* with a size that can be set by assigning a value to the **`LocalSize`** execution mode either in the shader code or via Specialization Constants. An invocation within a local workgroup can share data with other members of the local workgroup through shared variables and issue memory and control flow barriers to synchronize with other members of the local workgroup.

8.11 Interpolation Decorations

Interpolation decorations control the behavior of attribute interpolation in the fragment shader stage. Interpolation decorations can be applied to **`Input`** storage class variables in the fragment shader stage's interface, and control the interpolation behavior of those variables.

Inputs that could be interpolated can be decorated by at most one of the following decorations:

- **`Flat`**: no interpolation
- **`NoPerspective`**: linear interpolation (for lines and polygons).

Fragment input variables decorated with neither **`Flat`** nor **`NoPerspective`** use perspective-correct interpolation (for lines and polygons).

The presence of and type of interpolation is controlled by the above interpolation decorations as well as the auxiliary decorations **`Centroid`** and **`Sample`**.

A variable decorated with **`Flat`** will not be interpolated. Instead, it will have the same value for every fragment within a triangle. This value will come from a single provoking vertex. A variable decorated with **`Flat`** can also be decorated with **`Centroid`** or **`Sample`**, which will mean the same thing as decorating it only as **`Flat`**.

For fragment shader input variables decorated with neither **`Centroid`** nor **`Sample`**, the value of the assigned variable may be interpolated anywhere within the pixel and a single value may be assigned to each sample within the pixel.

`Centroid` and **`Sample`** can be used to control the location and frequency of the sampling of the decorated fragment shader input. If a fragment shader input is decorated with **`Centroid`**, a single value may be assigned to that variable for all samples in the pixel, but that value must be interpolated to a location that lies in both the pixel and in the primitive being rendered, including any of the pixel's samples covered by the primitive. Because the location at which the variable is interpolated may be different in neighboring pixels, and derivatives may be computed by computing differences between neighboring pixels, derivatives of centroid-sampled inputs may be less accurate than those for non-centroid interpolated variables. If a fragment shader input is decorated with **`Sample`**, a separate value must be assigned to that variable for each covered sample in the pixel, and that value must be sampled at the location of the individual sample. When *`rasterizationSamples`* is VK_SAMPLE_COUNT_1_BIT, the pixel center must be used for **`Centroid`**, **`Sample`**, and undecorated attribute interpolation.

Fragment shader inputs that are signed or unsigned integers, integer vectors, or any double-precision floating-point type must be decorated with **`Flat`**.

8.12 Static Use

A SPIR-V module declares a global object in memory using the **`OpVariable`** instruction, which results in a pointer **`x`** to that object. A specific entry point in a SPIR-V module is said to *statically use* that object if that entry-point's call tree contains a function that contains a memory instruction or image instruction with **`x`** as an **`id`** operand. See the *Memory Instructions* and *Image Instructions* subsections of section 3 *Binary Form* of the SPIR-V specification for the complete list of SPIR-V memory instructions.

8.13 Built-In Variables

Built-in variables are accessed in shaders by declaring a variable decorated using a **`BuiltIn`** decoration. The meaning of each **`BuiltIn`** decoration is as follows. In the remainder of this section, the name of a built-in is used interchangeably with a term equivalent to a variable decorated with that particular built-in. Built-ins that represent integer values can be declared as either signed or unsigned 32-bit integers.

`ClipDistance`

Variables decorated with the **`ClipDistance`** decoration provide the mechanism for controlling user clipping. Declared as an array, the i^{th} element of the variable decorated as **`ClipDistance`** specifies a clip distance for plane i. A clip distance of 0 means the vertex is on the plane, a positive distance means the vertex is inside the clip half-space, and a negative distance means the point is outside the clip half-space.

The **`ClipDistance`** array is explicitly sized by the shader.

The **`ClipDistance`** decoration can be applied to array inputs in tessellation control, tessellation evaluation and geometry shader stages which will contain the values written by the previous stage. It can be applied to outputs in vertex, tessellation evaluation and geometry shaders. In the last vertex processing stage, these values will be linearly interpolated across the primitive and the portion of the primitive with interpolated distances less than 0 will be considered outside the clip volume.

In the fragment shader, the **`ClipDistance`** decoration can be applied to an array of floating-point input variables and contains the linearly interpolated values described above.

`ClipDistance` must not be used in compute shaders.

`ClipDistance` must be declared as an array of 32-bit floating-point values.

`CullDistance`

A variable decorated as **`CullDistance`** provides a mechanism for a vertex processing stage to reject an entire primitive. **`CullDistance`** can be applied to an array variable. If any member of this array is assigned a negative value for all vertices belonging to a primitive, then the primitive is discarded before rasterization. **`CullDistance`** can be applied to an output variable in the last vertex processing stage (vertex, tessellation evaluation or geometry shader).

If applied to an input variable, that variable will contain the value of the corresponding output in the previous shader stage. **`CullDistance`** must not be applied to an input in the vertex shader or to an output in the fragment shader, and must not be used in compute shaders.

In fragment shaders, the values of the **`CullDistance`** array are linearly interpolated across each primitive.

`CullDistance` must be declared as an array of 32-bit floating-point values.

`FragCoord`

This variable contains the framebuffer coordinate $(x, y, z, \frac{1}{w})$ of the fragment being processed. The (x,y) coordinate (0,0) is the upper left corner of the upper left pixel in the framebuffer. The values of the x and y components of **`FragCoord`** reflect the location of the center of the pixel (i.e. fractional values of $(0.5, 0.5)$)

when sample shading is not enabled, and the location of the sample corresponding to the shader invocation when using sample shading.

The z component of **`FragCoord`** is the interpolated depth value of the primitive, and the w components is the interpolated $\frac{1}{w}$.

The **`FragCoord`** decoration is only supported in fragment shaders. The **`Centroid`** interpolation decoration is ignored on **`FragCoord`**.

`FragCoord` must be declared as a four-component vector of 32-bit floating-point values.

`FragDepth`
Writing to an output variable decorated with **`FragDepth`** from the fragment shader establishes a new depth value for all samples covered by the fragment. This value will be used for depth testing and, if the depth test passes, any subsequent write to the depth attachment. To write to **`FragDepth`**, a shader must declare the **`DepthReplacing`** execution mode. If a shader declares the **`DepthReplacing`** execution mode and there is an execution path through the shader that does not set **`FragDepth`**, then the fragment's depth value is undefined for executions of the shader that take that path. That is, if the fragment shader enables depth replacing, then it must always write it.

The **`FragDepth`** decoration is only supported in fragment shaders.

`FragDepth` must be declared as a scalar 32-bit floating-point value.

`FrontFacing`
The **`FrontFacing`** decoration can be applied to an input variable in the fragment shader. The value of this variable is non-zero if the current fragment is considered to be part of a front-facing primitive and is zero if the fragment is considered to be part of a back-facing primitive.

The **`FrontFacing`** decoration is not available to shader stages other than fragment.

`FrontFacing` must be declared as a scalar 32-bit integer.

Note
In GLSL, **`gl_FrontFacing`** is declared as a **`bool`**. To achieve similar semantics in SPIR-V, a variable of **`OpTypeBool`** can be declared and initialized as the result of the **`OpINotEqual`** operation with the operands of the **`FrontFacing`** variable and an appropriately typed constant zero.

`GlobalInvocationID`
An input variable decorated with **`GlobalInvocationID`** will contain the location of the current compute shader invocation within the global workgroup. The value in this variable is equal to the index of the local workgroup multiplied by the size of the local workgroup plus the value of **`LocalInvocationID`**.

The **`GlobalInvocationID`** decoration is only supported in compute shaders.

`GlobalInvocationID` must be declared as a three-component vector of 32-bit integers.

`HelperInvocation`
This variable is non-zero if the fragment being shaded is a helper invocation and zero otherwise. A helper invocation is an invocation of the shader that is produced to satisfy internal requirements such as the generation of derivatives.

The **`HelperInvocation`** decoration is only supported in fragment shaders.

`HelperInvocation` must be declared as a scalar 32-bit integer.

Note
It is very likely that a helper invocation will have a value of **`SampleMask`** fragment shader input value that is zero.

Note
In GLSL, **`HelperInvocation`** is declared as a **`bool`**. To achieve similar semantics in SPIR-V, a variable of **`OpTypeBool`** can be declared and initialized as the result of the **`OpINotEqual`** operation with the operands of the **`HelperInvocation`** variable and an appropriately typed constant zero.

`InvocationID`
In a geometry shader, an input variable decorated with the **`InvocationID`** decoration contains the index of the current shader invocation, which ranges from zero to the number of instances declared in the shader. If the instance count of the geometry shader is one or is not specified, then **`InvocationID`** will be zero.

In tessellation control shaders, and input variable decorated with the **`InvocationID`** decoration contains the index of the output patch vertex assigned to the tessellation control shader invocation.

The **`InvocationID`** decoration must not be used in vertex, tessellation evaluation, fragment, or compute shaders.

`InvocationID` must be declared as a scalar 32-bit integer.

`InstanceIndex`
The **`InstanceIndex`** decoration can be applied to a vertex shader input which will be filled with the index of the instance that is being processed by the current vertex shader invocation. The value of **`InstanceIndex`** begins at the value of the *`firstInstance`* parameter to `vkCmdDraw` or `vkCmdDrawIndexed` or at the value of the *`firstInstance`* member of a structure consumed by `vkCmdDrawIndirect` or `vkCmdDrawIndexedIndirect`.

The **`InstanceIndex`** decoration must not be used in any shader stage other than vertex.

`InstanceIndex` must be declared as a scalar 32-bit integer.

`Layer`
The **`Layer`** decoration can be applied to an output variable in the geometry shader that is written with the framebuffer layer index to which the primitive produced by the geometry shader will be directed. If a geometry shader entry point's interface does not include an output variable decorated with **`Layer`**, then the first layer is used. If a geometry shader entry point's interface includes an output variable decorated with **`Layer`**, it must write the same value to **`Layer`** for all output vertices of a given primitive. When used in a fragment shader, an input variable decorated with **`Layer`** contains the layer index of the primitive that the fragment invocation belongs to.

The **`Layer`** decoration is only supported in geometry and fragment shaders.

`Layer` must be declared as a scalar 32-bit integer.

`LocalInvocationID`
This variable contains the location of the current compute shader invocation within the local workgroup. The range of possible values for each component of LocalInvocationID range from zero through the size of the workgroup (as defined by **`LocalSize`**) in that dimension minus one. If the size of the workgroup in a particular dimension is one, then the value of LocalInvocationID in that dimension will be zero. That is, if the workgroup is effectively two-dimensional, then *`LocalInvocationID.z`* will be zero, and if the workgroup is one-dimensional, then the values of both *`LocalInvocationID.y`* and *`LocalInvocationID.z`* will be zero.

The **`LocalInvocationID`** decoration is only supported in compute shaders.

`LocalInvocationID` must be declared as a three-component vector of 32-bit integers.

`NumWorkGroups`
The **`NumWorkGroups`** decoration can be applied to a **`uvec3`** input variable in a compute shader, in which case it will contain the number of local workgroups that are part of the dispatch that the invocation belongs to.

It reflects the values passed to a call to `vkCmdDispatch` or through the structure referenced by `vkCmdDispatchIndirect`.

The **NumWorkGroups** decoration is only supported in compute shaders.

NumWorkGroups must be declared as a three-component vector of 32-bit integers.

PatchVertices

An input variable decorated with **PatchVertices** in the tessellation control or evaluation shader is an integer specifying the number of vertices in the input patch being processed by the shader. A single tessellation control or evaluation shader can read patches of differing sizes, so the value of the **PatchVertices** variable may differ between patches.

The **PatchVertices** decoration is only supported in tessellation control and evaluation shaders.

PatchVertices must be declared as scalar 32-bit integer.

PointCoord

During point rasterization, a variable decorated with **PointCoord** contains the coordinate of the current fragment within the point being rasterized, normalized to the size of the point with origin in the upper left corner of the point, as described in Basic Point Rasterization. If the primitive the fragment shader invocation belongs to is not a point then the value of **PointCoord** is undefined.

The **PointCoord** decoration is only supported in fragment shaders.

PointCoord must be declared as two-component vector of 32-bit floating-point values.

Note

Depending on how the point is rasterized, **PointCoord** may never reach (0,0) or (1,1).

PointSize

The **PointSize** built-in decoration is used to pass the size of point primitives between shader stages. It can be applied to inputs to tessellation control and geometry shaders. It can be applied to output variables in vertex, tessellation evaluation and geometry shaders. The value written to the variable decorated as **PointSize** by the last vertex processing stage in the pipeline is used as the framebuffer space size of points produced by rasterization. As an input, it reflects the value written to the output decorated with **PointSize** in the previous shader stage.

The **PointSize** decoration must not be applied to inputs in the vertex shader and must not be used in fragment or compute shaders.

PointSize must be declared as a scalar 32-bit floating-point value.

Position

The **Position** built-in decoration can be used on variables declared as input to tessellation control, tessellation evaluation and geometry shaders. It can be used on variables declared as outputs in the vertex, tessellation control, tessellation evaluation and geometry shaders. As an input, it contains the data written to the output variable decorated as **Position** in the previous shader stage. As an output, the data written to a variable decorated as **Position** is passed to the next shader stage. In the last vertex processing stage, the output position is used in subsequent primitive assembly, clipping and rasterization operations.

Variables decorated as **Position** must not be used as inputs in vertex shaders and are must not be used in fragment or compute shaders.

Position must be declared as a four-component vector of 32-bit floating-point values.

PrimitiveID

When the **PrimitiveID** decoration is applied to an input variable in the tessellation control shader, it will be filled with the number of patches processed since the current set of rendering primitives was started.

When the **PrimitiveID** decoration is applied to an input variable in the geometry shader, it will be filled with the number of primitives presented as input to the geometry shader since the current set of rendering primitives was started. When **PrimitiveID** is applied to an output in the geometry shader, the resulting value is seen as an input to the fragment shader.

When **PrimitiveID** is applied to an input in the fragment shader, it will be filled with the primitive index written by the geometry shader if a geometry shader is present, or with the value that would have been presented as input to the geometry shader had it been present. If a geometry shader is present and the fragment shader reads from an input variable decorated with **PrimitiveID**, then the geometry shader must write to an output variable decorated with **PrimitiveID** in all execution paths, otherwise the value of the **PrimitiveID** input in the fragment shader is undefined.

The **PrimitiveID** decoration must not be used in vertex, tessellation evaluation or compute shaders.

PrimitiveID must be declared as scalar 32-bit integer.

SampleID

The **SampleID** decoration can be applied to an integer input variable in the fragment shader. This variable will contain the zero-based index of the sample the invocation corresponds to. The value of **SampleID** ranges from zero to the number of samples in the framebuffer minus one. If a fragment shader entry point's interface includes an input variable decorated with **SampleID**, per-sample shading is enabled for draws that use that fragment shader.

SampleID is not available in shader stages other than fragment.

SampleID must be declared as a scalar 32-bit integer.

SampleMask

A fragment input variable decorated with **SampleMask** will contain a bitmask of the set of samples covered by the primitive generating the fragment during rasterization. It has a sample bit set if and only if the sample is considered covered for this fragment shader invocation. **SampleMask**[] is an array of integers. Bits are mapped to samples in a manner where bit B of mask M (`SampleMask[M]`) corresponds to sample $32 \times M + B$.

When state specifies multiple fragment shader invocations for a given fragment, the sample mask for any single fragment shader invocation specifies the subset of the covered samples for the fragment that correspond to the invocation. In this case, the bit corresponding to each covered sample will be set in exactly one fragment shader invocation.

A fragment output variable decorated with **SampleMask** is an array of integers forming a bit array in a manner similar an input variable decorated with **SampleMask**, but where each bit represents coverage as computed by the shader. Modifying the sample mask by writing zero to a bit of **SampleMask** causes the sample to be considered uncovered. However, setting sample mask bits to one will never enable samples not covered by the original primitive. If the fragment shader is being evaluated at any frequency other than per-fragment, bits of the sample mask not corresponding to the current fragment shader invocation are ignored. This array must be sized in the fragment shader either implicitly or explicitly, to be no larger than the implementation-dependent maximum sample-mask (as an array of 32-bit elements), determined by the maximum number of samples. If a fragment shader entry point's interface includes an output variable decorated with **SampleMask**, the sample mask will be undefined for any array elements of any fragment shader invocations that fail to assign a value. If a fragment shader entry point's interface does not include an output variable decorated with **SampleMask**, the sample mask has no effect on the processing of a fragment.

The **SampleMask** decoration is only supported in fragment shaders.

SampleMask must be declared as an array of 32-bit integers.

SamplePosition
This variable contains the sub-pixel position of the sample being shaded. The top left of the pixel is considered to be at coordinate (0,0) and the bottom right of the pixel is considered to be at coordinate (1,1). If a fragment shader entry point's interface includes an input variable decorated with **SamplePosition**, per-sample shading is enabled for draws that use that fragment shader.

The **SamplePosition** decoration is only supported in fragment shaders.

SamplePosition must be declared as a two-component vector of floating-point values.

TessellationCoord
The **TessellationCoord** is applied to an input variable in tessellation evaluation shaders and specifies the three-dimensional (u,v,w) barycentric coordinate of the tessellated vertex within the patch. The values of u, v, and w are in the $[0, 1]$ and vary linearly across the primitive being subdivided. For the tessellation modes of **Quads** or **IsoLines**, the third component is always zero.

The **TessellationCoord** decoration is only available to tessellation evaluation shaders.

TessellationCoord must be declared as three-component vector of 32-bit floating-point values.

TessellationLevelOuter
The **TessellationLevelOuter** decoration is used in tessellation control shaders to decorate an output variable to contain the outer tessellation factor for the resulting patch. This value is used by the tessellator to control primitive tessellation and can be read by tessellation evaluation shaders. When applied to an input variable in a tessellation evaluation shader, the shader can read the value written by the tessellation control shader.

The **TessellationLevelOuter** decoration is not available outside tessellation control and evaluation shaders.

TessellationLevelOuter must be declared as an array of size two, containing 32-bit floating-point values.

TessellationLevelInner
The **TessellationLevelInner** decoration is used in tessellation control shaders to decorate an output variable to contain the inner tessellation factor for the resulting patch. This value is used by the tessellator to control primitive tessellation and can be read by tessellation evaluation shaders. When applied to an input variable in a tessellation evaluation shader, the shader can read the value written by the tessellation control shader.

The **TessellationLevelInner** decoration is not available outside tessellation control and evaluation shaders.

TessellationLevelInner must be declared as an array of size four, containing 32-bit floating-point values.

VertexIndex
The **VertexIndex** decoration can be applied to a vertex shader input which will be filled with the index of the vertex that is being processed by the current vertex shader invocation. For non-indexed draws, the value of this variable begins at the value of the *firstVertex* parameter to `vkCmdDraw` or the *firstVertex* member of a structure consumed by `vkCmdDrawIndirect` and increments by one for each vertex in the draw. For indexed draws, its value is the content of the index buffer for the vertex plus the value of the *vertexOffset* parameter to `vkCmdDrawIndexed` or the *vertexOffset* member of the structure consumed by `vkCmdDrawIndexedIndirect`.

The value of **VertexIndex** starts at the same starting value for each instance.

The **VertexIndex** decoration must not be used in any shader stage other than vertex.

VertexIndex must be declared as a 32-bit integer.

`ViewportIndex`

The **`ViewportIndex`** decoration can be applied to an output variable in the geometry shader that is written with the viewport index to which the primitive produced by the geometry shader will be directed. The selected viewport index is used to select the viewport transform and scissor rectangle. If a geometry shader entry point's interface does not include an output variable decorated with **`ViewportIndex`**, then the first viewport is used. If a geometry shader entry point's interface includes an output variable decorated with **`ViewportIndex`**, it must write the same value to **`ViewportIndex`** for all output vertices of a given primitive. When used in a fragment shader, an input variable decorated with **`ViewportIndex`** contains the viewport index of the primitive that the fragment invocation belongs to.

The **`ViewportIndex`** decoration is only supported in geometry and fragment shaders.

`ViewportIndex` must be declared as a 32-bit integer.

`WorkgroupID`

The **`WorkgroupID`** built-in decoration can be applied to an input variable in the compute shader. It will contain a three dimensional integer index of the global workgroup that the current invocation is a member of. Each component ranges from zero to the values of the parameters passed into `vkCmdDispatch` or read from the VkDispatchIndirectCommand structure read through a call to `vkCmdDispatchIndirect`.

The **`WorkGroupID`** decoration is only supported in compute shaders.

`WorkGroupID` must be declared as a three-component vector of 32-bit integers.

Chapter 9

Pipelines

The following figure shows a block diagram of the Vulkan pipelines. Some Vulkan commands specify geometric objects to be drawn or computational work to be performed, while others specify state controlling how objects are handled by the various pipeline stages, or control data transfer between memory organized as images and buffers. Commands are effectively sent through a processing pipeline, either a *graphics pipeline* or a *compute pipeline*.

The first stage of the graphics pipeline (Input Assembler) assembles vertices to form geometric primitives such as points, lines, and triangles, based on a requested primitive topology. In the next stage (Vertex Shader) vertices can be transformed, computing positions and attributes for each vertex. If tessellation and/or geometry shaders are supported, they can then generate multiple primitives from a single input primitive, possibly changing the primitive topology or generating additional attribute data in the process.

The final resulting primitives are clipped to a clip volume in preparation for the next stage, Rasterization. The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or triangle. Each *fragment* so produced is fed to the next stage (Fragment Shader) that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect depth buffering), blending of incoming fragment colors with stored colors, as well as masking, stenciling, and other logical operations on fragment values.

Framebuffer operations read and write the color and depth/stencil attachments of the framebuffer for a given subpass of a render pass instance. The attachments can be used as input attachments in the fragment shader in a later subpass of the same render pass.

The compute pipeline is a separate pipeline from the graphics pipeline, which operates on one-, two-, or three-dimensional *work groups* which can read from and write to buffer and image memory.

This ordering is meant only as a tool for describing Vulkan, not as a strict rule of how Vulkan is implemented, and we present it only as a means to organize the various operations of the pipelines.

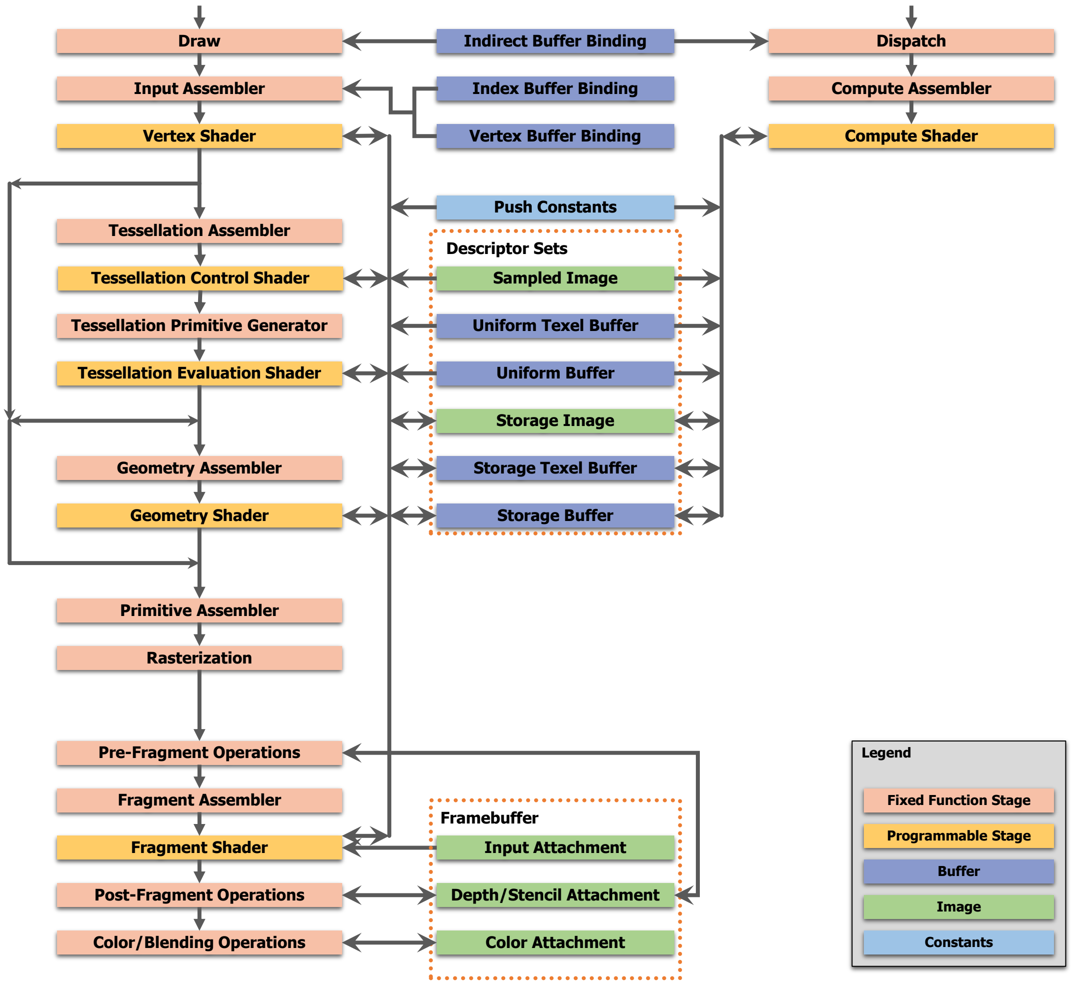


Figure 9.1: Block diagram of the Vulkan pipeline

Each pipeline is controlled by a monolithic object created from a description of all of the shader stages and any relevant fixed-function stages. Linking the whole pipeline together allows the optimization of shaders based on their input/outputs and eliminates expensive draw time state validation.

A pipeline object is bound to the device state in command buffers. Any pipeline object state that is marked as dynamic is not applied to the device state when the pipeline is bound. Dynamic state not set by binding the pipeline object can be modified at any time and persists for the lifetime of the command buffer, or until modified by another dynamic state command or another pipeline bind. No state, including dynamic state, is inherited from one command buffer to another. Only dynamic state that is required for the operations performed in the command buffer needs to be set. For example, if blending is disabled by the pipeline state then the dynamic color blend constants do not need to be specified in the command buffer, even if this state is marked as dynamic in the pipeline state object. If a new pipeline object is bound with state not marked as dynamic after a previous pipeline object with that same state as dynamic, the new pipeline object state will override the dynamic state. Modifying dynamic state that is not set as dynamic by the pipeline state object will lead to undefined results.

9.1 Compute Pipelines

Compute pipelines consist of a single static compute shader stage and the pipeline layout.

The compute pipeline encapsulates a compute shader and is created by calling **`vkCreateComputePipelines`** with *`module`* and *`pName`* selecting an entry point from a shader module, where that entry point defines a valid compute shader, in the VkPipelineShaderStageCreateInfo structure contained within the VkComputePipelineCreateInfo structure.

Compute pipelines are created by calling:

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

If *`pipelineCache`* is a valid pipeline cache object, use of that pipeline cache is enabled for the duration of the command; if it is VK_NULL_HANDLE, pipeline caching is disabled. *`createInfoCount`* is the number of VkComputePipelineCreateInfo structures in the *`pCreateInfos`* array. *`pPipelines`* is a pointer to the array of the returned compute pipeline objects. *`pAllocator`* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *`device`* must be a valid VkDevice handle
- If *`pipelineCache`* is not VK_NULL_HANDLE, *`pipelineCache`* must be a valid VkPipelineCache handle
- *`pCreateInfos`* must be a pointer to an array of *`createInfoCount`* valid VkComputePipelineCreateInfo structures
- If *`pAllocator`* is not `NULL`, *`pAllocator`* must be a pointer to a valid VkAllocationCallbacks structure
- *`pPipelines`* must be a pointer to an array of *`createInfoCount`* VkPipeline handles
- The value of *`createInfoCount`* must be greater than `0`
- If *`pipelineCache`* is a valid handle, it must have been created, allocated or retrieved from *`device`*

- Each of *device* and *pipelineCache* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- If the value of the *flags* member of any given element of *pCreateInfos* contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and the *basePipelineIndex* member of that same element is not `-1`, the value of *basePipelineIndex* must be less than the index into *pCreateInfos* that corresponds to that element

The definition of VkComputePipelineCreateInfo is:

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

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* provides options for pipeline creation, and is of type `VkPipelineCreateFlagBits`.
- *stage* is a `VkPipelineShaderStageCreateInfo` describing the compute shader.
- *layout* is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.
- *basePipelineHandle* is a pipeline to derive from
- *basePipelineIndex* is an index into the *pCreateInfos* parameter to use as a pipeline to derive from

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_COMPUTE_PIPELINE_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be a valid combination of `VkPipelineCreateFlagBits` values
- *stage* must be a valid VkPipelineShaderStageCreateInfo structure
- *layout* must be a valid VkPipelineLayout handle
- Each of *layout* and *basePipelineHandle* that are valid handles must have been created, allocated or retrieved from the same VkDevice

- If *flags* contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and *basePipelineIndex* is not `-1`, *basePipelineHandle* must be VK_NULL_HANDLE
- If *flags* contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and *basePipelineIndex* is not `-1`, it must be a valid index into the calling command's *pCreateInfos* parameter
- If *flags* contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and *basePipelineHandle* is not VK_NULL_HANDLE, *basePipelineIndex* must be `-1`
- If *flags* contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and *basePipelineHandle* is not VK_NULL_HANDLE, *basePipelineHandle* must be a valid VkPipeline handle
- If *flags* contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and *basePipelineHandle* is not VK_NULL_HANDLE, it must be a valid handle to a compute VkPipeline
- The *stage* member of *stage* must be VK_SHADER_STAGE_COMPUTE_BIT
- The shader code for the entry point identified by *stage* and the rest of the state identified by this structure must adhere to the pipeline linking rules described in the Pipeline Linking section
- *layout* must be consistent with all shaders specified in *pStages*

The parameters *basePipelineHandle* and *basePipelineIndex* are described in more detail in Pipeline Derivatives.

The parameter *stage* member of type VkPipelineShaderStageCreateInfo is:

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

The members of the VkPipelineShaderStageCreateInfo structure are as follows:

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *stage* is a `VkShaderStageFlagBits` naming the pipeline stage.
- *module* is a VkShaderModule object that contains the shader for this stage.
- *pName* is a null-terminated UTF-8 string specifying the entry point name of the shader for this stage.
- *pSpecializationInfo* is a pointer to `VkSpecializationInfo`, as described in Specialization Constants, and can be `NULL`.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO
- *pNext* must be NULL
- *flags* must be 0
- *stage* must be a valid VkShaderStageFlagBits value
- *module* must be a valid VkShaderModule handle
- *pName* must be a null-terminated string
- If *pSpecializationInfo* is not NULL, *pSpecializationInfo* must be a pointer to a valid VkSpecializationInfo structure
- If the geometry shaders feature is not enabled, *stage* must not be *VK_SHADER_STAGE_GEOMETRY_BIT*
- If the tessellation shaders feature is not enabled, *stage* must not be *VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT* or *VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT*
- *stage* must not be VK_SHADER_STAGE_ALL_GRAPHICS, or VK_SHADER_STAGE_ALL
- *pName* must be the name of an **OpEntryPoint** in *module* with an execution model that matches *stage*
- If the identified entry point includes any variable in its interface that is declared with the **ClipDistance BuiltIn** decoration, that variable must not have an array size greater than VkPhysicalDeviceLimits::*maxClipDistances*
- If the identified entry point includes any variable in its interface that is declared with the **CullDistance BuiltIn** decoration, that variable must not have an array size greater than VkPhysicalDeviceLimits::*maxCullDistances*
- If the identified entry point includes any variables in its interface that are declared with the **ClipDistance** or **CullDistance BuiltIn** decoration, those variables must not have array sizes which sum to more than VkPhysicalDeviceLimits::*maxCombinedClipAndCullDistances*
- If the identified entry point includes any variable in its interface that is declared with the **SampleMask BuiltIn** decoration, that variable must not have an array size greater than VkPhysicalDeviceLimits::*maxSampleMaskWords*
- If *stage* is VK_SHADER_STAGE_VERTEX_BIT, the identified entry point must not include any input variable in its interface that is decorated with **CullDistance**
- If *stage* is VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT or VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT, and the identified entry point has an **OpExecutionMode** instruction that specifies a patch size with **OutputVertices**, the patch size must be greater than 0 and less than or equal to VkPhysicalDeviceLimits::*maxTessellationPatchSize*
- If *stage* is VK_SHADER_STAGE_GEOMETRY_BIT, the identified entry point must have an **OpExecutionMode** instruction that specifies a maximum output vertex count that is greater than 0 and less than or equal to VkPhysicalDeviceLimits::*maxGeometryOutputVertices*
- If *stage* is VK_SHADER_STAGE_GEOMETRY_BIT, the identified entry point must have an **OpExecutionMode** instruction that specifies an invocation count that is greater than 0 and less than or equal to VkPhysicalDeviceLimits::*maxGeometryShaderInvocations*

- If *stage* is VK_SHADER_STAGE_GEOMETRY_BIT, and the identified entry point writes to **Layer** for any primitive, it must write the same value to **Layer** for all vertices of a given primitive
- If *stage* is VK_SHADER_STAGE_GEOMETRY_BIT, and the identified entry point writes to **ViewportIndex** for any primitive, it must write the same value to **ViewportIndex** for all vertices of a given primitive
- If *stage* is VK_SHADER_STAGE_FRAGMENT_BIT, the identified entry point must not include any output variables in its interface decorated with **CullDistance**
- If *stage* is VK_SHADER_STAGE_FRAGMENT_BIT, and the identified entry point writes to **FragDepth** in any execution path, it must write to **FragDepth** in all execution paths

The `VkShaderStageFlagBits` flags are defined as:

```
typedef enum VkShaderStageFlagBits {
    VK_SHADER_STAGE_VERTEX_BIT = 0x00000001,
    VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT = 0x00000002,
    VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT = 0x00000004,
    VK_SHADER_STAGE_GEOMETRY_BIT = 0x00000008,
    VK_SHADER_STAGE_FRAGMENT_BIT = 0x00000010,
    VK_SHADER_STAGE_COMPUTE_BIT = 0x00000020,
    VK_SHADER_STAGE_ALL_GRAPHICS = 0x1F,
    VK_SHADER_STAGE_ALL = 0x7FFFFFFF,
} VkShaderStageFlagBits;
```

9.2 Graphics Pipelines

Graphics pipelines consist of multiple shader stages, multiple fixed-function pipeline stages, and a pipeline layout, and are created by calling **vkCreateGraphicsPipelines**:

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

If *pipelineCache* is a valid pipeline cache object, use of that pipeline cache is enabled for the duration of the command; if it is VK_NULL_HANDLE, pipeline caching is disabled. *createInfoCount* is the number of VkGraphicsPipelineCreateInfo structures in the *pCreateInfos* array. *pPipelines* is the pointer to the array for the returned pipeline objects. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *pipelineCache* is not VK_NULL_HANDLE, *pipelineCache* must be a valid VkPipelineCache handle
- *pCreateInfos* must be a pointer to an array of *createInfoCount* valid VkGraphicsPipelineCreateInfo structures
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pPipelines* must be a pointer to an array of *createInfoCount* VkPipeline handles
- The value of *createInfoCount* must be greater than `0`
- If *pipelineCache* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *pipelineCache* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- If the value of the *flags* member of any given element of *pCreateInfos* contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and the *basePipelineIndex* member of that same element is not `-1`, the value of *basePipelineIndex* must be less than the index into *pCreateInfos* that corresponds to that element

The VkGraphicsPipelineCreateInfo structure includes an array of shader create info structures containing all the desired active shader stages, as well as creation info to define all relevant fixed-function stages, and a pipeline layout. The definition of VkGraphicsPipelineCreateInfo is:

```
typedef struct VkGraphicsPipelineCreateInfo {
    VkStructureType                              sType;
    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                             layout;
    VkRenderPass                                 renderPass;
    uint32_t                                     subpass;
    VkPipeline                                   basePipelineHandle;
    int32_t                                      basePipelineIndex;
} VkGraphicsPipelineCreateInfo;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is a bitfield of `VkPipelineCreateFlagBits` controlling how the pipeline will be generated, as described below.

- *stageCount* is the number of entries in the *pStages* array.
- *pStages* is an array of size *stageCount* structures of type VkPipelineShaderStageCreateInfo describing the set of the shader stages to be included in the graphics pipeline.
- *pVertexInputState* is a pointer to an instance of the VkPipelineVertexInputStateCreateInfo structure.
- *pInputAssemblyState* is a pointer to an instance of the VkPipelineInputAssemblyStateCreateInfo structure which determines input assembly behavior, as described in Drawing Commands.
- *pTessellationState* is a pointer to an instance of the VkPipelineTessellationStateCreateInfo structure, or NULL if the pipeline does not include a tessellation control shader stage and tessellation evaluation shader stage.
- *pViewportState* is a pointer to an instance of the VkPipelineViewportStateCreateInfo structure, or NULL if the pipeline has rasterization disabled.
- *pRasterState* is a pointer to an instance of the VkPipelineRasterizationStateCreateInfo structure.
- *pMultisampleState* is a pointer to an instance of the VkPipelineMultisampleStateCreateInfo, or NULL if the pipeline has rasterization disabled.
- *pDepthStencilState* is a pointer to an instance of the VkPipelineDepthStencilStateCreateInfo structure, or NULL if the pipeline has rasterization disabled or if the subpass of the render pass the pipeline is created against does not use a depth/stencil attachment.
- *pColorBlendState* is a pointer to an instance of the VkPipelineColorBlendStateCreateInfo structure, or NULL if the pipeline has rasterization disabled or if the subpass of the render pass the pipeline is created against does not use any color attachments.
- *pDynamicState* is a pointer to VkPipelineDynamicStateCreateInfo and is used to indicate which properties of the pipeline state object are dynamic and can be changed independently of the pipeline state. This can be NULL, which means no state in the pipeline is considered dynamic.
- *layout* is the description of binding locations used by both the pipeline and descriptor sets used with the pipeline.
- *renderPass* is a handle to a render pass object describing the environment in which the pipeline will be used; the pipeline can be used with an instance of any render pass compatible with the one provided. See Render Pass Compatibility for more information.
- *subpass* is the index of the subpass in *renderPass* where this pipeline will be used.
- *basePipelineHandle* is a pipeline to derive from.
- *basePipelineIndex* is an index into the *pCreateInfos* parameter to use as a pipeline to derive from.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO
- *pNext* must be NULL

- `flags` must be a valid combination of `VkPipelineCreateFlagBits` values
- `pStages` must be a pointer to an array of `stageCount` valid VkPipelineShaderStageCreateInfo structures
- `pVertexInputState` must be a pointer to a valid VkPipelineVertexInputStateCreateInfo structure
- `pInputAssemblyState` must be a pointer to a valid VkPipelineInputAssemblyStateCreateInfo structure
- `pRasterizationState` must be a pointer to a valid VkPipelineRasterizationStateCreateInfo structure
- If `pDynamicState` is not `NULL`, `pDynamicState` must be a pointer to a valid VkPipelineDynamicStateCreateInfo structure
- `layout` must be a valid VkPipelineLayout handle
- `renderPass` must be a valid VkRenderPass handle
- The value of `stageCount` must be greater than `0`
- Each of `layout`, `renderPass` and `basePipelineHandle` that are valid handles must have been created, allocated or retrieved from the same VkDevice
- If `flags` contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and `basePipelineIndex` is not `-1`, `basePipelineHandle` must be VK_NULL_HANDLE
- If `flags` contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and `basePipelineIndex` is not `-1`, it must be a valid index into the calling command's `pCreateInfos` parameter
- If `flags` contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and `basePipelineHandle` is not VK_NULL_HANDLE, `basePipelineIndex` must be `-1`
- If `flags` contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and `basePipelineHandle` is not VK_NULL_HANDLE, `basePipelineHandle` must be a valid VkPipeline handle
- If `flags` contains the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag, and `basePipelineHandle` is not VK_NULL_HANDLE, it must be a valid handle to a graphics VkPipeline
- `stageCount` must be greater than or equal to `1`
- The `stage` member of each element of `pStages` must be unique
- The `stage` member of one element of `pStages` must be VK_SHADER_STAGE_VERTEX_BIT
- The `stage` member of any given element of `pStages` must not be VK_SHADER_STAGE_COMPUTE_BIT
- If `pStages` includes a tessellation control shader stage, it must include a tessellation evaluation shader stage
- If `pStages` includes a tessellation evaluation shader stage, it must include a tessellation control shader stage
- If `pStages` includes a tessellation control shader stage and a tessellation evaluation shader stage, `pTessellationState` must not be `NULL`
- If `pStages` includes both a tessellation control shader stage and a tessellation evaluation shader stage, the shader code of at least one must contain an **`OpExecutionMode`** instruction that specifies the type of subdivision in the pipeline
- If `pStages` includes both a tessellation control shader stage and a tessellation evaluation shader stage, and the shader code of both contain an **`OpExecutionMode`** instruction that specifies the type of subdivision in the pipeline, they must both specify the same subdivision mode

- If `pStages` includes both a tessellation control shader stage and a tessellation evaluation shader stage, the shader code of at least one must contain an **`OpExecutionMode`** instruction that specifies the output patch size in the pipeline
- If `pStages` includes both a tessellation control shader stage and a tessellation evaluation shader stage, and the shader code of both contain an **`OpExecutionMode`** instruction that specifies the out patch size in the pipeline, they must both specify the same patch size
- If `pStages` includes tessellation shader stages, the `topology` member of `pInputAssembly` must be VK_PRIMITIVE_TOPOLOGY_PATCH_LIST
- If `pStages` includes a geometry shader stage, and doesn't include any tessellation shader stages, its shader code must contain an **`OpExecutionMode`** instruction that specifies an input primitive type that is compatible with the primitive topology specified in `pInputAssembly`
- If `pStages` includes a geometry shader stage, and also includes tessellation shader stages, its shader code must contain an **`OpExecutionMode`** instruction that specifies an input primitive type that is compatible with the primitive topology that is output by the tessellation stages
- If `pStages` includes a fragment shader stage and a geometry shader stage, and the fragment shader code reads from an input variable that is decorated with **`PrimitiveID`**, then the geometry shader code must write to a matching output variable, decorated with **`PrimitiveID`**, in all execution paths
- If `pStages` includes a fragment shader stage, its shader code must not read from any input attachment that is defined as VK_ATTACHMENT_UNUSED in `subpass`
- The shader code for the entry points identified by `pStages`, and the rest of the state identified by this structure must adhere to the pipeline linking rules described in the Pipeline Linking section
- If `subpass` references a depth attachment in `renderpass` that has a layout of VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL in the VkAttachmentReference defined by `subpass`, and `pDepthStencilState` is not `NULL`, the `depthWriteEnable` member of `pDepthStencilState` must be VK_FALSE
- If `subpass` references a stencil attachment in `renderpass` that has a layout of VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL in the VkAttachmentReference defined by `subpass`, and `pDepthStencilState` is not `NULL`, the value of the `failOp`, `passOp` and `depthFailOp` members of each of the `front` and `back` members of `pDepthStencilState` must be VK_STENCIL_OP_KEEP
- If `pColorBlendState` is not `NULL`, the value of the `blendEnable` member of each element of the `pAttachment` member of `pColorBlendState` must be VK_FALSE if the `format` of the attachment referred to in `subpass` of `renderPass` does not support color blend operations, as specified by the VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT flag in VkFormatProperties::`linearTilingFeatures` or VkFormatProperties::`optimalTilingFeatures` returned by **`vkGetPhysicalDeviceFormatProperties`**
- If `pColorBlendState` is not `NULL`, The `attachmentCount` member of `pColorBlendState` must be equal to the value of `colorAttachmentCount` used to create `subpass`
- If no element of the `pDynamicStates` member of `pDynamicState` is VK_DYNAMIC_STATE_VIEWPORT, the `pViewports` member of `pViewportState` must be a pointer to an array of `pViewportState`→viewportCount VkViewport structures
- If no element of the `pDynamicStates` member of `pDynamicState` is VK_DYNAMIC_STATE_SCISSOR, the `pScissors` member of `pViewportState` must be a pointer to an array of `pViewportState`→scissorCount VkRect2D structures

- If the wide lines feature is not enabled, and no element of the `pDynamicStates` member of `pDynamicState` is VK_DYNAMIC_STATE_LINE_WIDTH, the `lineWidth` member of `pRasterizationState` must be `1.0`
- If the `rasterizerDiscardEnable` member of `pRasterizationState` is VK_FALSE, `pViewportState` must be a pointer to a valid VkPipelineViewportStateCreateInfo structure
- If the `rasterizerDiscardEnable` member of `pRasterizationState` is VK_FALSE, `pMultisampleState` must be a pointer to a valid VkPipelineMultisampleStateCreateInfo structure
- If the `rasterizerDiscardEnable` member of `pRasterizationState` is VK_FALSE, and `subpass` includes references to depth and/or stencil attachments, `pDepthStencilState` must be a pointer to a valid VkPipelineDepthStencilStateCreateInfo structure
- If the `rasterizerDiscardEnable` member of `pRasterizationState` is VK_FALSE, and `subpass` includes references to color attachments, `pColorBlendState` must be a pointer to a valid VkPipelineColorBlendStateCreateInfo structure
- If the depth bias clamping feature is not enabled, no element of the `pDynamicStates` member of `pDynamicState` is VK_DYNAMIC_STATE_DEPTH_BIAS, and the `depthBiasEnable` member of `pDepthStencil` is VK_TRUE, the `depthBiasClamp` member of `pDepthStencil` must be `0.0`
- If no element of the `pDynamicStates` member of `pDynamicState` is VK_DYNAMIC_STATE_DEPTH_BOUNDS, and the `depthBoundsTestEnable` member of `pDepthStencil` is VK_TRUE, the value of the `minDepthBounds` and `maxDepthBounds` members of `pDepthStencil` must be between `0.0` and `1.0`, inclusive
- `layout` must be consistent with all shaders specified in `pStages`
- If `subpass` references color and/or depth/stencil attachments, then the `rasterizationSamples` member of `pMultisampleState` must be the same as the sample count for those subpass attachments
- If `subpass` does not reference any color and/or depth/stencil attachments, then the `rasterizationSamples` member of `pMultisampleState` must follow the rules for a zero-attachment subpass
- `subpass` must be a valid subpass within `renderpass`

The parameters `basePipelineHandle` and `basePipelineIndex` are described in more detail in Pipeline Derivatives.

`pStages` points to an array of `VkPipelineShaderStageCreateInfo` structures, which were previously described in Compute Pipelines.

Bits which can be set in `flags` are:

```
typedef enum VkPipelineCreateFlagBits {
    VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT = 0x00000001,
    VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT = 0x00000002,
    VK_PIPELINE_CREATE_DERIVATIVE_BIT = 0x00000004,
} VkPipelineCreateFlagBits;
```

- VK_PIPELINE_CREATE_DISABLE_OPTIMIZATION_BIT specifies that the created pipeline will not be optimized. Using this flag may reduce the time taken to create the pipeline.

- VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT specifies that the pipeline to be created is allowed to be the parent of a pipeline that will be created in a subsequent call to `vkCreateGraphicsPipelines`.
- VK_PIPELINE_CREATE_DERIVATIVE_BIT specifies that the pipeline to be created will be a child of a previously created parent pipeline.

It is valid to set both VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT and VK_PIPELINE_CREATE_DERIVATIVE_BIT. This allows a pipeline to be both a parent and possibly a child in a pipeline hierarchy. See Pipeline Derivatives for more information.

The definition of the *`pDynamicState`* member of type VkPipelineDynamicStateCreateInfo is:

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

The members of the VkPipelineDynamicStateCreateInfo structure are as follows:

- *`sType`* is the type of this structure.
- *`pNext`* is `NULL` or a pointer to an extension-specific structure.
- *`flags`* is reserved for future use.
- *`dynamicStateCount`* is the number of elements in the *`pDynamicStates`* array.
- *`pDynamicStates`* is an array of `VkDynamicState` enums which indicate which pieces of pipeline state will use the values from dynamic state commands rather than from the pipeline state creation info.

Valid Usage

- *`sType`* must be VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO
- *`pNext`* must be `NULL`
- *`flags`* must be `0`
- *`pDynamicStates`* must be a pointer to an array of *`dynamicStateCount`* valid `VkDynamicState` values
- The value of *`dynamicStateCount`* must be greater than `0`

The definition of the `VkDynamicState` enumeration is as follows:

```
typedef enum VkDynamicState {
    VK_DYNAMIC_STATE_VIEWPORT = 0,
    VK_DYNAMIC_STATE_SCISSOR = 1,
    VK_DYNAMIC_STATE_LINE_WIDTH = 2,
    VK_DYNAMIC_STATE_DEPTH_BIAS = 3,
    VK_DYNAMIC_STATE_BLEND_CONSTANTS = 4,
    VK_DYNAMIC_STATE_DEPTH_BOUNDS = 5,
    VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK = 6,
    VK_DYNAMIC_STATE_STENCIL_WRITE_MASK = 7,
    VK_DYNAMIC_STATE_STENCIL_REFERENCE = 8,
} VkDynamicState;
```

- VK_DYNAMIC_STATE_VIEWPORT indicates that the *pViewports* state in VkPipelineViewportStateCreateInfo will be ignored and must be set dynamically with `vkCmdSetViewport` before any draw commands. The number of viewports used by a pipeline is still specified by the *viewportCount* member of VkPipelineViewportStateCreateInfo.
- VK_DYNAMIC_STATE_SCISSOR indicates that the *pScissors* state in VkPipelineViewportStateCreateInfo will be ignored and must be set dynamically with `vkCmdSetScissor` before any draw commands. The number of scissor rectangles used by a pipeline is still specified by the *scissorCount* member of VkPipelineViewportStateCreateInfo.
- VK_DYNAMIC_STATE_LINE_WIDTH indicates that the *lineWidth* state in VkPipelineRasterizationStateCreateInfo will be ignored and must be set dynamically with `vkCmdSetLineWidth` before any draw commands that generate line primitives for the rasterizer.
- VK_DYNAMIC_STATE_DEPTH_BIAS indicates that the *depthBiasConstantFactor*, *depthBiasClamp* and *depthBiasSlopeFactor* states in VkPipelineRasterizationStateCreateInfo will be ignored and must be set dynamically with `vkCmdSetDepthBias` before any draws are performed with *depthBiasEnable* in VkPipelineRasterizationStateCreateInfo set to VK_TRUE.
- VK_DYNAMIC_STATE_BLEND_CONSTANTS indicates that the *blendConstants* state in VkPipelineColorBlendStateCreateInfo will be ignored and must be set dynamically with `vkCmdSetBlendConstants` before any draws are performed with a pipeline state with VkPipelineColorBlendAttachmentState member *blendEnable* set to VK_TRUE and any of the blend functions using a constant blend color.
- VK_DYNAMIC_STATE_DEPTH_BOUNDS indicates that the *minDepthBounds* and *maxDepthBounds* states of `VkPipelineDepthStencilStateCreateInfo` will be ignored and must be set dynamically with `vkCmdSetDepthBounds` before any draws are performed with a pipeline state with VkPipelineDepthStencilStateCreateInfo member *depthBoundsTestEnable* set to VK_TRUE.
- VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK indicates that the *compareMask* state in VkPipelineDepthStencilStateCreateInfo for both *front* and *back* will be ignored and must be set dynamically with `vkCmdSetStencilCompareMask` before any draws are performed with a pipeline state with VkPipelineDepthStencilStateCreateInfo member *stencilTestEnable* set to VK_TRUE
- VK_DYNAMIC_STATE_STENCIL_WRITE_MASK indicates that the *writeMask* state in VkPipelineDepthStencilStateCreateInfo for both *front* and *back* will be ignored and must be set dynamically with `vkCmdSetStencilWriteMask` before any draws are performed with a pipeline state with VkPipelineDepthStencilStateCreateInfo member *stencilTestEnable* set to VK_TRUE
- VK_DYNAMIC_STATE_STENCIL_REFERENCE indicates that the *reference* state in VkPipelineDepthStencilStateCreateInfo for both *front* and *back* will be ignored and must be set dynamically with `vkCmdSetStencilReference` before any draws are performed with a pipeline state with VkPipelineDepthStencilStateCreateInfo member *stencilTestEnable* set to VK_TRUE

If tessellation shader stages are omitted, the tessellation shading and fixed-function stages of the pipeline are skipped.

If a geometry shader is omitted, the geometry shading stage is skipped.

If a fragment shader is omitted, the results of fragment processing are undefined. Specifically, any fragment color outputs are considered to have undefined values, and the fragment depth is considered to be unmodified. This can be useful for depth-only rendering.

Presence of a shader stage in a pipeline is indicated by including a valid VkPipelineShaderStageCreateInfo with *`module`* and *`pName`* selecting an entry point from a shader module, where that entry point is valid for the stage specified by *`stage`*.

Presence of some of the fixed-function stages in the pipeline is implicitly derived from enabled shaders and provided state. For example, the fixed-function tessellator is always present when the pipeline has valid Tessellation Control and Tessellation Evaluation shaders.

FOR EXAMPLE:

- Depth-stencil only rendering in a subpass with no color attachments
  - Active Pipeline Shader Stages
    - Vertex Shader
  - Required: Fixed-Function Pipeline Stages
    - `VkPipelineVertexInputStateCreateInfo`
    - `VkPipelineInputAssemblyStateCreateInfo`
    - `VkPipelineViewportStateCreateInfo`
    - `VkPipelineRasterizationStateCreateInfo`
    - `VkPipelineMultisampleStateCreateInfo`
    - `VkPipelineDepthStencilStateCreateInfo`
- Color only rendering in a subpass with no depth/stencil attachment
  - Active Pipeline Shader Stages
    - Vertex Shader
    - Fragment Shader
  - Required: Fixed-Function Pipeline Stages
    - `VkPipelineVertexInputStateCreateInfo`
    - `VkPipelineInputAssemblyStateCreateInfo`
    - `VkPipelineViewportStateCreateInfo`
    - `VkPipelineRasterizationStateCreateInfo`
    - `VkPipelineMultisampleStateCreateInfo`
    - `VkPipelineColorBlendStateCreateInfo`
- Rendering pipeline with tessellation and geometry shaders
  - Active Pipeline Shader Stages
    - Vertex Shader
    - Tessellation Control Shader
    - Tessellation Evaluation Shader
    - Geometry Shader
    - Fragment Shader

- Required: Fixed-Function Pipeline Stages
  * `VkPipelineVertexInputStateCreateInfo`
  * `VkPipelineInputAssemblyStateCreateInfo`
  * `VkPipelineTessellationStateCreateInfo`
  * `VkPipelineViewportStateCreateInfo`
  * `VkPipelineRasterizationStateCreateInfo`
  * `VkPipelineMultisampleStateCreateInfo`
  * `VkPipelineDepthStencilStateCreateInfo`
  * `VkPipelineColorBlendStateCreateInfo`

9.3 Pipeline Linking

When a pipeline is created, the set of shaders specified in the corresponding Vk*PipelineCreateInfo structure are implicitly linked at a number of different interfaces.

- Shader Input and Output Interface
- Vertex Input Interface
- Fragment Output Interface
- Fragment Input Attachment Interface
- Shader Resource Interface

9.3.1 Shader Input and Output Interfaces

When multiple stages are present in a pipeline, the outputs of one stage form an interface with the inputs of the next stage. When such an interface involves a shader, shader outputs are matched against the inputs of the next stage, and shader inputs are matched against the outputs of the previous stage.

There are two classes of variables that can be matched between shader stages, built-in variables and user-defined variables. Each class has a different set of matching criteria. Generally, when non-shader stages are between shader stages, the user-defined variables, and most built-in variables, form an interface between the shader stages.

The variables forming the input or output *interfaces* are listed as operands to the **`OpEntryPoint`** instruction and are declared with the **`Input`** or **`Output`** storage classes, respectively, in the SPIR-V module.

9.3.1.1 Built-in Interface Block

Shader built-in variables meeting the following requirements define the *built-in interface block*. They must be:

- explicitly declared (there are no implicit built-ins),
- identified with a **`BuiltIn`** decoration,
- form object types as described in the Built-in Variables section, and
- declared in a block whose top-level members are the built-ins.

Built-ins only participate in interface matching if they are declared in such a block. They must not have any **`Location`** or **`Component`** decorations.

There must be no more than one built-in interface block per shader per interface.

9.3.1.2 User-defined Variable Interface

The remaining variables listed by `OpEntryPoint` with the `Input` or `Output` storage class form the *user-defined variable interface*. These variables must be identified with a `Location` decoration and can also be identified with a `Component` decoration.

9.3.1.3 Interface Matching

A user-defined output variable is considered to match an input variable in the subsequent stage if the two variables are declared with the same `Location` and `Component` decoration and match in type and decoration, except that interpolation decorations are not required to match. For the purposes of interface matching, variables declared without a `Component` decoration are considered to have a `Component` decoration of zero.

Variables or block members declared as structures are considered to match in type if and only if the structure members match in type, decoration, number, and declaration order. Variables or block members declared as arrays are considered to match in type only if both declarations specify the same element type and size.

Tessellation control shader per-vertex output variables and blocks, and tessellation control, tessellation evaluation, and geometry shader per-vertex input variables and blocks are required to be declared as arrays, with each element representing input or output values for a single vertex of a multi-vertex primitive. For the purposes of interface matching, the outermost array dimension of such variables and blocks is ignored.

At an interface between two non-fragment shader stages, the built-in interface block must match exactly, as described above. At an interface involving the fragment shader inputs, the presence or absence of any built-in output does not affect the interface matching.

Any input value to a shader stage is well-defined as long as the preceeding stages writes to a matching output, as described above.

Additionally, scalar and vector inputs are well-defined if there is a corresponding output satisfying all of the following conditions:

- the input and output match exactly in decoration,
- the output is a vector with the same basic type and has at least as many components as the input, and
- the common component type of the input and output is 32-bit integer or floating-point (64-bit component types are excluded).

In this case, the components of the input will be taken from the first components of the output, and any extra components of the output will be ignored.

9.3.1.4 Location Assignment

This section describes how many locations are consumed by a given type. As mentioned above, geometry shader inputs, tessellation control shader inputs and outputs, and tessellation evaluation inputs all have an additional level of arrayness relative to other shader inputs and outputs. This outer array level is removed from the type before considering how many locations the type consumes.

The `Location` value specifies an interface slot comprised of a 32-bit four-component vector conveyed between stages. The `Component` specifies components within these vector locations. Only types with widths of 32 or 64 are supported in shader interfaces.

Inputs and outputs of the following types consume a single interface location:

- 32-bit scalar and vector types, and
- 64-bit scalar and 2-component vector types.

64-bit three- and four-component vectors consume two consecutive locations.

If a declared input or output is an array of size *n* and each element takes *m* locations, it will be assigned *m* * *n* consecutive locations starting with the location specified.

If the declared input or output is an *n* x *m* 32- or 64-bit matrix, it will be assigned multiple locations starting with the location specified. The number of locations assigned for each matrix will be the same as for an *n*-element array of *m*-component vectors.

The layout of a structure type used as an **`Input`** or **`Output`** depends on whether it is also a **`Block`** (i.e. has a **`Block`** decoration).

If it is a not a **`Block`**, then the structure type must have a **`Location`** decoration. Its members are assigned consecutive locations in their declaration order, with the first member assigned to the location specified for the structure type. The members, and their nested types, must not themselves have **`Location`** decorations.

If the structure type is a **`Block`** but without a **`Location`**, then each of its members must have a **`Location`** decoration. If it is a **`Block`** with a **`Location`** decoration, then its first member is assigned to the location specified for the **`Block`**, any member with its own **`Location`** decoration is assigned that location, and otherwise each subsequent member is assigned consecutive locations in declaration order.

The locations consumed by block and structure members are determined by applying the rules above in a depth-first traversal of the instantiated members as though the structure or block member were declared as in input or output variable of the same type.

Any two inputs listed as operands on the same **`OpEntryPoint`** must not be assigned the same location, either explicitly or implicitly. Any two outputs listed as operands on the same **`OpEntryPoint`** must not be assigned the same location, either explicitly or implicitly.

The number of input and output locations available for a shader input or output interface are limited, and dependent on the shader stage as described in Table 9.1.

Table 9.1: Shader Input and Output Locations

Shader Interface	**Locations Available**
vertex input	*`maxVertexInputAttributes`*
vertex output	*`maxVertexOutputComponents`* / 4
tessellation control input	*`maxTessellationControlPerVertexInputComponents`* / 4
tessellation control output	*`maxTessellationControlPerVertexOutputComponents`* / 4
tessellation evaluation input	*`maxTessellationEvaluationInputComponents`* / 4
tessellation evaluation output	*`maxTessellationEvaluationOutputComponents`* / 4
geometry input	*`maxGeometryInputComponents`* / 4
geometry output	*`maxGeometryOutputComponents`* / 4
fragment input	*`maxFragmentInputComponents`* / 4
fragment output	*`maxFragmentOutputAttachments`*

9.3.1.5 Component Assignment

The **`Component`** decoration allows the **`Location`** to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. The components within a location are 0, 1, 2, and 3. A variable or block member starting at component N will consume components N, N+1, N+2, . . . up through its size. For single precision types, it is invalid if this sequence of components gets larger than 3. A scalar 64-bit type will consume two of these components in sequence, and a two-component 64-bit vector type will consume all four components available within a location. A three- or four-component 64-bit vector type must not specify a **`Component`** decoration. A three-component 64-bit vector type will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations.

A scalar or two-component 64-bit data type must not specify a **`Component`** decoration of 1 or 3. A **`Component`** decoration must not be specified any type that is not a scalar or vector.

9.3.2 Vertex Input Interface

When the vertex stage is present in a pipeline, the vertex shader input variables form an interface with the vertex input attributes. The vertex shader input variables are matched by the **`Location`** and **`Component`** decorations to the vertex input attributes specified in the *`pVertexInputState`* member of the `VkGraphicsPipelineCreateInfo` structure.

The vertex shader input variables listed by **`OpEntryPoint`** with the **`Input`** storage class form the *vertex input interface*. These variables must be identified with a **`Location`** decoration and can also be identified with a **`Component`** decoration.

For the purposes of interface matching: variables declared without a **`Component`** decoration are considered to have a **`Component`** decoration of zero. The number of available vertex input locations is given by the *`maxVertexInputAttributes`* member of the VkPhysicalDeviceLimits structure.

See Section 19.1.1 for details.

All vertex shader inputs declared as above must have a corresponding attribute and binding in the pipeline.

9.3.3 Fragment Output Interface

When the fragment stage is present in a pipeline, the fragment shader outputs form an interface with the output attachments of the current subpass. The fragment shader output variables are matched by the **`Location`** and **`Component`** decorations to the color attachments specified in the *`pColorAttachments`* array of the `VkSubpassDescription` structure that describes the subpass that the fragment shader is executed in.

The fragment shader output variables listed by **`OpEntryPoint`** with the **`Output`** storage class form the *fragment output interface*. These variables must be identified with a **`Location`** decoration. They can also be identified with a **`Component`** decoration and/or an **`Index`** decoration. For the purposes of interface matching: variables declared without a **`Component`** decoration are considered to have a **`Component`** decoration of zero, and variables declared without an **`Index`** decoration are considered to have an **`Index`** decoration of zero.

A fragment shader output variable identified with a **`Location`** decoration of *i* is directed to the color attachment indicated by *`pColorAttachments`*[*i*], after passing through the blending unit as described in Section 25.1, if enabled. Locations are consumed as described in Location Assignment. The number of available fragment output locations is given by the *`maxFragmentOutputAttachments`* member of the VkPhysicalDeviceLimits structure.

Components of the output variables are assigned as described in Component Assignment. Output components identified as 0, 1, 2, and 3 will be directed to the R, G, B, and A inputs to the blending unit, respectively, or to the output attachment if blending is disabled. If two variables are placed within the same location, they must have the same underlying type (floating-point or integer).

Fragment outputs identified with an **`Index`** of zero are directed to the first input of the blending unit associated with the corresponding **`Location`**. Outputs identified with an **`Index`** of one are directed to the second input of the corresponding blending unit.

No *component aliasing* of output variables is allowed, that is there must not be two output variables which have the same location, component, and index, either explicitly declared or implied.

Output values written by a fragment shader must be declared with either **`OpTypeFloat`** or **`OpTypeInt`**, and a Width of 32. Composites of these types are also permitted. If the color attachment has a signed or unsigned normalized fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in Section 2.7.1; otherwise no type conversion is applied. If the type of the values written by the fragment shader do not match the format of the corresponding color attachment, the result is undefined for those components.

9.3.4 Fragment Input Attachment Interface

When a fragment stage is present in a pipeline, the fragment shader subpass inputs form an interface with the input attachments of the current subpass. The fragment shader subpass input variables are matched by **`InputAttachmentIndex`** decorations to the input attachments specified in the *`pInputAttachments`* array of the `VkSubpassDescription` structure that describes the subpass that the fragment shader is executed in.

The fragment shader input variables listed by **`OpEntryPoint`** with the **`Input`** storage class and a decoration of **`InputAttachmentIndex`** form the *fragment input attachment interface*. These variables must be declared with a type of **`OpImageType`** and a **`Dim`** operand of **`SubpassData`**.

A fragment shader input variable identified with a **`InputAttachmentIndex`** decoration of *i* reads from the input attachment indicated by *`pInputAttachments`*[*i*]. If the input variable is declared as an array of size N, it consumes N consecutive input attachments, starting with the index specified. There must not be more than one input variable with the same **`InputAttachmentIndex`** whether explicitly declared or implied by an array declaration. The number of available input attachment indices is given by the *`maxPerStageDescriptorInputAttachments`* member of the VkPhysicalDeviceLimits structure.

Variables identified with the **`InputAttachmentIndex`** must only be used by a fragment stage. The basic data type (floating-point, integer, unsigned integer) of the subpass input must match the basic format of the corresponding input attachment, or the values of subpass loads from these variables are undefined.

See Section 13.1.11 for more details.

9.3.5 Shader Resource Interface

When a shader stage accesses buffer or image resources, as described in the Resource Descriptors section, the shader resource variables must be matched with the pipeline layout that is provided at pipeline creation time.

The set of shader resources that form the *shader resource interface* for a stage are the variables statically used by **`OpEntryPoint`** with the storage classes of **`Uniform`**, **`UniformConstant`**, and **`PushConstant`**. For the fragment shader, the variables identified by operands to **`OpEntryPoint`** with a storage class of **`Input`** and a decoration of **`InputAttachmentIndex`** are also included in this interface.

The shader resource interface can be further broken down into two sub-interfaces: the push constant interface and the descriptor set interface.

9.3.5.1 Push Constant Interface

The shader variables defined with a storage class of **PushConstant** that are statically used by the shader entry points for the pipeline define the *push constant interface*. They must be:

- typed as **OpTypeStruct**,
- identified with a **Block** decoration, and
- laid out explicitly using the **Offset**, **ArrayStride**, and **MatrixStride** decorations as specified in Offset and Stride Assignment.

There must be no more than one push constant block statically used per shader entry point.

Each variable in a push constant block must be placed at an **Offset** such that the entire constant value is entirely contained within the `VkPushConstantRange` for each **OpEntryPoint** that uses it, and the *stageFlags* for that range must specify the appropriate `VkShaderStageFlagBits` for that stage. The **Offset** decoration for any variable in a push constant block must not cause the space required for that variable to extend outside the range $[0, maxPushConstantsSize)$.

Any variable in a push constant block that is declared as an array must only be accessed with dynamically uniform indices.

9.3.5.2 Descriptor Set Interface

The *descriptor set interface* is comprised of the shader variables with the storage classes of **Uniform**, **UniformConstant**, and the variables in the fragment input attachment interface, that are statically used by the shader entry points for the pipeline.

These variables must have **DescriptorSet** and **Binding** decorations specified, which are assigned and matched with the VkDescriptorSetLayout objects in the pipeline layout as described in DescriptorSet and Binding Assignment.

Variables identified with the **UniformConstant** storage class are used only as handles to refer to opaque resources. Such variables must be typed as **OpTypeImage**, **OpTypeSampler**, **OpTypeSampledImage**, or arrays of only these types. Variables of type **OpTypeImage** must have a **Sampled** operand of 1 (sampled image) or 2 (storage image).

Any array of these types must only be indexed with constant integral expressions, except under the following conditions:

- For arrays of **OpTypeImage** variables with **Sampled** operand of 2, if the *shaderStorageImageArrayDynamicIndexing* feature is enabled and the shader module declares the **StorageImageArrayDynamicIndexing** capability, the array must only be indexed by dynamically uniform expressions.
- For arrays of **OpTypeSampler**, **OpTypeSampledImage** variables, or **OpTypeImage** variables with **Sampled** operand of 1, if the *shaderSampledImageArrayDynamicIndexing* feature is enabled and the shader module declares the **SampledImageArrayDynamicIndexing** capability, the array must only be indexed by dynamically uniform expressions.

The **Sampled Type** of an **OpTypeImage** declaration must match the same basic data type as the corresponding resource, or the values obtained by reading or sampling from this image are undefined.

The **Image Format** of an **OpTypeImage** declaration must not be **Unknown**, for variables which are used for **OpImageRead** or **OpImageWrite** operations, except under the following conditions:

- For **OpImageWrite**, if the *shaderStorageImageWriteWithoutFormat* feature is enabled and the shader module declares the **StorageImageWriteWithoutFormat** capability.
- For **OpImageRead**, if the *shaderStorageImageReadWithoutFormat* feature is enabled and the shader module declares the **StorageImageReadWithoutFormat** capability.

Variables identified with the **Uniform** storage class are used to access transparent buffer backed resources. Such variables must be:

- typed as **OpTypeStruct**, or arrays of only this type,
- identified with a **Block** or **BufferBlock** decoration, and
- laid out explicitly using the **Offset**, **ArrayStride**, and **MatrixStride** decorations as specified in Offset and Stride Assignment.

Any array of these types must only be indexed with constant integral expressions, except under the following conditions.

- For arrays of **Block** variables, if the *shaderUniformBufferArrayDynamicIndexing* feature is enabled and the shader module declares the **UniformBufferArrayDynamicIndexing** capability, the array must only be indexed by dynamically uniform expressions.
- For arrays of **BufferBlock** variables, if the *shaderStorageBufferArrayDynamicIndexing* feature is enabled and the shader module declares the **StorageBufferArrayDynamicIndexing** capability, the array must only be indexed by dynamically uniform expressions.

The **Offset** decoration for any variable in a **Block** must not cause the space required for that variable to extend outside the range $[0, maxUniformBufferRange)$. The **Offset** decoration for any variable in a **BufferBlock** must not cause the space required for that variable to extend outside the range $[0, maxStorageBufferRange)$.

Variables identified with a storage class of **Input** and a decoration of **InputAttachmentIndex** must be declared as described above.

Each shader variable declaration must refer to the same type of resource as is indicated by the *descriptorType*. See Shader Resource and Descriptor Type Correspondence for the relationship between shader declarations and descriptor types.

Table 9.2: Shader Resource and Descriptor Type Correspondence

Resource type	**Descriptor Type**	**Storage Class**	**Type**	**Decoration(s)**[1]
sampler	VK_DESCRIPTOR_ TYPE_SAMPLER	**UniformConstant**	**OpTypeSamp ler**	
sampled image	VK_DESCRIPTOR_ TYPE_SAMPLED_ IMAGE	**UniformConstant**	**OpTypeImage** (**Sampled**=1)	
storage image	VK_DESCRIPTOR_ TYPE_STORAGE_ IMAGE	**UniformConstant**	**OpTypeImage** (**Sampled**=2)	
combined image sampler	VK_DESCRIPTOR_ TYPE_COMBINED_ IMAGE_SAMPLER	**UniformConstant**	**OpTypeSample dImage**	

Table 9.2: (continued)

Resource type	Descriptor Type	Storage Class	Type	Decoration(s)[1]
uniform texel buffer	VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER	**UniformConstant**	**OpTypeImage** (**Dim**=**Buffer**, **Sampled**=1)	
storage texel buffer	VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER	**UniformConstant**	**OpTypeImage** (**Dim**=**Buffer**, **Sampled**=2)	
uniform buffer	VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC	**Uniform**	**OpTypeStruct**	**Block**, **Offset**, (**ArrayStride**), (**MatrixStride**)
storage buffer	VK_DESCRIPTOR_TYPE_STORAGE_BUFFER VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC	**Uniform**	**OpTypeStruct**	**BufferBlock**, **Offset**, (**ArrayStride**), (**MatrixStride**)
input attachment	VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT	**Input**	**OpTypeImage** (**Dim**=**SubpassData**)	**InputAttachmentIndex**

1

in addition to **DescriptorSet** and **Binding**

9.3.5.3 DescriptorSet and Binding Assignment

A variable identified with a **DescriptorSet** decoration of *s* and a **Binding** decoration of *b* indicates that this variable is associated with the `VkDescriptorSetLayoutBinding` that has a *binding* equal to *b* in *pSetLayouts*[*s*] that was specified in `VkPipelineLayoutCreateInfo`.

The range of descriptor sets is between zero and *maxBoundDescriptorSets* minus one, inclusive. If a descriptor set value is statically used by an entry point there must be an associated *pSetLayout* in the corresponding pipeline layout as described in Pipeline Layouts consistency.

If the **Binding** decoration is used with an array, the entire array is identified with that binding value. The size of the array declaration must be no larger than the *descriptorCount* of that VkDescriptorSetLayoutBinding. The index of each element of the array is referred to as the *arrayElement*. For the purposes of interface matching and descriptor set operations, if a resource variable is not an array, it is treated as if it has an arrayElement of zero.

The binding can be any 32-bit unsigned integer value, as described in Section 13.2.1. Each descriptor set has its own binding name space.

There is a limit on the number of resources of each type that can be referenced by a pipeline stage as shown in Shader Resource Limits. The "Resources Per Stage" column gives the limit on the number each type of resource that can be statically used for an entry point in any given stage in a pipeline. The "Resource Types" column lists which resource types are counted against the limit. Some resource types count against multiple limits.

If multiple entry points in the same pipeline refer to the same set and binding, all variable definitions with that **`DescriptorSet`** and **`Binding`** must have the same basic type.

Not all descriptor sets and bindings specified in a pipeline layout need to be used in a particular shader stage or pipeline, but if a **`DescriptorSet`** and **`Binding`** decoration is specified for a variable that is statically used in that shader there must be a pipeline layout entry identified with that descriptor set and *`binding`* and the corresponding *`stageFlags`* must specify the appropriate `VkShaderStageFlagBits` for that stage.

Table 9.3: Shader Resource Limits

Resources per Stage	**Resource Types**
maxPerStageDescriptorSamplers	sampler
	combined image sampler
maxPerStageDescriptorSampledImages	sampled image
	combined image sampler
	uniform texel buffer
maxPerStageDescriptorStorageImages	storage image
	storage texel buffer
maxPerStageDescriptorUniformBuffers	uniform buffer
	uniform buffer dynamic
maxPerStageDescriptorStorageBuffers	storage buffer
	storage buffer dynamic
maxPerStageDescriptorInputAttachments	input attachment[1]

1
Input attachments can only be used in the fragment shader stage

9.3.5.4 Offset and Stride Assignment

All variables with a storage class of **`PushConstant`** or **`Uniform`** must be explicitly laid out using the **`Offset`**, **`ArrayStride`**, and **`MatrixStride`** decorations. There are two different layouts requirements depending on the specific resources.

Standard Uniform Buffer Layout

Member variables of an **`OpTypeStructure`** with storage class of **`Uniform`** and a decoration of **`Block`** (uniform buffers) must be laid out according to the following rules.

- The **`Offset`** Decoration must be a multiple of its base alignment, computed recursively as follows:
  - a scalar of size *N* has a base alignment of *N*
  - a two-component vector, with components of size *N*, has a base alignment of 2*N*
  - a three- or four-component vector, with components of size *N*, has a base alignment of 4*N*
  - an array has a base alignment equal to the base alignment of its element type, rounded up to a multiple of 16
  - a structure has a base alignment equal to the largest base alignment of any of its members, rounded up to a multiple of 16
  - a row-major matrix of *C* columns has a base alignment equal to the base alignment of vector of *C* matrix components

  - a column-major matrix has a base alignment equal to the base alignment of the matrix column type

- Any **ArrayStride** or **MatrixStride** decoration must equal the base alignment of the array or matrix from above.

Note
The **std140 layout** in GLSL satisfies these rules.

Standard Storage Buffer Layout

Member variables of an **OpTypeStructure** with a storage class of **PushConstant** (push constants), or a storage class of **Uniform** with a decoration of **BufferBlock** (storage buffers) must be laid out as above, except for array and structure base alignment which do not need to be rounded up to a multiple of 16.

Note
The **std430 layout** in GLSL satisfies these rules.

9.4 Pipeline destruction

To destroy a graphics or compute pipeline, call:

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

device is the device to be used to destroy the pipeline. *pipeline* is the handle of the pipeline to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *pipeline* is not VK_NULL_HANDLE, *pipeline* must be a valid VkPipeline handle
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *pipeline* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *pipeline* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to *pipeline* must have completed execution

- If VkAllocationCallbacks were provided when *pipeline* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *pipeline* was created, *pAllocator* must be NULL

Host Synchronization

- Host access to *pipeline* must be externally synchronized

9.5 Multiple Pipeline Creation

Multiple pipelines can be created simultaneously by passing an array of VkGraphicsPipelineCreateInfo or VkComputePipelineCreateInfo structures into the `vkCreateGraphicsPipelines` and `vkCreateComputePipelines` commands, respectively. Applications can group together similar pipelines to be created in a single call, and implementations are encouraged to look for reuse opportunities within a group-create.

When an application attempts to create many pipelines in a single command, it is possible that some subset may fail creation. In that case, the corresponding entries in the *pPipelines* output array will be filled with VK_NULL_HANDLE values. If any pipeline fails creation (for example, due to out of memory errors), the **vkCreate*Pipelines** commands will return an error code. The implementation will attempt to create all pipelines, and only return VK_NULL_HANDLE values for those that actually failed.

9.6 Pipeline Derivatives

A pipeline derivative is a child pipeline created from a parent pipeline, where the child and parent are expected to have much commonality. The goal of derivative pipelines is that they be cheaper to create using the parent as a starting point, and that it be more efficient (on either host or device) to switch/bind between children of the same parent.

A derivative pipeline is created by setting the VK_PIPELINE_CREATE_DERIVATIVE_BIT flag in the Vk*PipelineCreateInfo structure. If this is set, then exactly one of *basePipelineHandle* or *basePipelineIndex* members of the structure must have a valid handle/index, and indicates the parent pipeline. If *basePipelineHandle* is used, the parent pipeline must have already been created. If *basePipelineIndex* is used, then the parent is being created in the same command. VK_NULL_HANDLE acts as the invalid handle for *basePipelineHandle*, and -1 is the invalid index for *basePipelineIndex*. If *basePipelineIndex* is used, the base pipeline must appear earlier in the array. The base pipeline must have been created with the VK_PIPELINE_CREATE_ALLOW_DERIVATIVES_BIT flag set.

9.7 Pipeline Cache

Pipeline cache objects allow the result of pipeline construction to be reused between pipelines and between runs of an application. Reuse between pipelines is achieved by passing the same pipeline cache object when creating multiple related pipelines. Reuse across runs of an application is achieved by retrieving pipeline cache contents in one run of an application, saving the contents, and using them to preinitialize a pipeline cache on a subsequent run. The contents and size of the pipeline cache objects is managed by the implementation. Applications can control the amount of data retrieved from a pipeline cache object.

Pipeline cache objects are created by calling:

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

device is the device to be used to create the pipeline cache object. The resulting pipeline cache object handle is returned in *pPipelineCache*. *pCreateInfo* is a pointer to a VkPipelineCacheCreateInfo structure that contains the initial parameters for the pipeline cache object. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- *pCreateInfo* must be a pointer to a valid VkPipelineCacheCreateInfo structure
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pPipelineCache* must be a pointer to a VkPipelineCache handle

The definition of VkPipelineCacheCreateInfo is:

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

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *initialDataSize* is the number of bytes in *pInitialData*. If *initialDataSize* is zero, the pipeline cache will initially be empty.
- *pInitialData* is a pointer to previously retrieved pipeline cache data. If the pipeline cache data is incompatible (as defined below) with the device, the pipeline cache will be initially empty. If *initialDataSize* is zero, *pInitialData* is ignored.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_PIPELINE_CACHE_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`
- If *initialDataSize* is not `0`, *pInitialData* must be a pointer to an array of *initialDataSize* bytes
- If *initialDataSize* is not `0`, it must be equal to the size of *pInitialData*, as returned by **vkGetPipelineCacheData** when *pInitialData* was originally retrieved
- If *initialDataSize* is not `0`, *pInitialData* must have been retrieved from a previous call to **vkGetPipelineCacheData**

Once created, a pipeline cache can be passed to the **vkCreateGraphicsPipelines** and **vkCreateComputePipelines** commands. If the pipeline cache passed into these commands is not VK_NULL_HANDLE, the implementation will query it for possible reuse opportunities and update it with new content. The use of the pipeline cache object in these commands is internally synchronized, and the same pipeline cache object can be used in multiple threads simultaneously.

Note
Implementations should make every effort to limit any critical sections to the actual accesses to the cache, which is expected to be significantly shorter than the duration of the **vkCreateGraphicsPipelines** and **vkCreateComputePipelines** commands.

Pipeline caches can be merged using the command:

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

device is the device on which the pipeline cache objects were created. *dstCache* is the handle of the pipeline cache to merge results into. *pSrcCaches* is an array of *srcCacheCount* pipeline cache handles, which will be merged into *dstCache*. The previous contents of *dstCache* are included after the merge.

Valid Usage

- *device* must be a valid VkDevice handle
- *dstCache* must be a valid VkPipelineCache handle

- *pSrcCaches* must be a pointer to an array of *srcCacheCount* valid VkPipelineCache handles
- The value of *srcCacheCount* must be greater than 0
- *dstCache* must have been created, allocated or retrieved from *device*
- Each element of *pSrcCaches* must have been created, allocated or retrieved from *device*
- Each of *device*, *dstCache* and the elements of *pSrcCaches* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *dstCache* must not appear in the list of source caches

Host Synchronization

- Host access to *dstCache* must be externally synchronized

Note
The details of the merge operation are implementation dependent, but implementations are recommended: to merge the contents of the specified pipelines and prune duplicate entries.

Data can be retrieved from a pipeline cache using the command:

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

device is the device on which the pipeline cache object was created, and *pipelineCache* is the handle of the pipeline cache to retrieve data from. If *pData* is NULL, then the maximum size of the data that can be retrieved from the pipeline cache, in bytes, is returned in *pDataSize*. If *pData* is not NULL, then *pDataSize* points at a variable set by the user to the size of the data buffer pointed at by *pData*, and it is overwritten with the amount of data, in bytes, actually written to *pData*. If the value of *dataSize* is less than the maximum size that can be retrieved by the pipeline cache, at most *pDataSize* bytes will be written to *pData*, and **vkGetPipelineCacheData** will return VK_INCOMPLETE. Any data written to *pData* is valid and can be provided as the *pInitialData* member of the VkPipelineCacheCreateInfo structure passed to **vkCreatePipelineCache**.

Valid Usage

- *device* must be a valid VkDevice handle
- *pipelineCache* must be a valid VkPipelineCache handle
- *pDataSize* must be a pointer to a size_t value
- If the value referenced by *pDataSize* is not `0`, and *pData* is not `NULL`, *pData* must be a pointer to an array of *pDataSize* bytes
- *pipelineCache* must have been created, allocated or retrieved from *device*
- Each of *device* and *pipelineCache* must have been created, allocated or retrieved from the same VkPhysicalDevice

Applications can store the data retrieved from the pipeline cache, and use these data, possibly in a future run of the application, to populate new pipeline cache objects. The results of pipeline compiles, however, may depend on the vendor ID, device ID, driver version, and other details of the device. To enable applications to detect when previously retrieved data is incompatible with the device, the initial bytes written to *pData* must be a header consisting of the following members:

Table 9.4: Layout for pipeline cache header version VK_PIPELINE_CACHE_HEADER_VERSION_ONE

Offset	**Size**	**Meaning**
0	4	length in bytes of the entire pipeline cache header written as a stream of bytes, with the least significant byte first
4	4	a `VkPipelineCacheHeaderVersion` value written as a stream of bytes, with the least significant byte first
8	4	a vendor ID equal to VkPhysicalDeviceProperties::*vendorID* written as a stream of bytes, with the least significant byte first
12	4	a device ID equal to VkPhysicalDeviceProperties::*deviceID* written as a stream of bytes, with the least significant byte first
16	VK_UUID_SIZE	a pipeline cache ID equal to VkPhysicalDeviceProperties::*pipelineCacheUUID*

The first four bytes encode the length of the entire pipeline header, in bytes. This value includes all fields in the header including the pipeline cache version field and the size of the length field.

The next four bytes encode the pipeline cache version. This field is a `VkPipelineCacheHeaderVersion` value. A consumer of the pipeline cache should use this value to interpret the remainder of the cache header.

If the value of *`dataSize`* is less than what is necessary to store this header, nothing will be written to *`pData`* and zero will be written to *`dataSize`*.

To destroy a pipeline cache, call:

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

`device` is the device to be used to destroy the pipeline cache. *`pipelineCache`* is the handle of the pipeline cache to destroy. *`pAllocator`* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *`device`* must be a valid VkDevice handle
- If *`pipelineCache`* is not VK_NULL_HANDLE, *`pipelineCache`* must be a valid VkPipelineCache handle
- If *`pAllocator`* is not `NULL`, *`pAllocator`* must be a pointer to a valid VkAllocationCallbacks structure
- If *`pipelineCache`* is a valid handle, it must have been created, allocated or retrieved from *`device`*
- Each of *`device`* and *`pipelineCache`* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- If VkAllocationCallbacks were provided when *`pipelineCache`* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *`pipelineCache`* was created, *`pAllocator`* must be `NULL`

Host Synchronization

- Host access to *`pipelineCache`* must be externally synchronized

9.8 Specialization Constants

Specialization constants are a mechanism whereby constants in a SPIR-V module can have their constant value specified at the time the VkPipeline is created. This allows a SPIR-V module to have constants that can be modified while executing an application that uses the Vulkan API.

Note
Specialization constants are useful to allow a compute shader to have its work group size changed at runtime by the user, for example.

Each instance of the VkPipelineShaderStageCreateInfo structure contains a parameter *pSpecializationInfo*, which can be NULL to indicate no specialization constants. The definition of the VkSpecializationInfo structure is:

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

The members of VkSpecializationInfo are as follows:

- *mapEntryCount* is the number of entries in the *pMapEntries* array.
- *pMapEntries* is a pointer to an array of VkSpecializationMapEntry which maps constant IDs to offsets in *pData*.
- *dataSize* is the byte size of the *pData* buffer.
- *pData* contains the actual constant values to specialize with.

Valid Usage

- If *mapEntryCount* is not 0, *pMapEntries* must be a pointer to an array of *mapEntryCount* VkSpecializationMapEntry structures
- If *dataSize* is not 0, *pData* must be a pointer to an array of *dataSize* bytes
- The *offset* member of any given element of *pMapEntries* must be less than *dataSize*
- The sum of the *offset* and *size* members of any given element of *pMapEntries* must be less than or equal to *dataSize*

The definition of the *pMapEntries* member of type VkSpecializationMapEntry is:

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

The members of VkSpecializationMapEntry are as follows:

- *constantID* ID of the specialization constant in SPIR-V.

- *offset* byte offset of the specialization constant value within the supplied data buffer.
- *size* byte size of the specialization constant value within the supplied data buffer.

If a *constantID* value is not a specialization constant ID used in the shader, that map entry does not affect the behavior of the pipeline.

In human readable SPIR-V:

```
OpDecorate %x SpecId 13 ; decorate .x component of WorkgroupSize with ID 13
OpDecorate %y SpecId 42 ; decorate .y component of WorkgroupSize with ID 42
OpDecorate %z SpecId 3  ; decorate .z component of WorkgroupSize with ID 3
OpDecorate %wgsize BuiltIn WorkgroupSize ; decorate WorkgroupSize onto constant
%i32 = OpTypeInt 32 0 ; declare an unsigned 32-bit type
%uvec3 = OpTypeVector %i32 3 ; declare a 3 element vector type of unsigned 32-bit
%x = OpSpecConstant %i32 1 ; declare the .x component of WorkgroupSize
%y = OpSpecConstant %i32 1 ; declare the .y component of WorkgroupSize
%z = OpSpecConstant %i32 1 ; declare the .z component of WorkgroupSize
%wgsize = OpSpecConstantComposite %uvec3 %x %y %z ; declare WorkgroupSize
```

From the above we have three specialization constants, one for each of the x, y & z elements of the WorkgroupSize vector.

Now to specialize the above via the specialization constants mechanism:

```
const VkSpecializationMapEntry entries[] =
{
    {
        13,                             // constantID
        0 * sizeof(uint32_t),           // offset
        sizeof(uint32_t)                // size
    },
    {
        42,                             // constantID
        1 * sizeof(uint32_t),           // offset
        sizeof(uint32_t)                // size
    },
    {
        3,                              // constantID
        2 * sizeof(uint32_t),           // offset
        sizeof(uint32_t)                // size
    }
};

const uint32_t data[] = { 16, 8, 4 }; // our workgroup size is 16x8x4

const VkSpecializationInfo info =
{
    3,                                  // mapEntryCount
    entries,                            // pMapEntries
    3 * sizeof(uint32_t),               // dataSize
    data,                               // pData
};
```

Then when calling **vkCreateComputePipelines**, and passing the VkSpecializationInfo we defined as the *pSpecializationInfo* parameter of VkPipelineShaderStageCreateInfo, we will create a compute pipeline with the runtime specified work group size.

Another example would be that an application has a SPIR-V module that has some platform-dependent constants they wish to use.

In human readable SPIR-V:

```
OpDecorate %1 SpecId 0  ; decorate our signed 32-bit integer constant
OpDecorate %2 SpecId 12 ; decorate our 32-bit floating-point constant
%i32 = OpTypeInt 32 1   ; declare a signed 32-bit type
%float = OpTypeFloat 32 ; declare a 32-bit floating-point type
%1 = OpSpecConstant %i32 -1 ; some signed 32-bit integer constant
%2 = OpSpecConstant %float 0.5 ; some 32-bit floating-point constant
```

From the above we have two specialization constants, one is a signed 32-bit integer and the second is a 32-bit floating-point.

Now to specialize the above via the specialization constants mechanism:

```
VkSpecializationMapEntry entries[2];

const VkSpecializationMapEntry entries[] =
{
    {
        0,                              // constantID
        0 * sizeof(int32_t),            // offset
        sizeof(int32_t)                 // size
    },
    {
        12,                             // constantID
        1 * sizeof(int32_t),            // offset
        sizeof(float)                   // size
    }
};

int32_t data[2];

data[0] = -42; // set the data for the 32-bit integer
((float*)data)[1] = 42.0f; // set the data for the 32-bit floating-point

const VkSpecializationInfo info =
{
    2,                                  // mapEntryCount
    entries,                            // pMapEntries
    2 * sizeof(int32_t),                // dataSize
    data,                               // pData
};
```

It is legal for a SPIR-V module with specializations to be compiled into a pipeline where no specialization info was provided. SPIR-V specialization constants contain default values such that if a specialization is not provided, the default value will be used. In the examples above, it would be valid for an application to only specialize some of the specialization constants within the SPIR-V module, and let the other constants use their default values encoded within the OpSpecConstant declarations.

9.9 Pipeline Binding

Once a pipeline has been created, it can be bound to the command buffer using the command:

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

commandBuffer is the command buffer that the pipeline will be bound to.

pipelineBindPoint must have one of the values

```
typedef enum VkPipelineBindPoint {
    VK_PIPELINE_BIND_POINT_GRAPHICS = 0,
    VK_PIPELINE_BIND_POINT_COMPUTE = 1,
} VkPipelineBindPoint;
```

specifying whether *pipeline* will be bound as a compute (VK_PIPELINE_BIND_POINT_COMPUTE) or graphics (VK_PIPELINE_BIND_POINT_GRAPHICS) pipeline. There are separate bind points for each of graphics and compute, so binding one does not disturb the other.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *pipelineBindPoint* must be a valid `VkPipelineBindPoint` value
- *pipeline* must be a valid VkPipeline handle
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics or compute operations
- Each of *commandBuffer* and *pipeline* must have been created, allocated or retrieved from the same VkDevice
- If the value of *pipelineBindPoint* is VK_PIPELINE_BIND_POINT_COMPUTE, the VkCommandPool that *commandBuffer* was allocated from must support compute operations
- If the value of *pipelineBindPoint* is VK_PIPELINE_BIND_POINT_GRAPHICS, the VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- If the value of *pipelineBindPoint* is VK_PIPELINE_BIND_POINT_COMPUTE, *pipeline* must be a compute pipeline
- If the value of *pipelineBindPoint* is VK_PIPELINE_BIND_POINT_GRAPHICS, *pipeline* must be a graphics pipeline
- If the variable multisample rate feature is not supported, *pipeline* is a graphics pipeline, the current subpass has no attachments, and this is not the first call to this function with a graphics pipeline after transitioning to the current subpass, then the sample count specified by this pipeline must match that set in the previous pipeline

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Once bound, a pipeline binding affects subsequent graphics or compute commands in the command buffer until a different pipeline is bound to the bind point. The pipeline bound to VK_PIPELINE_BIND_POINT_COMPUTE controls the behavior of `vkCmdDispatch` and `vkCmdDispatchIndirect`. The pipeline bound to VK_ PIPELINE_BIND_POINT_GRAPHICS controls the behavior of `vkCmdDraw`, `vkCmdDrawIndexed`, `vkCmdDrawIndirect`, and `vkCmdDrawIndexedIndirect`. No other commands are affected by the pipeline state.

Chapter 10

Memory Allocation

Vulkan memory is broken up into two categories, *host memory* and *device memory*.

10.1 Host Memory

Host memory is memory needed by the Vulkan implementation for non-device-visible storage. This storage may be used for e.g. internal software structures.

Vulkan provides applications the opportunity to perform host memory allocations on behalf of the Vulkan implementation. If this feature is not used, the implementation will perform its own memory allocations. Since most memory allocations are off the critical path, this is not meant as a performance feature. Rather, this can be useful for certain embedded systems, for debugging purposes (e.g. putting a guard page after all host allocations), or for memory allocation logging.

Allocators are provided by the application as a pointer to a VkAllocationCallbacks structure:

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

- *`pUserData`* is a value to be interpreted by the implementation of the callbacks. When any of the callbacks in VkAllocationCallbacks are called, the Vulkan implementation will pass this value as the first parameter to the callback. This value can vary each time an allocator is passed into a command, even when the same object takes an allocator in multiple commands.
- *`pfnAllocation`* is a pointer to an application-defined memory allocation function of type `PFN_vkAllocationFunction`.
- *`pfnReallocation`* is a pointer to an application-defined memory reallocation function of type `PFN_vkReallocationFunction`.
- *`pfnFree`* is a pointer to an application-defined memory free function of type `PFN_vkFreeFunction`.

- *pfnInternalAllocation* is a pointer to an application-defined function that is called by the implementation when the implementation makes internal allocations, and it is of type `PFN_vkInternalAllocationNotification`.
- *pfnInternalFree* is a pointer to an application-defined function that is called by the implementation when the implementation frees internal allocations, and it is of type `PFN_vkInternalFreeNotification`.

Valid Usage

- *pfnAllocation* must be a pointer to a valid user-defined PFN_vkAllocationFunction
- *pfnReallocation* must be a pointer to a valid user-defined PFN_vkReallocationFunction
- *pfnFree* must be a pointer to a valid user-defined PFN_vkFreeFunction
- If either of *pfnInternalAllocatione* or *pfnInternalFree* is not `NULL`, both must be valid callbacks

An allocator indicates an error condition by returning `NULL` from *pfnAllocation* or *pfnReallocation*. If this occurs, the implementation should treat it as a run time error and should report VK_ERROR_OUT_OF_HOST_MEMORY at the appropriate time for the command in which the condition was detected, as described in Section 2.5.2.

The type of *pfnAllocation* is:

```
typedef void* (VKAPI_PTR *PFN_vkAllocationFunction)(
    void*                                       pUserData,
    size_t                                      size,
    size_t                                      alignment,
    VkSystemAllocationScope                     allocationScope);
```

size is the size in bytes of the requested allocation. *alignment* is the requested alignment of the allocation in bytes and must be a power of two. This function must either return `NULL` (in case of allocation failure or if *size* is zero) or a valid pointer to a memory allocation containing at least *size* bytes, and with the pointer value being a multiple of *alignment*. *allocationScope* is the scope of the lifetime of the allocation, as described here.

The type of *pfnReallocation* is:

```
typedef void* (VKAPI_PTR *PFN_vkReallocationFunction)(
    void*                                       pUserData,
    void*                                       pOriginal,
    size_t                                      size,
    size_t                                      alignment,
    VkSystemAllocationScope                     allocationScope);
```

This function must alter the size of an allocation, either by shrinking or growing it, to accommodate the new size. *pOriginal* must either be a pointer previously returned by *pfnReallocation* or *pfnAllocation* of the same allocator or be `NULL`. The remaining parameters have meanings identical to those of `PFN_vkAllocationFunction`.

If *pOriginal* is `NULL`, then this function must behave similarly to *pfnAllocation*. If *size* is zero, then this function must behave similarly to `PFN_vkFreeFunction`. If the new allocation is larger than the old allocation, then the contents of the additional space are undefined. If the new alignment is different from the old alignment, the new allocation must satisfy the new alignment. If satisfying these requirements involves creating a new allocation, then the old allocation must be freed. If this function fails, it must return `NULL` and not free the old allocation.

The type of *`pfnFree`* is:

```
typedef void (VKAPI_PTR *PFN_vkFreeFunction)(
    void*                                       pUserData,
    void*                                       pMemory);
```

`pMemory` may be `NULL`, which the callback must handle safely. If *`pMemory`* is non-`NULL`, it must be a pointer previously allocated by *`pfnAllocation`* or *`pfnReallocation`* and must be freed by the function.

The type of *`pfnInternalAllocation`* is:

```
typedef void (VKAPI_PTR *PFN_vkInternalAllocationNotification)(
    void*                                       pUserData,
    size_t                                      size,
    VkInternalAllocationType                    allocationType,
    VkSystemAllocationScope                     allocationScope);
```

The type of the allocation is indicated by the *`allocationType`* parameter. *`allocationScope`* is the scope of the lifetime of the allocation, as described here. This is a purely informational callback.

The type of *`pfnInternalFree`* is:

```
typedef void (VKAPI_PTR *PFN_vkInternalFreeNotification)(
    void*                                       pUserData,
    size_t                                      size,
    VkInternalAllocationType                    allocationType,
    VkSystemAllocationScope                     allocationScope);
```

The type of the allocation is indicated by the *`allocationType`* parameter. *`allocationScope`* is the scope of the lifetime of the allocation, as described here. This is a purely informational callback.

Each allocation has a *scope* which defines its lifetime and which object it is associated with. The scope is provided in the *`allocationScope`* parameter and takes a value of type `VkSystemAllocationScope`:

```
typedef enum VkSystemAllocationScope {
    VK_SYSTEM_ALLOCATION_SCOPE_COMMAND = 0,
    VK_SYSTEM_ALLOCATION_SCOPE_OBJECT = 1,
    VK_SYSTEM_ALLOCATION_SCOPE_CACHE = 2,
    VK_SYSTEM_ALLOCATION_SCOPE_DEVICE = 3,
    VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE = 4,
} VkSystemAllocationScope;
```

- VK_SYSTEM_ALLOCATION_SCOPE_COMMAND - The allocation is scoped to the lifetime of the Vulkan command.
- VK_SYSTEM_ALLOCATION_SCOPE_OBJECT - The allocation is scoped to the lifetime of the Vulkan object that is being created or used.
- VK_SYSTEM_ALLOCATION_SCOPE_CACHE - The allocation is scoped to the lifetime of a VkPipelineCache object.
- VK_SYSTEM_ALLOCATION_SCOPE_DEVICE - The allocation is scoped to the lifetime of the Vulkan device.

- VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE - The allocation is scoped to the lifetime of the Vulkan instance.

Most Vulkan commands operate on a single object, or there is a sole object that is being created or manipulated. When an allocation uses a scope of VK_SYSTEM_ALLOCATION_SCOPE_OBJECT or VK_SYSTEM_ALLOCATION_SCOPE_CACHE, the allocation is scoped to the object being created or manipulated.

When an implementation requires host memory, it will make callbacks to the application using the most specific allocator and scope available:

- If an allocation is scoped to the duration of a command, the allocator will use the VK_SYSTEM_ALLOCATION_SCOPE_COMMAND scope. The most specific allocator available is used: if the object being created or manipulated has an allocator, that object's allocator will be used, else if the parent VkDevice has an allocator it will be used, else if the parent VkInstance has an allocator it will be used. Else,
- If an allocation is associated with an object of type VkPipelineCache, the allocator will use the VK_SYSTEM_ALLOCATION_SCOPE_CACHE scope. The most specific allocator available is used (pipeline cache, else device, else instance). Else,
- If an allocation is scoped to the lifetime of an object, that object is being created or manipulated by the command, and that object's type is not VkDevice or VkInstance, the allocator will use a scope of VK_SYSTEM_ALLOCATION_SCOPE_OBJECT. The most specific allocator available is used (object, else device, else instance). Else,
- If an allocation is scoped to the lifetime of a device, the allocator will use scope of VK_SYSTEM_ALLOCATION_SCOPE_DEVICE. The most specific allocator available is used (device, else instance). Else,
- If the allocation is scoped to the lifetime of an instance and the instance has an allocator, its allocator will be used with a scope of VK_SYSTEM_ALLOCATION_SCOPE_INSTANCE.
- Otherwise an implementation will allocate memory through an alternative mechanism that is unspecified.

Objects that are allocated from pools do not specify their own allocator. When an implementation requires host memory for such an object, that memory is sourced from the object's parent pool's allocator.

The application is not expected to handle allocating memory that is intended for execution by the host due to the complexities of differing security implementations across multiple platforms. The implementation will allocate such memory internally and invoke an application provided informational callback when these "internal" allocations are allocated and freed. Upon allocation of executable memory, *`pfnInternalAllocation`* will be called. Upon freeing executable memory, *`pfnInternalFree`* will be called. An implementation will only call an informational callback for executable memory allocations and frees.

The *`allocationType`* parameter to the *`pfnInternalAllocation`* and *`pfnInternalFree`* functions may be one of the following values

```
typedef enum VkInternalAllocationType {
    VK_INTERNAL_ALLOCATION_TYPE_EXECUTABLE = 0,
} VkInternalAllocationType;
```

- VK_INTERNAL_ALLOCATION_TYPE_EXECUTABLE - The allocation is intended for execution by the host.

An implementation must only make calls into an application-provided allocator from within the scope of an API command. An implementation must only make calls into an application-provided allocator from the same thread that called the provoking API command. The implementation should not synchronize calls to any of the callbacks. If synchronization is needed, the callbacks must provide it themselves. The informational callbacks are subject to the same restrictions as the allocation callbacks.

If an implementation intends to make calls through an VkAllocationCallbacks structure between the time a **vkCreate*** command returns and the time a corresponding **vkDestroy*** command begins, that implementation must save a copy of the allocator before the **vkCreate*** command returns. The callback functions and any data structures they rely upon must remain valid for the lifetime of the object they are associated with.

If an allocator is provided to a **vkCreate*** command, a *compatible* allocator must be provided to the corresponding **vkDestroy*** command. Two VkAllocationCallbacks structures are compatible if memory created with *pfnAllocation* or *pfnReallocation* in each can be freed with *pfnReallocation* or *pfnFree* in the other. An allocator must not be provided to a **vkDestroy*** command if an allocator was not provided to the corresponding **vkCreate*** command.

If a non-`NULL` allocator is used, the *pfnAllocation*, *pfnReallocation* and *pfnFree* members must be non-`NULL` and point to valid implementations of the callbacks. An application can choose to not provide informational callbacks by setting both *pfnInternalAllocation* and *pfnInternalFree* to `NULL`. *pfnInternalAllocation* and *pfnInternalFree* must either both be `NULL` or both be non-`NULL`.

If *pfnAllocation* or *pfnReallocation* fail, the implementation may fail object creation and/or generate an VK_ ERROR_OUT_OF_HOST_MEMORY error, as appropriate.

The following sets of rules define when an implementation is permitted to call the allocator callbacks.

pfnAllocation or *pfnReallocation* may be called in the following situations:

- Host memory scoped to the lifetime of a VkDevice or VkInstance may be allocated from any API command.
- Host memory scoped to the lifetime of a VkPipelineCache may only be allocated from:
  - **vkCreatePipelineCache**
  - **vkMergePipelineCaches** for *dstCache*
  - **vkCreateGraphicsPipelines** for *pPipelineCache*
  - **vkCreateComputePipelines** for *pPipelineCache*
- Host memory scoped to the lifetime of a VkDescriptorPool may only be allocated from:
  - any command that takes the pool as a direct argument
  - **vkAllocateDescriptorSets** for the *descriptorPool* member of its *pAllocateInfo* parameter
  - **vkCreateDescriptorPool**
- Host memory scoped to the lifetime of a VkCommandPool may only be allocated from:
  - any command that takes the pool as a direct argument
  - **vkCreateCommandPool**
  - **vkAllocateCommandBuffers** for the *commandPool* member of its *pAllocateInfo* parameter
  - any **vkCmd*** command whose *commandBuffer* was created from that VkCommandPool
- Host memory scoped to the lifetime of any other object may only be allocated in that object's **vkCreate*** command.

pfnFree may be called in the following situations:

- Host memory scoped to the lifetime of a VkDevice or VkInstance may be freed from any API command.

- Host memory scoped to the lifetime of a VkPipelineCache may be freed from **`vkDestroyPipelineCache`**.
- Host memory scoped to the lifetime of a VkDescriptorPool may be freed from
  - any command that takes the pool as a direct argument
- Host memory scoped to the lifetime of a VkCommandPool may be freed from:
  - any command that takes the pool as a direct argument
  - **`vkResetCommandBuffer`** whose *`commandBuffer`* was created from that VkCommandPool
- Host memory scoped to the lifetime of any other object may be freed in that object's **`vkDestroy*`** command.
- Any command that allocates host memory may also free host memory of the same scope.

10.2 Device Memory

Device memory is memory that is visible to the device, for example the contents of opaque images that can be natively used by the device, or uniform buffer objects that reside in on-device memory.

The memory properties of the physical device describe the memory heaps and memory types available to a physical device. These can be queried by calling:

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

`physicalDevice` is the handle to the device whose properties to query. *`pMemoryProperties`* points to an instance of the VkPhysicalDeviceMemoryProperties structure that will be filled with information.

Valid Usage

- *`physicalDevice`* must be a valid VkPhysicalDevice handle
- *`pMemoryProperties`* must be a pointer to a VkPhysicalDeviceMemoryProperties structure

The definition of VkPhysicalDeviceMemoryProperties is:

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

The VkPhysicalDeviceMemoryProperties structure describes a number of *memory heaps* as well as a number of *memory types* that can be used to access memory allocated in those heaps. Each heap describes a memory resource of a particular size, and each memory type describes a set of memory properties (e.g. host cached vs uncached) that can be used with a given memory heap. Allocations using a particular memory type will consume resources from the heap indicated by that memory type's heap index. More than one memory type may share each heap, and the heaps and memory types provide a mechanism to advertise an accurate size of the physical memory resources while allowing the memory to be used with a variety of different properties.

The number of memory heaps is given by *memoryHeapCount* and is less than or equal to VK_MAX_MEMORY_HEAPS. Each heap is described by an element of the *memoryHeaps* array, as a VkMemoryHeap structure. The number of memory types available across all memory heaps is given by *memoryTypeCount* and is less than or equal to VK_MAX_MEMORY_TYPES. Each memory type is described by an element of the *memoryTypes* array, as a VkMemoryType structure.

The definition of VkMemoryHeap is:

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

- *size* is the total memory size in bytes in the heap.
- *flags* is a bitmask of attribute flags for the heap. The bits specified in *flags* are:

  ```
  typedef enum VkMemoryHeapFlagBits {
      VK_MEMORY_HEAP_DEVICE_LOCAL_BIT = 0x00000001,
  } VkMemoryHeapFlagBits;
  ```

  – if *flags* contains VK_MEMORY_HEAP_DEVICE_LOCAL_BIT, it means the heap corresponds to device local memory. Device local memory may have different performance characteristics than host local memory, and may support different memory property flags.

In a unified memory architecture (UMA) system, there is often only a single memory heap which is considered to be equally "local" to the host and to the device. If there is only one heap, that heap must be marked as VK_MEMORY_HEAP_DEVICE_LOCAL_BIT. If there are multiple heaps that all have similar performance characteristics, they may all be marked as VK_MEMORY_HEAP_DEVICE_LOCAL_BIT, but at least one will be device local.

The definition of VkMemoryType is:

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

- *heapIndex* describes which memory heap this memory type corresponds to, and must be less than *memoryHeapCount* from the VkPhysicalDeviceMemoryProperties structure.
- *propertyFlags* is a bitmask of properties for this memory type. The bits specified in *propertyFlags* are:

  ```
  typedef enum VkMemoryPropertyFlagBits {
      VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT = 0x00000001,
      VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT = 0x00000002,
      VK_MEMORY_PROPERTY_HOST_COHERENT_BIT = 0x00000004,
      VK_MEMORY_PROPERTY_HOST_CACHED_BIT = 0x00000008,
      VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT = 0x00000010,
  } VkMemoryPropertyFlagBits;
  ```

- if *propertyFlags* has the VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT bit set, memory allocated with this type is the most efficient for device access. This property will only be set for memory types belonging to heaps with the VK_MEMORY_HEAP_DEVICE_LOCAL_BIT set.
- if *propertyFlags* has the VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT bit set, memory allocated with this type can be mapped using `vkMapMemory` so that it can be accessed on the host.
- if *propertyFlags* has the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT bit set, host cache management commands **`vkFlushMappedMemoryRanges`** and **`vkInvalidateMappedMemoryRanges`** are not needed to make host writes visible to the device or device writes visible to the host, respectively.
- if *propertyFlags* has the VK_MEMORY_PROPERTY_HOST_CACHED_BIT bit set, memory allocated with this type is cached on the host. Host memory accesses to uncached memory are slower than to cached memory, however uncached memory is always host coherent.
- if *propertyFlags* has the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set, the memory type only allows device access to the memory. Memory types must not have both VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT and VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT set. Additionally, the object's backing memory may be provided by the implementation lazily as specified in Lazily Allocated Memory.

Each memory type returned by `vkGetPhysicalDeviceMemoryProperties` must have its *propertyFlags* set to one of the following values:

- 0
- VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT
- VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
- VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT
- VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
- VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
- VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT
- VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_CACHED_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT
- VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT | VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT

It is guaranteed that there is at least one memory type that has its *propertyFlags* with the VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT bit set and the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT bit set.

The memory types are sorted according to a partial order which serves to aid in easily selecting an appropriate memory type. Given two memory types X and Y, the partial order defines $X \leq Y$ if:

- the memory property bits set for X are a subset of the memory property bits set for Y. Or,
- the memory property bits set for X are the same as the memory property bits set for Y, and X uses a memory heap with greater or equal performance (as determined in an implementation-specific manner).

Memory types are ordered in the list such that X is assigned a lesser *memoryTypeIndex* than Y if $X \leq Y$ according to the partial order. Note that the list of all allowed memory property flag combinations above satisfies this partial order, but other orders would as well. The goal of this ordering is to enable applications to use a simple search loop in selecting the proper memory type, along the lines of:

```
// Searching for the best match for "properties"
for (i = 0; i < memoryTypeCount; ++i)
    if ((memoryTypes[i].propertyFlags & properties) == properties)
        return i;
```

This loop will find the first entry that has all bits requested in **properties** set. If there is no exact match, it will find a closest match (i.e. a memory type with the fewest additional bits set), which has some additional bits set but which are not detrimental to the behaviors requested by **properties**. If there are multiple heaps with the same properties, it will choose the most performant memory.

A Vulkan device operates on data in device memory via memory objects that are represented in the API by a VkDeviceMemory handle. Memory objects are allocated by calling **vkAllocateMemory**:

```
VkResult vkAllocateMemory(
    VkDevice                                    device,
    const VkMemoryAllocateInfo*                 pAllocateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDeviceMemory*                             pMemory);
```

The *device* passed in will own the memory returned by the output parameter *pMemory*. *pAllocateInfo* is a pointer to a structure of type `VkMemoryAllocateInfo`, which contains parameters of the allocation. A successful returned allocation must use the requested parameters—no substitution is permitted by the implementation.

Valid Usage

- *device* must be a valid VkDevice handle
- *pAllocateInfo* must be a pointer to a valid VkMemoryAllocateInfo structure
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pMemory* must be a pointer to a VkDeviceMemory handle
- The number of currently valid memory objects, allocated from *device*, must be less than VkPhysicalDeviceLimits::*maxMemoryAllocationCount*

VkMemoryAllocateInfo is defined as:

```
typedef struct VkMemoryAllocateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkDeviceSize                                allocationSize;
    uint32_t                                    memoryTypeIndex;
} VkMemoryAllocateInfo;
```

- *sType* is the type of this structure.

- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *allocationSize* is the size of the allocation in bytes
- *memoryTypeIndex* is the memory type index, which selects the properties of the memory to be allocated, as well as the heap the memory will come from.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO
- *pNext* must be `NULL`
- The value of *allocationSize* must be less than or equal to the amount of memory available to the VkMemoryHeap specified by *memoryTypeIndex* and the calling command's VkDevice
- The value of *allocationSize* must be greater than `0`

Allocations returned by **`vkAllocateMemory`** are guaranteed to meet any alignment requirement by the implementation. For example, if an implementation requires 128 byte alignment for images and 64 byte alignment for buffers, the device memory returned through this mechanism would be 128-byte aligned. This ensures that applications can correctly suballocate objects of different types (with potentially different alignment requirements) in the same memory object.

When memory is allocated, its contents are undefined.

There is an implementation-dependent maximum number of memory allocations which can be simultaneously created on a device. This is specified by the maxMemoryAllocationCount member of the VkPhysicalDeviceLimits structure. If *maxMemoryAllocationCount* is exceeded, **`vkAllocateMemory`** will return VK_ERROR_TOO_MANY_OBJECTS.

A memory object is freed by calling:

```
void vkFreeMemory(
    VkDevice                                    device,
    VkDeviceMemory                              memory,
    const VkAllocationCallbacks*                pAllocator);
```

memory is the VkDeviceMemory object to be freed. The logical device which owns the memory object *memory* is specified by *device*.

Valid Usage

- *device* must be a valid VkDevice handle
- If *memory* is not VK_NULL_HANDLE, *memory* must be a valid VkDeviceMemory handle

- If *pAllocator* is not NULL, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *memory* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *memory* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to *memory* (via images or buffers) must have completed execution

Host Synchronization

- Host access to *memory* must be externally synchronized

Before freeing a memory object, an application must ensure the memory object is no longer in use by the device—for example by command buffers queued for execution. The memory can remain bound to images or buffers at the time the memory object is freed, but any further use of them (on host or device) for anything other than destroying those objects will result in undefined behavior. If there are still any bound images or buffers, the memory may not be immediately released by the implementation, but must be released by the time all bound images and buffers have been destroyed. Once memory is released, it is returned to the heap from which it was allocated.

How memory objects are bound to Images and Buffers is described in detail in the Resource Memory Association section.

10.2.1 Host Access to Device Memory Objects

Memory objects created with **vkAllocateMemory** are not directly host accessible.

Memory objects created with the memory property VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT are considered *mappable*. Memory objects must be mappable in order to be successfully mapped on the host. An application retrieves a host virtual address pointer to a region of a mappable memory object by calling:

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

The logical device which owns the memory object *memory* is specified by *device*. *offset* is a zero-based byte offset from the beginning of the memory object, and *size* is the size of the memory range to map (or VK_WHOLE_SIZE to map from *offset* to the end of the allocation). *flags* is reserved for future use, and must be zero. If successful, *ppData* is filled with a host-accessible pointer to the beginning of the mapped range; this pointer minus *offset* must be aligned to at least VkPhysicalDeviceLimits::*minMemoryMapAlignment*.

Valid Usage

- *`device`* must be a valid VkDevice handle
- *`memory`* must be a valid VkDeviceMemory handle
- *`flags`* must be `0`
- *`ppData`* must be a pointer to a pointer
- *`memory`* must have been created, allocated or retrieved from *`device`*
- Each of *`device`* and *`memory`* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *`memory`* must not currently be mapped
- *`offset`* must be less than the size of *`memory`*
- If *`size`* is not equal to VK_WHOLE_SIZE, the sum of *`offset`* and *`size`* must be less than or equal to the *`size`* of the *`memory`*
- *`memory`* must have been created with a memory type that reports VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT

Host Synchronization

- Host access to *`memory`* must be externally synchronized

It is an application error to call **`vkMapMemory`** on a memory object that is already mapped.

`vkMapMemory` does not check whether the device memory is currently in use before returning the host-accessible pointer. The application must guarantee that any previously submitted command that writes to this sub-range has completed before the host reads from or writes to that sub-range, and that any previously submitted command that reads from that sub-range has completed before the host writes to that region (see here for details on fulfilling such a guarantee). If the device memory was allocated without the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT set, these guarantees must be made for an extended sub-range: the application must round down the start of the sub-range to the previous multiple of VkPhysicalDeviceLimits::*`nonCoherentAtomSize`*, and round the end of the range up to the nearest multiple of VkPhysicalDeviceLimits::*`nonCoherentAtomSize`*.

While a range of device memory is mapped for host access, the application is responsible for synchronizing both device and host access to that memory range.

Note
It is important for the application developer to become meticulously familiar with all of the mechanisms described in the chapter on Synchronization and Cache Control as they are crucial to maintaining memory access ordering.

Host-visible memory types that advertise the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT property still require memory barriers between host and device in order to be coherent, but do not require additional cache management operations (**`vkFlushMappedMemoryRanges`** or **`vkInvalidateMappedMemoryRanges`**) to achieve coherency. For host writes to be seen by subsequent command buffer operations, a pipeline barrier from a source of VK_ACCESS_HOST_WRITE_BIT and VK_PIPELINE_STAGE_HOST_BIT to a destination of the relevant device pipeline stages and access types must be performed. Note that such a barrier is performed implicitly upon each command buffer submission, so an explicit barrier is only rarely needed (e.g. if a command buffer waits upon an event signaled by the host, where the host wrote some data after submission). For device writes to be seen by subsequent host reads, a pipeline barrier is required to make the writes visible.

In order to enable applications to work with non-coherent memory allocations, two entry points are provided. To flush host write caches, an application must use **`vkFlushMappedMemoryRanges`**, while **`vkInvalidateMappedMemoryRanges`** allows invalidating host input caches so that device writes become visible to the host. **`vkFlushMappedMemoryRanges`** must be called after the host writes to non-coherent memory have completed and before command buffers that will read or write any of those memory locations are submitted to a queue. Similarly, **`vkInvalidateMappedMemoryRanges`** must be called after command buffers that execute and flush (via memory barriers) the device writes have completed, and before the host will read or write any of those locations.

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

`device` is the logical Vulkan device. *`pMemoryRanges`* refers to an array with *`memoryRangeCount`* elements of type `VkMappedMemoryRange`.

Valid Usage

- *`device`* must be a valid VkDevice handle
- *`pMemoryRanges`* must be a pointer to an array of *`memoryRangeCount`* valid VkMappedMemoryRange structures
- The value of *`memoryRangeCount`* must be greater than `0`
- The memory ranges specified by *`pMemoryRanges`* must all currently be mapped

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

`device` is the logical Vulkan device. *`pMemoryRanges`* refers to an array with *`memoryRangeCount`* elements of type `VkMappedMemoryRange`.

Valid Usage

- *device* must be a valid VkDevice handle
- *pMemoryRanges* must be a pointer to an array of *memoryRangeCount* valid VkMappedMemoryRange structures
- The value of *memoryRangeCount* must be greater than 0
- The memory ranges specified by *pMemoryRanges* must all currently be mapped

VkMappedMemoryRange is defined as:

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

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to an extension-specific structure.
- *memory* is the memory object to which this range belongs.
- *offset* is the zero-based byte offset from the beginning of the memory object.
- *size* is either the size of range, or VK_WHOLE_SIZE to affect the range from *offset* to the end of the current mapping of the allocation.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE
- *pNext* must be NULL
- *memory* must be a valid VkDeviceMemory handle
- *memory* must currently be mapped
- *offset* must be less than the size of the currently mapped range of *memory*
- If *size* is not equal to VK_WHOLE_SIZE, the sum of *offset* and *size* must be less than or equal to the size of the currently mapped range of *memory*
- *offset* and *size* must each be a multiple of VkPhysicalDeviceLimits::*nonCoherentAtomSize*

Note
If the memory object was created with the VK_MEMORY_PROPERTY_HOST_COHERENT_BIT set, **`vkFlushMappedMemoryRanges`** and **`vkInvalidateMappedMemoryRanges`** are unnecessary and may have performance cost.

Once host access to a memory object is no longer needed by the application, it can be unmapped by calling :

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

`device` is the logical Vulkan device. *`memory`* is the memory object to be unmapped.

Valid Usage

- *`device`* must be a valid VkDevice handle
- *`memory`* must be a valid VkDeviceMemory handle
- *`memory`* must have been created, allocated or retrieved from *`device`*
- Each of *`device`* and *`memory`* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *`memory`* must currently be mapped

Host Synchronization

- Host access to *`memory`* must be externally synchronized

10.2.2 Lazily Allocated Memory

If the memory object is allocated from a heap with the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set, that object's backing memory may be provided by the implementation lazily. The actual committed size of the memory may initially be as small as zero (or as large as the requested size), and monotonically increases as additional memory is needed.

A memory type with this flag set is only allowed to be bound to a VkImage whose usage flags include VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT.

Note
Using lazily allocated memory objects for framebuffer attachments that are not needed once a render pass instance has completed may allow some implementations to never allocate memory for such attachments.

Determining the amount of lazily-allocated memory that is currently committed for a memory object is achieved by calling:

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

device is the logical Vulkan device. *memory* is the memory object being queried. Upon successful return, *pCommittedMemoryInBytes* will contain the number of bytes currently committed.

Valid Usage

- *device* must be a valid VkDevice handle
- *memory* must be a valid VkDeviceMemory handle
- *pCommittedMemoryInBytes* must be a pointer to a VkDeviceSize value
- *memory* must have been created, allocated or retrieved from *device*
- Each of *device* and *memory* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *memory* must have been created with a memory type that reports VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT

The implementation may update the commitment at any time, and the value returned by this query may be out of date.

The implementation guarantees to allocate any committed memory from the heapIndex indicated by the memory type that the memory object was created with.

Chapter 11

Resource Creation

Vulkan supports two primary resource types: *buffers* and *images*. Resources are views of memory with associated formatting and dimensionality. Buffers are essentially unformatted arrays of bytes whereas images contain format information, can be multidimensional and may have associated metadata.

11.1 Buffers

Buffers represent linear arrays of data which are used for various purposes by binding them to the graphics pipeline via descriptor sets or via certain commands, or by directly specifying them as parameters to certain commands.

Buffers are created by calling:

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

device is the device to be used to create the buffer object. The resulting buffer object handle is returned in *pBuffer*. *pCreateInfo* is a pointer to an instance of the VkBufferCreateInfo structure containing parameters affecting creation of the buffer. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- *pCreateInfo* must be a pointer to a valid VkBufferCreateInfo structure
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pBuffer* must be a pointer to a VkBuffer handle

- If the *flags* member of *pCreateInfo* includes VK_BUFFER_CREATE_SPARSE_BINDING_BIT or VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT, creating this VkBuffer must not cause the total required sparse memory for all currently valid sparse resources on the device to exceed VkPhysicalDeviceLimits::*sparseAddressSpaceSize*

The definition of VkBufferCreateInfo is:

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

The members of VkBufferCreateInfo have the following meanings:

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is a bitfield describing additional parameters of the buffer. See `VkBufferCreateFlagBits` below for a description of the supported bits.
- *size* is the size in bytes of the buffer to be created.
- *usage* is a bitfield describing the allowed usages of the buffer. See `VkBufferUsageFlagBits` below for a description of the supported bits.
- *sharingMode* is the sharing mode of the buffer when it is referenced from multiple queue families, see `VkSharingMode` in the Resource Sharing section below for supported values.
- *queueFamilyIndexCount* is the number of entries in the *pQueueFamilyIndices* array.
- *pQueueFamilyIndices* is a list of queue families that will reference this buffer (ignored if *sharingMode* is not VK_SHARING_MODE_CONCURRENT).

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be a valid combination of `VkBufferCreateFlagBits` values
- *usage* must be a valid combination of `VkBufferUsageFlagBits` values

- *usage* must not be 0
- *sharingMode* must be a valid VkSharingMode value
- The value of *size* must be greater than 0
- If *sharingMode* is VK_SHARING_MODE_CONCURRENT, *pQueueFamilyIndices* must be a pointer to an array of *queueFamilyIndexCount* uint32_t values
- If *sharingMode* is VK_SHARING_MODE_CONCURRENT, *queueFamilyIndexCount* must be greater than 1
- If the sparse bindings feature is not enabled, *flags* must not contain VK_BUFFER_CREATE_SPARSE_BINDING_BIT
- If the sparse buffer residency feature is not enabled, *flags* must not contain VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT
- If the sparse aliased residency feature is not enabled, *flags* must not contain VK_BUFFER_CREATE_SPARSE_ALIASED_BIT
- If *flags* contains VK_BUFFER_CREATE_SPARSE_ALIASED_BIT, it must also contain at least one of VK_BUFFER_CREATE_SPARSE_BINDING_BIT or VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT

Bits which may be set in *usage* are:

```
typedef enum VkBufferUsageFlagBits {
    VK_BUFFER_USAGE_TRANSFER_SRC_BIT = 0x00000001,
    VK_BUFFER_USAGE_TRANSFER_DST_BIT = 0x00000002,
    VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT = 0x00000004,
    VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT = 0x00000008,
    VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT = 0x00000010,
    VK_BUFFER_USAGE_STORAGE_BUFFER_BIT = 0x00000020,
    VK_BUFFER_USAGE_INDEX_BUFFER_BIT = 0x00000040,
    VK_BUFFER_USAGE_VERTEX_BUFFER_BIT = 0x00000080,
    VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT = 0x00000100,
} VkBufferUsageFlagBits;
```

- VK_BUFFER_USAGE_TRANSFER_SRC_BIT indicates that the buffer can be used as the source of a *transfer command* (see the definition of VK_PIPELINE_STAGE_TRANSFER_BIT).
- VK_BUFFER_USAGE_TRANSFER_DST_BIT indicates that the buffer can be used as the destination of a transfer command.
- VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT indicates that the buffer can be used to create a VkBufferView suitable for occupying a VkDescriptorSet slot of type VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER.
- VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT indicates that the buffer can be used to create a VkBufferView suitable for occupying a VkDescriptorSet slot of type VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER.

- VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT indicates that the buffer can be used in a VkDescriptorBufferInfo suitable for occupying a VkDescriptorSet slot either of type VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC.
- VK_BUFFER_USAGE_STORAGE_BUFFER_BIT indicates that the buffer can be used in a VkDescriptorBufferInfo suitable for occupying a VkDescriptorSet slot either of type VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC.
- VK_BUFFER_USAGE_INDEX_BUFFER_BIT indicates that the buffer is suitable for passing as the *buffer* parameter to **vkCmdBindIndexBuffer**.
- VK_BUFFER_USAGE_VERTEX_BUFFER_BIT indicates that the buffer is suitable for passing as an element of the *pBuffers* array to **vkCmdBindVertexBuffers**.
- VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT indicates that the buffer is suitable for passing as the *buffer* parameter to **vkCmdDrawIndirect**, **vkCmdDrawIndexedIndirect**, or **vkCmdDispatchIndirect**.

Any combination of bits can be specified for *usage*, but at least one of the bits must be set in order to create a valid buffer.

Bits which may be set in *flags* are:

```
typedef enum VkBufferCreateFlagBits {
    VK_BUFFER_CREATE_SPARSE_BINDING_BIT = 0x00000001,
    VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT = 0x00000002,
    VK_BUFFER_CREATE_SPARSE_ALIASED_BIT = 0x00000004,
} VkBufferCreateFlagBits;
```

These bitfields have the following meanings:

- VK_BUFFER_CREATE_SPARSE_BINDING_BIT indicates that the buffer will be backed using sparse memory binding.
- VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT indicates that the buffer can be partially backed using sparse memory binding.
- VK_BUFFER_CREATE_SPARSE_ALIASED_BIT indicates that the buffer will be backed using sparse memory binding with memory ranges that might also simultaneously be backing another buffer (or another portion of the same buffer).

See Sparse Resource Features and Physical Device Features for details of the sparse memory features supported on a device.

To destroy a buffer, call:

```
void vkDestroyBuffer(
    VkDevice                                    device,
    VkBuffer                                    buffer,
    const VkAllocationCallbacks*                pAllocator);
```

device is the device to be used to destroy the buffer. *buffer* is the handle of the buffer to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- `device` must be a valid VkDevice handle
- If `buffer` is not VK_NULL_HANDLE, `buffer` must be a valid VkBuffer handle
- If `pAllocator` is not `NULL`, `pAllocator` must be a pointer to a valid VkAllocationCallbacks structure
- If `buffer` is a valid handle, it must have been created, allocated or retrieved from `device`
- Each of `device` and `buffer` that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to `buffer`, either directly or via a VkBufferView, must have completed execution
- If VkAllocationCallbacks were provided when `buffer` was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when `buffer` was created, `pAllocator` must be `NULL`

Host Synchronization

- Host access to `buffer` must be externally synchronized

11.2 Buffer Views

A *buffer view* represents a contiguous range of a buffer and a specific format to be used to interpret the data. Buffer views are used to enable shaders to access buffer contents interpreted as formatted data. In order to create a valid buffer view, the buffer must have been created with at least one of the following usage flags:

- VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT
- VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT

A buffer view is created by calling:

```
VkResult vkCreateBufferView(
    VkDevice                                    device,
    const VkBufferViewCreateInfo*               pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkBufferView*                               pView);
```

`device` is the device to be used to create the buffer view. The resulting buffer view object handle is returned in `pView`. `pCreateInfo` is a pointer to an instance of the VkBufferViewCreateInfo structure containing parameters to be used to create the buffer. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- *pCreateInfo* must be a pointer to a valid VkBufferViewCreateInfo structure
- If *pAllocator* is not NULL, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pView* must be a pointer to a VkBufferView handle

The definition of VkBufferViewCreateInfo is:

```
typedef struct VkBufferViewCreateInfo {
    VkStructureType                                          sType;
    const void*                                              pNext;
    VkBufferViewCreateFlags                                  flags;
    VkBuffer                                                 buffer;
    VkFormat                                                 format;
    VkDeviceSize                                             offset;
    VkDeviceSize                                             range;
} VkBufferViewCreateInfo;
```

The members of VkBufferViewCreateInfo have the following meanings:

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *buffer* is a VkBuffer on which the view will be created.
- *format* is a VkFormat describing the format of the data elements in the buffer.
- *offset* is an offset in bytes from the base address of the buffer. Accesses to the buffer view from shaders use addressing that is relative to this starting offset.
- *range* is a size in bytes of the buffer view. If *range* is equal to VK_WHOLE_SIZE, the range from *offset* to the end of the buffer is used. If VK_WHOLE_SIZE is used and the remaining size of the buffer is not a multiple of the element size of *format*, then the nearest smaller multiple is used.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_BUFFER_VIEW_CREATE_INFO

- *pNext* must be `NULL`
- *flags* must be `0`
- *buffer* must be a valid VkBuffer handle
- *format* must be a valid `VkFormat` value
- The value of *offset* must be a multiple of VkPhysicalDeviceLimits::*minTexelBufferOffsetAlignment*
- The value of *range* must be greater than `0`
- If *range* is not equal to VK_WHOLE_SIZE, the sum of *offset* and *range* must be less than or equal to the size of *buffer*
- If *range* is not equal to VK_WHOLE_SIZE, the value of *range* must be a multiple of the element size of *format*
- The value of *range*, divided by the size of an element of *format*, must be less than or equal to the value of VkPhysicalDeviceLimits::*maxTexelBufferElements*
- *buffer* must have been created with a *usage* value containing at least one of VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT or VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT
- If *buffer* was created with *usage* containing VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT, *format* must be supported for uniform texel buffers, as specified by the VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT flag in VkFormatProperties::*bufferFeatures* returned by **vkGetPhysicalDeviceFormatProperties**
- If *buffer* was created with *usage* containing VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT, *format* must be supported for storage texel buffers, as specified by the VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT flag in VkFormatProperties::*bufferFeatures* returned by **vkGetPhysicalDeviceFormatProperties**

To destroy a buffer view, call:

```
void vkDestroyBufferView(
    VkDevice                                    device,
    VkBufferView                                bufferView,
    const VkAllocationCallbacks*                pAllocator);
```

device is the device to be used to destroy the buffer view. *bufferView* is the handle of the buffer view to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *bufferView* is not VK_NULL_HANDLE, *bufferView* must be a valid VkBufferView handle
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure

- If *bufferView* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *bufferView* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to *bufferView* must have completed execution
- If VkAllocationCallbacks were provided when *bufferView* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *bufferView* was created, *pAllocator* must be `NULL`

Host Synchronization

- Host access to *bufferView* must be externally synchronized

11.3 Images

Images represent multidimensional - up to 3 - arrays of data which can be used for various purposes (e.g. attachments, textures), by binding them to the graphics pipeline via descriptor sets, or by directly specifying them as parameters to certain commands.

Images are created by calling:

```
VkResult vkCreateImage(
    VkDevice                                    device,
    const VkImageCreateInfo*                    pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkImage*                                    pImage);
```

device is the device to be used to create the image object. The resulting image object handle is returned in *pImage*. *pCreateInfo* is a pointer to an instance of the VkImageCreateInfo structure containing parameters to be used to create the image. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- *pCreateInfo* must be a pointer to a valid VkImageCreateInfo structure
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure

- *pImage* must be a pointer to a VkImage handle
- If the *flags* member of *pCreateInfo* includes VK_IMAGE_CREATE_SPARSE_BINDING_BIT or VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT, creating this VkImage must not cause the total required sparse memory for all currently valid sparse resources on the device to exceed VkPhysicalDeviceLimits::*sparseAddressSpaceSize*

The definition of VkImageCreateInfo is:

```
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;
```

The members of VkImageCreateInfo have the following meanings:

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is a bitfield describing additional parameters of the image. See `VkImageCreateFlagBits` below for a description of the supported bits.
- *imageType* is the basic dimensionality of the image, and must be one of the values

  ```
  typedef enum VkImageType {
      VK_IMAGE_TYPE_1D = 0,
      VK_IMAGE_TYPE_2D = 1,
      VK_IMAGE_TYPE_3D = 2,
  } VkImageType;
  ```

  specifying one-, two-, or three-dimensionality, respectively. Layers in array textures do not count as a dimension for the purposes of the image type.
- *format* is a `VkFormat` describing the format and type of the data elements that will be contained in the image.
- *extent* is a `VkExtent3D` describing the number of data elements in each dimension of the base level.
- *mipLevels* describes the number of levels of detail available for minified sampling of the image.
- *arrayLayers* is the number of layers in the image.

- `samples` is the number of sub-data element samples in the image as defined in `VkSampleCountFlagBits`. See Multisampling.
- `tiling` is the tiling arrangement of the data elements in memory, and must have one of the values:

  ```
  typedef enum VkImageTiling {
      VK_IMAGE_TILING_OPTIMAL = 0,
      VK_IMAGE_TILING_LINEAR = 1,
  } VkImageTiling;
  ```

  VK_IMAGE_TILING_OPTIMAL specifies optimal tiling (texels are laid out in an implementation-dependent arrangement, for more optimal memory access), and VK_IMAGE_TILING_LINEAR specifies linear tiling (texels are laid out in memory in row-major order, possibly with some padding on each row).
- `usage` is a bitfield describing the intended usage of the image. See `VkImageUsageFlagBits` below for a description of the supported bits.
- `sharingMode` is the sharing mode of the image when it is referenced from multiple queue families, and must be one of the values described for `VkSharingMode` in the Resource Sharing section below.
- `queueFamilyIndexCount` is the number of entries in the `pQueueFamilyIndices` array.
- `pQueueFamilyIndices` is a list of queue families that will reference this image (ignored if `sharingMode` is not VK_SHARING_MODE_CONCURRENT).
- `initialLayout` selects the initial `VkImageLayout` state of all subresources of the image. See Image Layouts. `initialLayout` must be VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED.

Valid Usage

- `sType` must be VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO
- `pNext` must be `NULL`
- `flags` must be a valid combination of `VkImageCreateFlagBits` values
- `imageType` must be a valid `VkImageType` value
- `format` must be a valid `VkFormat` value
- `samples` must be a valid `VkSampleCountFlagBits` value
- `tiling` must be a valid `VkImageTiling` value
- `usage` must be a valid combination of `VkImageUsageFlagBits` values
- `usage` must not be `0`
- `sharingMode` must be a valid `VkSharingMode` value
- `initialLayout` must be a valid `VkImageLayout` value
- If `sharingMode` is VK_SHARING_MODE_CONCURRENT, `pQueueFamilyIndices` must be a pointer to an array of `queueFamilyIndexCount` uint32_t values

- If *sharingMode* is VK_SHARING_MODE_CONCURRENT, *queueFamilyIndexCount* must be greater than 1
- *format* must not be VK_FORMAT_UNDEFINED
- The values of the *width*, *height* and *depth* members of *extent* must all be greater than 0
- The value of *mipLevels* must be greater than 0
- The value of *arrayLayers* must be greater than 0
- If *imageType* is VK_IMAGE_TYPE_1D, the value of *extent.width* must be less than or equal to the value of VkPhysicalDeviceLimits::*maxImageDimension1D*, or the value of VkImageFormatProperties::*maxExtent.width* (as returned by **vkGetPhysicalDeviceImageFormatProperties** with values of *format*, *type*, *tiling*, *usage* and *flags* equal to those in this structure) - whichever is higher
- If *imageType* is VK_IMAGE_TYPE_2D and *flags* does not contain VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, the value of *extent.width* and *extent.height* must be less than or equal to the value of VkPhysicalDeviceLimits::*maxImageDimension2D*, or the value of VkImageFormatProperties::*maxExtent.width*/height (as returned by **vkGetPhysicalDeviceImageFormatProperties** with values of *format*, *type*, *tiling*, *usage* and *flags* equal to those in this structure) - whichever is higher
- If *imageType* is VK_IMAGE_TYPE_2D and *flags* contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, the value of *extent.width* and *extent.height* must be less than or equal to the value of VkPhysicalDeviceLimits::*maxImageDimensionCube*, or the value of VkImageFormatProperties::*maxExtent.width*/height (as returned by **vkGetPhysicalDeviceImageFormatProperties** with values of *format*, *type*, *tiling*, *usage* and *flags* equal to those in this structure) - whichever is higher
- If *imageType* is VK_IMAGE_TYPE_2D and *flags* contains VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT, the value of *extent.width* and *extent.height* must be equal
- If *imageType* is VK_IMAGE_TYPE_3D, the value of *extent.width*, *extent.height* and *extent.depth* must be less than or equal to the value of VkPhysicalDeviceLimits::*maxImageDimension3D*, or the value of VkImageFormatProperties::*maxExtent.width*/height/depth (as returned by **vkGetPhysicalDeviceImageFormatProperties** with values of *format*, *type*, *tiling*, *usage* and *flags* equal to those in this structure) - whichever is higher
- The value of *mipLevels* must be less than or equal to or equal to the value of $\lfloor \log_2(\max(extent.width, extent.height, extent.depth)) \rfloor + 1$
- If the values of any of *extent.width*, *extent.height* or *extent.depth* are greater than the values of the equivalently named members of VkPhysicalDeviceLimits::*maxImageDimension3D*, *mipLevels* must be less than or equal to the value of VkImageFormatProperties::*maxMipLevels* (as returned by **vkGetPhysicalDeviceImageFormatProperties** with values of *format*, *type*, *tiling*, *usage* and *flags* equal to those in this structure)
- If the values of any of *extent.width*, *extent.height* or *extent.depth* are greater than the values of the equivalently named members of VkPhysicalDeviceLimits::*maxImageDimension3D*, (*extent.width* * *extent.height* * *extent.depth*) must be less than or equal to the value of VkImageFormatProperties::*maxResourceSize* (as returned by `vkGetPhysicalDeviceImageFormatProperties` with values of *format*, *type*, *tiling*, *usage* and *flags* equal to those in this structure)

- The value of *arrayLayers* must be less than or equal to the value of VkPhysicalDeviceLimits::*maxImageArrayLayers*, or the value of VkImageFormatProperties::*maxArrayLayers* (as returned by **vkGetPhysicalDeviceImageFormatProperties** with values of *format*, *type*, *tiling*, *usage* and *flags* equal to those in this structure) - whichever is higher
- The value of *samples* must be a bit value that is set in the value of VkPhysicalDeviceLimits::*sampleCounts* returned by vkGetPhysicalDeviceProperties, or the value of VkImageFormatProperties::*maxExtent.sampleCounts* returned by **vkGetPhysicalDeviceImageFormatProperties** with values of *format*, *type*, *tiling*, *usage* and *flags* equal to those in this structure
- If *usage* includes VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT, the value of *extent.width* must be less than or equal to VkPhysicalDeviceLimits::*maxFramebufferWidth*
- If *usage* includes VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT or VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT, the value of *extent.height* must be less than or equal to VkPhysicalDeviceLimits::*maxFramebufferHeight*
- If *usage* includes VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, the value of *samples* must be a bit value that is set in the value of VkPhysicalDeviceLimits::*maxFramebufferColorSamples*
- If *usage* includes VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, and *format* includes a depth aspect, the value of *samples* must be a bit value that is set in the value of VkPhysicalDeviceLimits::*maxFramebufferDepthSamples*
- If *usage* includes VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, and *format* includes a stencil aspect, the value of *samples* must be a bit value that is set in the value of VkPhysicalDeviceLimits::*maxFramebufferStencilSamples*
- If *usage* includes VK_IMAGE_USAGE_SAMPLED_BIT, and *format* includes a color aspect, the value of *samples* must be a bit value that is set in the value of VkPhysicalDeviceLimits::*maxSampledImageColorSamples*
- If *usage* includes VK_IMAGE_USAGE_SAMPLED_BIT, and *format* includes a depth aspect, the value of *samples* must be a bit value that is set in the value of VkPhysicalDeviceLimits::*maxSampledImageDepthSamples*
- If *usage* includes VK_IMAGE_USAGE_SAMPLED_BIT, and *format* is an integer format, the value of *samples* must be a bit value that is set in the value of VkPhysicalDeviceLimits::*maxSampledImageIntegerSamples*
- If *usage* includes VK_IMAGE_USAGE_STORAGE_BIT, the value of *samples* must be a bit value that is set in the value of VkPhysicalDeviceLimits::*maxStorageImageSamples*
- If the ETC2 texture compression feature is not enabled, *format* must not be VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK, VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK, VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK, VK_FORMAT_EAC_R11_UNORM_BLOCK, VK_FORMAT_EAC_R11_SNORM_BLOCK, VK_FORMAT_EAC_R11G11_UNORM_BLOCK, or VK_FORMAT_EAC_R11G11_SNORM_BLOCK

- If the ASTC LDR texture compression feature is not enabled, `format` must not be VK_FORMAT_ASTC_4x4_UNORM_BLOCK, VK_FORMAT_ASTC_4x4_SRGB_BLOCK, VK_FORMAT_ASTC_5x4_UNORM_BLOCK, VK_FORMAT_ASTC_5x4_SRGB_BLOCK, VK_FORMAT_ASTC_5x5_UNORM_BLOCK, VK_FORMAT_ASTC_5x5_SRGB_BLOCK, VK_FORMAT_ASTC_6x5_UNORM_BLOCK, VK_FORMAT_ASTC_6x5_SRGB_BLOCK, VK_FORMAT_ASTC_6x6_UNORM_BLOCK, VK_FORMAT_ASTC_6x6_SRGB_BLOCK, VK_FORMAT_ASTC_8x5_UNORM_BLOCK, VK_FORMAT_ASTC_8x5_SRGB_BLOCK, VK_FORMAT_ASTC_8x6_UNORM_BLOCK, VK_FORMAT_ASTC_8x6_SRGB_BLOCK, VK_FORMAT_ASTC_8x8_UNORM_BLOCK, VK_FORMAT_ASTC_8x8_SRGB_BLOCK, VK_FORMAT_ASTC_10x5_UNORM_BLOCK, VK_FORMAT_ASTC_10x5_SRGB_BLOCK, VK_FORMAT_ASTC_10x6_UNORM_BLOCK, VK_FORMAT_ASTC_10x6_SRGB_BLOCK, VK_FORMAT_ASTC_10x8_UNORM_BLOCK, VK_FORMAT_ASTC_10x8_SRGB_BLOCK, VK_FORMAT_ASTC_10x10_UNORM_BLOCK, VK_FORMAT_ASTC_10x10_SRGB_BLOCK, VK_FORMAT_ASTC_12x10_UNORM_BLOCK, VK_FORMAT_ASTC_12x10_SRGB_BLOCK, VK_FORMAT_ASTC_12x12_UNORM_BLOCK, or VK_FORMAT_ASTC_12x12_SRGB_BLOCK
- If the BC texture compression feature is not enabled, `format` must not be VK_FORMAT_BC1_RGB_UNORM_BLOCK, VK_FORMAT_BC1_RGB_SRGB_BLOCK, VK_FORMAT_BC1_RGBA_UNORM_BLOCK, VK_FORMAT_BC1_RGBA_SRGB_BLOCK, VK_FORMAT_BC2_UNORM_BLOCK, VK_FORMAT_BC2_SRGB_BLOCK, VK_FORMAT_BC3_UNORM_BLOCK, VK_FORMAT_BC3_SRGB_BLOCK, VK_FORMAT_BC4_UNORM_BLOCK, VK_FORMAT_BC4_SNORM_BLOCK, VK_FORMAT_BC5_UNORM_BLOCK, VK_FORMAT_BC5_SNORM_BLOCK, VK_FORMAT_BC6H_UFLOAT_BLOCK, VK_FORMAT_BC6H_SFLOAT_BLOCK, VK_FORMAT_BC7_UNORM_BLOCK, or VK_FORMAT_BC7_SRGB_BLOCK
- If the multisampled storage images feature is not enabled, and `usage` contains VK_IMAGE_USAGE_STORAGE_BIT, `samples` must be VK_SAMPLE_COUNT_1_BIT
- If the sparse bindings feature is not enabled, `flags` must not contain VK_IMAGE_CREATE_SPARSE_BINDING_BIT
- If the sparse residency for 2D images feature is not enabled, and `imageType` is VK_IMAGE_TYPE_2D, `flags` must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT
- If the sparse residency for 3D images feature is not enabled, and `imageType` is VK_IMAGE_TYPE_3D, `flags` must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT
- If the sparse residency for images with 2 samples feature is not enabled, `imageType` is VK_IMAGE_TYPE_2D, and `samples` is VK_SAMPLE_COUNT_2_BIT, `flags` must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT
- If the sparse residency for images with 4 samples feature is not enabled, `imageType` is VK_IMAGE_TYPE_2D, and `samples` is VK_SAMPLE_COUNT_4_BIT, `flags` must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT
- If the sparse residency for images with 8 samples feature is not enabled, `imageType` is VK_IMAGE_TYPE_2D, and `samples` is VK_SAMPLE_COUNT_8_BIT, `flags` must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT
- If the sparse residency for images with 16 samples feature is not enabled, `imageType` is VK_IMAGE_TYPE_2D, and `samples` is VK_SAMPLE_COUNT_16_BIT, `flags` must not contain VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT
- If the value of `tiling` is VK_IMAGE_TILING_LINEAR, and the value of VkFormatProperties::`linearTilingFeatures` (as returned by

vkGetPhysicalDeviceFormatProperties with the same value of *format*) does not include VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT, *usage* must not contain VK_IMAGE_USAGE_SAMPLED_BIT

- If the value of *tiling* is VK_IMAGE_TILING_LINEAR, and the value of VkFormatProperties::*linearTilingFeatures* (as returned by **vkGetPhysicalDeviceFormatProperties** with the same value of *format*) does not include VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT, *usage* must not contain VK_IMAGE_USAGE_STORAGE_BIT
- If the value of *tiling* is VK_IMAGE_TILING_LINEAR, and the value of VkFormatProperties::*linearTilingFeatures* (as returned by **vkGetPhysicalDeviceFormatProperties** with the same value of *format*) does not include VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT, *usage* must not contain VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
- If the value of *tiling* is VK_IMAGE_TILING_LINEAR, and the value of VkFormatProperties::*linearTilingFeatures* (as returned by **vkGetPhysicalDeviceFormatProperties** with the same value of *format*) does not include VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT, *usage* must not contain VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
- If the value of *tiling* is VK_IMAGE_TILING_OPTIMAL, and the value of VkFormatProperties::*optimalTilingFeatures* (as returned by **vkGetPhysicalDeviceFormatProperties** with the same value of *format*) does not include VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT, *usage* must not contain VK_IMAGE_USAGE_SAMPLED_BIT
- If the value of *tiling* is VK_IMAGE_TILING_OPTIMAL, and the value of VkFormatProperties::*optimalTilingFeatures* (as returned by **vkGetPhysicalDeviceFormatProperties** with the same value of *format*) does not include VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT, *usage* must not contain VK_IMAGE_USAGE_STORAGE_BIT
- If the value of *tiling* is VK_IMAGE_TILING_OPTIMAL, and the value of VkFormatProperties::*optimalTilingFeatures* (as returned by **vkGetPhysicalDeviceFormatProperties** with the same value of *format*) does not include VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT, *usage* must not contain VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT
- If the value of *tiling* is VK_IMAGE_TILING_OPTIMAL, and the value of VkFormatProperties::*optimalTilingFeatures* (as returned by **vkGetPhysicalDeviceFormatProperties** with the same value of *format*) does not include VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT, *usage* must not contain VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT
- If *flags* contains VK_IMAGE_CREATE_SPARSE_ALIASED_BIT, it must also contain at least one of VK_IMAGE_CREATE_SPARSE_BINDING_BIT or VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT

Valid limits for the image *extent*, *mipLevels*, *arrayLayers* and *samples* members are queried with the `vkGetPhysicalDeviceImageFormatProperties` command.

Images created with *tiling* equal to VK_IMAGE_TILING_LINEAR have further restrictions on their limits and capabilities compared to images created with *tiling* equal to VK_IMAGE_TILING_OPTIMAL. Creation of images with tiling VK_IMAGE_TILING_LINEAR may not be supported unless other parameters meet all of the constraints:

- *imageType* is VK_IMAGE_TYPE_2D
- *format* is not a depth/stencil format
- *mipLevels* is 1
- *arrayLayers* is 1
- *samples* is VK_SAMPLE_COUNT_1_BIT
- *usage* only includes VK_IMAGE_USAGE_TRANSFER_SRC_BIT and/or VK_IMAGE_USAGE_TRANSFER_DST_BIT

Implementations may support additional limits and capabilities beyond those listed above. To determine the specific capabilities of an implementation, query the valid *usage* bits by calling `vkGetPhysicalDeviceFormatProperties` and the valid limits for *mipLevels* and *arrayLayers* by calling `vkGetPhysicalDeviceImageFormatProperties`.

Bits which may be set in *usage* are:

```
typedef enum VkImageUsageFlagBits {
    VK_IMAGE_USAGE_TRANSFER_SRC_BIT = 0x00000001,
    VK_IMAGE_USAGE_TRANSFER_DST_BIT = 0x00000002,
    VK_IMAGE_USAGE_SAMPLED_BIT = 0x00000004,
    VK_IMAGE_USAGE_STORAGE_BIT = 0x00000008,
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT = 0x00000010,
    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000020,
    VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT = 0x00000040,
    VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT = 0x00000080,
} VkImageUsageFlagBits;
```

These bitfields have the following meanings:

- VK_IMAGE_USAGE_TRANSFER_SRC_BIT indicates that the image can be used as the source of a transfer command.
- VK_IMAGE_USAGE_TRANSFER_DST_BIT indicates that the image can be used as the destination of a transfer command.
- VK_IMAGE_USAGE_SAMPLED_BIT indicates that the image can be used to create a VkImageView suitable for occupying a VkDescriptorSet slot either of type VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and be sampled by a shader.
- VK_IMAGE_USAGE_STORAGE_BIT indicates that the image can be used to create a VkImageView suitable for occupying a VkDescriptorSet slot of type VK_DESCRIPTOR_TYPE_STORAGE_IMAGE.
- VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT indicates that the image can be used to create a VkImageView suitable for use as a color or resolve attachment in a VkFramebuffer.
- VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT indicates that the image can be used to create a VkImageView suitable for use as a depth stencil attachment in a VkFramebuffer.

- VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT indicates that the memory bound to this image will have been allocated with the VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT (see Chapter 10 for more detail). If this is set, then bits other than VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, and VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT must not be set.

- VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT indicates that the image can be used to create a VkImageView suitable for occupying VkDescriptorSet slot of type VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT; be read from a shader as an input attachment; and be used as an input attachment in a framebuffer.

Bits which may be set in *flags* are:

```
typedef enum VkImageCreateFlagBits {
    VK_IMAGE_CREATE_SPARSE_BINDING_BIT = 0x00000001,
    VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT = 0x00000002,
    VK_IMAGE_CREATE_SPARSE_ALIASED_BIT = 0x00000004,
    VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT = 0x00000008,
    VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT = 0x00000010,
} VkImageCreateFlagBits;
```

These bitfields have the following meanings:

- VK_IMAGE_CREATE_SPARSE_BINDING_BIT indicates that the image will be backed using sparse memory binding.

- VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT indicates that the image can be partially backed using sparse memory binding.

- VK_IMAGE_CREATE_SPARSE_ALIASED_BIT indicates that the image will be backed using sparse memory binding with memory ranges that might also simultaneously be backing another image (or another portion of the same image). Sparse images created with this flag must also be created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

If any of these three bits are set, VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT must not also be set.

See Sparse Resource Features and Sparse Physical Device Featuers for more details.

- VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT indicates that the image can be used to create a `VkImageView` with a different format from the image.

- VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT indicates that the image can be used to create a `VkImageView` of type VK_IMAGE_VIEW_TYPE_CUBE or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY.

The layout of a subresource (mipLevel/arrayLayer) of an image created with linear tiling is queried by calling:

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

where *device* is the device on which the image was created, and *image* is the image whose layout is being queried. *pSubresource* points to a VkImageSubresource selecting a specific image for the subresource.

Valid Usage

- *device* must be a valid VkDevice handle
- *image* must be a valid VkImage handle
- *pSubresource* must be a pointer to a valid VkImageSubresource structure
- *pLayout* must be a pointer to a VkSubresourceLayout structure
- *image* must have been created, allocated or retrieved from *device*
- Each of *device* and *image* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *image* must have been created with *tiling* equal to VK_IMAGE_TILING_LINEAR
- The *aspectMask* member of *pSubresource* must only have a single bit set

The definition of the VkImageSubresource structure is:

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

- *aspectMask* is a `VkImageAspectFlags` selecting the image aspect.
- *mipLevel* selects the mipmap level.
- *arrayLayer* selects the array layer.

Valid Usage

- *aspectMask* must be a valid combination of `VkImageAspectFlagBits` values
- *aspectMask* must not be `0`

Information about the layout of the subresource is returned in a VkSubresourceLayout structure:

```
typedef struct VkSubresourceLayout {
    VkDeviceSize                                offset;
    VkDeviceSize                                size;
    VkDeviceSize                                rowPitch;
    VkDeviceSize                                arrayPitch;
    VkDeviceSize                                depthPitch;
} VkSubresourceLayout;
```

* `offset` is the byte offset from the start of the image where the subresource begins.
* `size` is the size in bytes of the subresource. `size` includes any extra memory that is required based on the value of `rowPitch`.
* `rowPitch` describes the number of bytes between each row of texels in an image.
* `arrayPitch` describes the number of bytes between each array layer of an image.
* `depthPitch` describes the number of bytes between each slice of 3D image.

For images created with linear tiling, `rowPitch`, `arrayPitch` and `depthPitch` describe the layout of the subresource in linear memory. For uncompressed formats, `rowPitch` is the number of bytes between texels with the same x coordinate in adjacent rows (y coordinates differ by one). `arrayPitch` is the number of bytes between texels with the same x and y coordinate in adjacent array layers of the image (array layer values differ by one). `depthPitch` is the number of bytes between texels with the same x and y coordinate in adjacent slices of a 3D image (z coordinates differ by one). Expressed as an addressing formula, the starting byte of a texel in the subresource has address:

```
// (x,y,z,layer) are in texel coordinates
address(x,y,z,layer) = layer*arrayPitch + z*depthPitch + y*rowPitch + x*texelSize +  ↩
    offset
```

For compressed formats, the `rowPitch` is the number of bytes between compressed blocks in adjacent rows. `arrayPitch` is the number of bytes between blocks in adjacent array layers. `depthPitch` is the number of bytes between blocks in adjacent slices of a 3D image.

```
// (x,y,z,layer) are in block coordinates
address(x,y,z,layer) = layer*arrayPitch + z*depthPitch + y*rowPitch + x*blockSize +  ↩
    offset;
```

`arrayPitch` is undefined for images that were not created as arrays. `depthPitch` is defined only for 3D images.

For color formats, the `aspectMask` member of VkImageSubresource must be VK_IMAGE_ASPECT_COLOR_BIT. For depth/stencil formats, `aspect` must be either VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT. On implementations that store depth and stencil aspects separately, querying each of these subresource layouts will return a different `offset` and `size` representing the region of memory used for that aspect. On implementations that store depth and stencil aspects interleaved, the same `offset` and `size` are returned and represent the interleaved memory allocation.

To destroy an image, call:

```
void vkDestroyImage(
    VkDevice                                    device,
    VkImage                                     image,
    const VkAllocationCallbacks*                pAllocator);
```

`device` is the device to be used to destroy the image. `image` is the handle of the image to destroy. `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *image* is not VK_NULL_HANDLE, *image* must be a valid VkImage handle
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *image* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *image* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to *image*, either directly or via a VkImageView, must have completed execution
- If VkAllocationCallbacks were provided when *image* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *image* was created, *pAllocator* must be `NULL`

Host Synchronization

- Host access to *image* must be externally synchronized

11.4 Image Layouts

Images are stored in implementation-dependent opaque layouts in memory. Implementations may support several opaque layouts, and the layout used at any given time is determined by the `VkImageLayout` state of the subresource. Each layout has limitations on what kinds of operations are supported for subresources using the layout. Applications have control over which layout each image subresource uses, and can transition an image subresource from one layout to another. Transitions can happen with an image memory barrier, included as part of a **`vkCmdPipelineBarrier`** or a **`vkCmdWaitEvents`** command buffer command (see Section 6.5.6), or as part of a subpass dependency within a render pass (see VkSubpassDependency). The image layout state is per-subresource, and separate subresources of the same image can be in different layouts at the same time with one exception - depth and stencil aspects of a given subresource must always be in the same layout.

Note
Each layout may offer optimal performance for a specific usage of image memory. For example, an image with a layout of VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL may provide optimal performance for use as a color attachment, but be unsupported for use in transfer commands. Applications can transition an image subresource from one layout to another in order to achieve optimal performance when the subresource is used for multiple kinds of operations. After initialization, applications need not use any layout other than the general layout, though this may produce suboptimal performance on some implementations.

Upon creation, all subresources of an image are initially in the same layout, where that layout is selected by the VkImageCreateInfo::*initialLayout* member. The *initialLayout* must be either VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED. If it is VK_IMAGE_LAYOUT_PREINITIALIZED, then the image data can be pre-initialized by the host while using this layout, and the transition away from this layout will preserve that data. If it is VK_IMAGE_LAYOUT_UNDEFINED, then the contents of the data are considered to be undefined, and the transition away from this layout is not guaranteed to preserve that data. For either of these initial layouts, any subresources must be transitioned to another layout before they are accessed by the device.

Host access to image memory is only well-defined for images created with VK_IMAGE_TILING_LINEAR tiling and for subresources of those images which are currently in either the VK_IMAGE_LAYOUT_PREINITIALIZED or VK_IMAGE_LAYOUT_GENERAL layout.

The set of image layouts consists of:

```
typedef enum VkImageLayout {
    VK_IMAGE_LAYOUT_UNDEFINED = 0,
    VK_IMAGE_LAYOUT_GENERAL = 1,
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL = 2,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL = 3,
    VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL = 4,
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL = 5,
    VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL = 6,
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL = 7,
    VK_IMAGE_LAYOUT_PREINITIALIZED = 8,
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR = 1000001002,
} VkImageLayout;
```

The type(s) of device access supported by each layout are:

- VK_IMAGE_LAYOUT_UNDEFINED: Supports no device access. This layout must only be used as an *initialLayout* or as the *oldLayout* in an image transition. When transitioning out of this layout, the contents of the memory are not guaranteed to be preserved.
- VK_IMAGE_LAYOUT_PREINITIALIZED: Supports no device access. This layout must only be used as an *initialLayout* or as the *oldLayout* in an image transition. When transitioning out of this layout, the contents of the memory are preserved. This layout is intended to be used as the initial layout for an image whose contents are written by the host, and hence the data can be written to memory immediately, without first executing a layout transition. Currently, VK_IMAGE_LAYOUT_PREINITIALIZED is only useful with VK_IMAGE_TILING_LINEAR images because there is not a standard layout defined for VK_IMAGE_TILING_OPTIMAL images.
- VK_IMAGE_LAYOUT_GENERAL: Supports all types of device access.
- VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL: must only be used as a color or resolve attachment in a VkFramebuffer.
- VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL: must only be used as a depth/stencil attachment in a VkFramebuffer.
- VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL: must only be used as a read-only depth/stencil attachment in a VkFramebuffer and/or as a read-only image in a shader (which can be read as a sampled image, combined image/sampler and/or input attachment).
- VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL: must only be used as a read-only image in a shader (which can be read as a sampled image, combined image/sampler and/or input attachment).
- VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL: must only be used as a source image of a transfer command (see the definition of VK_PIPELINE_STAGE_TRANSFER_BIT).

- VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL: must only be used as a destination image of a transfer command.

For each mechanism of accessing an image in the API, there is a parameter or structure member that controls the image layout used to access the image. For transfer commands, this is a parameter to the command (see Chapter 16 and Chapter 17). For use as a framebuffer attachment, this is a member in the substructures of the VkRenderPassCreateInfo (see Render Pass). For use in a descriptor set, this is a member in the VkDescriptorImageInfo structure (see Section 13.2.4). At the time that any command buffer command accessing an image executes on any queue, the layouts of the image subresources that are accessed must all match the layout specified via the API controlling those accesses.

The image layout of each image subresource must be well-defined at each point in the subresource's lifetime. This means that when performing a layout transition on the subresource, the old layout value must either equal the current layout of the subresource (at the time the transition executes), or else be VK_IMAGE_LAYOUT_UNDEFINED (implying that the contents of the subresource need not be preserved). The new layout used in a transition must not be VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED.

11.5 Image Views

Image objects are not directly accessed by pipeline shaders for reading or writing image data. Instead, *image views* representing contiguous ranges of the image subresources and containing additional metadata are used for that purpose. Views must be created on images of compatible types, and must represent a valid subset of image subresources.

The types of image views that can be created are:

```
typedef enum VkImageViewType {
    VK_IMAGE_VIEW_TYPE_1D = 0,
    VK_IMAGE_VIEW_TYPE_2D = 1,
    VK_IMAGE_VIEW_TYPE_3D = 2,
    VK_IMAGE_VIEW_TYPE_CUBE = 3,
    VK_IMAGE_VIEW_TYPE_1D_ARRAY = 4,
    VK_IMAGE_VIEW_TYPE_2D_ARRAY = 5,
    VK_IMAGE_VIEW_TYPE_CUBE_ARRAY = 6,
} VkImageViewType;
```

The exact image view type is partially implicit, based on the image's type and sample count, as well as the view creation parameters as described in the table below. This table also shows which SPIR-V OpTypeImage Dim and Arrayed parameters correspond to each image view type.

An image view is created by calling **vkCreateImageView**:

```
VkResult vkCreateImageView(
    VkDevice                                    device,
    const VkImageViewCreateInfo*                pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkImageView*                                pView);
```

pAllocator controls host memory allocation as described in the Memory Allocation chapter. Some of the image creation parameters are inherited by the view. The remaining parameters are contained in the *pCreateInfo*.

Valid Usage

- *device* must be a valid VkDevice handle
- *pCreateInfo* must be a pointer to a valid VkImageViewCreateInfo structure
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pView* must be a pointer to a VkImageView handle

The VkImageViewCreateInfo structure is defined as:

```
typedef struct VkImageViewCreateInfo {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkImageViewCreateFlags                          flags;
    VkImage                                         image;
    VkImageViewType                                 viewType;
    VkFormat                                        format;
    VkComponentMapping                              components;
    VkImageSubresourceRange                         subresourceRange;
} VkImageViewCreateInfo;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *image* is a VkImage on which the view will be created.
- *viewType* is the type of the image view.
- *format* is a `VkFormat` describing the format and type used to interpret data elements in the image.
- *components* specifies a remapping of color components (or of depth/stencil components after they have been converted into color components). See `VkComponentMapping`.
- *subresourceRange* selects the set of mipmap levels and array layers to be accessible to the view.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`
- *image* must be a valid VkImage handle

- *`viewType`* must be a valid `VkImageViewType` value
- *`format`* must be a valid `VkFormat` value
- *`components`* must be a valid VkComponentMapping structure
- *`subresourceRange`* must be a valid VkImageSubresourceRange structure
- If *`image`* was not created with VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT then *`viewType`* must not be VK_IMAGE_VIEW_TYPE_CUBE or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY
- If the image cubemap arrays feature is not enabled, *`viewType`* must not be VK_IMAGE_VIEW_TYPE_CUBE_ARRAY
- If the ETC2 texture compression feature is not enabled, *`format`* must not be VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK, VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK, VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK, VK_FORMAT_EAC_R11_UNORM_BLOCK, VK_FORMAT_EAC_R11_SNORM_BLOCK, VK_FORMAT_EAC_R11G11_UNORM_BLOCK, or VK_FORMAT_EAC_R11G11_SNORM_BLOCK
- If the ASTC LDR texture compression feature is not enabled, *`format`* must not be VK_FORMAT_ASTC_4x4_UNORM_BLOCK, VK_FORMAT_ASTC_4x4_SRGB_BLOCK, VK_FORMAT_ASTC_5x4_UNORM_BLOCK, VK_FORMAT_ASTC_5x4_SRGB_BLOCK, VK_FORMAT_ASTC_5x5_UNORM_BLOCK, VK_FORMAT_ASTC_5x5_SRGB_BLOCK, VK_FORMAT_ASTC_6x5_UNORM_BLOCK, VK_FORMAT_ASTC_6x5_SRGB_BLOCK, VK_FORMAT_ASTC_6x6_UNORM_BLOCK, VK_FORMAT_ASTC_6x6_SRGB_BLOCK, VK_FORMAT_ASTC_8x5_UNORM_BLOCK, VK_FORMAT_ASTC_8x5_SRGB_BLOCK, VK_FORMAT_ASTC_8x6_UNORM_BLOCK, VK_FORMAT_ASTC_8x6_SRGB_BLOCK, VK_FORMAT_ASTC_8x8_UNORM_BLOCK, VK_FORMAT_ASTC_8x8_SRGB_BLOCK, VK_FORMAT_ASTC_10x5_UNORM_BLOCK, VK_FORMAT_ASTC_10x5_SRGB_BLOCK, VK_FORMAT_ASTC_10x6_UNORM_BLOCK, VK_FORMAT_ASTC_10x6_SRGB_BLOCK, VK_FORMAT_ASTC_10x8_UNORM_BLOCK, VK_FORMAT_ASTC_10x8_SRGB_BLOCK, VK_FORMAT_ASTC_10x10_UNORM_BLOCK, VK_FORMAT_ASTC_10x10_SRGB_BLOCK, VK_FORMAT_ASTC_12x10_UNORM_BLOCK, VK_FORMAT_ASTC_12x10_SRGB_BLOCK, VK_FORMAT_ASTC_12x12_UNORM_BLOCK, or VK_FORMAT_ASTC_12x12_SRGB_BLOCK
- If the BC texture compression feature is not enabled, *`format`* must not be VK_FORMAT_BC1_RGB_UNORM_BLOCK, VK_FORMAT_BC1_RGB_SRGB_BLOCK, VK_FORMAT_BC1_RGBA_UNORM_BLOCK, VK_FORMAT_BC1_RGBA_SRGB_BLOCK, VK_FORMAT_BC2_UNORM_BLOCK, VK_FORMAT_BC2_SRGB_BLOCK, VK_FORMAT_BC3_UNORM_BLOCK, VK_FORMAT_BC3_SRGB_BLOCK, VK_FORMAT_BC4_UNORM_BLOCK, VK_FORMAT_BC4_SNORM_BLOCK, VK_FORMAT_BC5_UNORM_BLOCK, VK_FORMAT_BC5_SNORM_BLOCK, VK_FORMAT_BC6H_UFLOAT_BLOCK, VK_FORMAT_BC6H_SFLOAT_BLOCK, VK_FORMAT_BC7_UNORM_BLOCK, or VK_FORMAT_BC7_SRGB_BLOCK
- If *`image`* was created with VK_IMAGE_TILING_LINEAR and *`usage`* containing VK_IMAGE_USAGE_SAMPLED_BIT, *`format`* must be supported for sampled images, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT flag in VkFormatProperties::*`linearTilingFeatures`* returned by **`vkGetPhysicalDeviceFormatProperties`**
- If *`image`* was created with VK_IMAGE_TILING_LINEAR and *`usage`* containing VK_IMAGE_USAGE_STORAGE_BIT, *`format`* must be supported for storage images, as specified by the VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT flag in VkFormatProperties::*`linearTilingFeatures`* returned by **`vkGetPhysicalDeviceFormatProperties`**

- If *image* was created with VK_IMAGE_TILING_LINEAR and *usage* containing VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, *format* must be supported for color attachments, as specified by the VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT flag in VkFormatProperties::*linearTilingFeatures* returned by **vkGetPhysicalDeviceFormatProperties**
- If *image* was created with VK_IMAGE_TILING_LINEAR and *usage* containing VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, *format* must be supported for depth/stencil attachments, as specified by the VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT flag in VkFormatProperties::*linearTilingFeatures* (as returned by **vkGetPhysicalDeviceFormatProperties**
- If *image* was created with VK_IMAGE_TILING_OPTIMAL and *usage* containing VK_IMAGE_USAGE_SAMPLED_BIT, *format* must be supported for sampled images, as specified by the VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT flag in VkFormatProperties::*optimalTilingFeatures* returned by **vkGetPhysicalDeviceFormatProperties**
- If *image* was created with VK_IMAGE_TILING_OPTIMAL and *usage* containing VK_IMAGE_USAGE_STORAGE_BIT, *format* must be supported for storage images, as specified by the VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT flag in VkFormatProperties::*optimalTilingFeatures* returned by **vkGetPhysicalDeviceFormatProperties**
- If *image* was created with VK_IMAGE_TILING_OPTIMAL and *usage* containing VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT, *format* must be supported for color attachments, as specified by the VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT flag in VkFormatProperties::*optimalTilingFeatures* returned by **vkGetPhysicalDeviceFormatProperties**
- If *image* was created with VK_IMAGE_TILING_OPTIMAL and *usage* containing VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT, *format* must be supported for depth/stencil attachments, as specified by the VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT flag in VkFormatProperties::*optimalTilingFeatures* returned by **vkGetPhysicalDeviceFormatProperties**
- *subresourceRange* must be a valid subresource range for *image* (see Section 11.5)
- If *image* was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, *format* must be compatible with the *format* used to create *image*, as defined in Format Compatibility Classes
- If *image* was not created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, *format* must be identical to the *format* used to create *image*
- *subResourceRange* and *viewType* must be compatible with the image, as described in the table below

If *image* was created with the VK_IMAGE_CREATE_MUTABLE_FORMAT_BIT flag, *format* can be different from the image's format, but if they are not equal they must be *compatible*. Image format compatibility is defined in the Format Compatibility Classes section.

Table 11.1: Image and image view parameter compatibility requirements

Table 11.1: (continued)

Dim, Arrayed, MS	Image parameters	View parameters

1D, 0, 0	imageType = IMAGE_TYPE_1D
	width >= 1
	height = 1
	depth = 1
	arrayLayers >= 1
	samples = 1
		viewType = VIEW_TYPE_1D
		baseArrayLayer >= 0
		arrayLayers = 1

1D, 1, 0	imageType = IMAGE_TYPE_1D
	width >= 1
	height = 1
	depth = 1
	arrayLayers >= 1
	samples = 1
		viewType = VIEW_TYPE_1D_
		ARRAY
		baseArrayLayer >= 0
		arrayLayers >= 1

2D, 0, 0	imageType = IMAGE_TYPE_2D
	width >= 1
	height >= 1
	depth = 1
	arrayLayers >= 1
	samples = 1
		viewType = VIEW_TYPE_2D
		baseArrayLayer >= 0
		arrayLayers = 1

2D, 1, 0	imageType = IMAGE_TYPE_2D
	width >= 1
	height >= 1
	depth = 1
	arrayLayers >= 1
	samples = 1
		viewType = VIEW_TYPE_2D_
		ARRAY
		baseArrayLayer >= 0
		arrayLayers >= 1

2D, 0, 1	imageType = IMAGE_TYPE_2D width >= 1 height >= 1 depth = 1 arrayLayers >= 1 samples > 1	viewType = VIEW_TYPE_2D baseArrayLayer >= 0 arrayLayers = 1
2D, 1, 1	imageType = IMAGE_TYPE_2D width >= 1 height >= 1 depth = 1 arrayLayers >= 1 samples > 1	viewType = VIEW_TYPE_2D_ ARRAY baseArrayLayer >= 0 arrayLayers >= 1

CUBE, 0, 0	imageType = IMAGE_TYPE_2D
	width >= 1
	height = width
	depth = 1
	arrayLayers >= 6
	samples = 1
	flags include VK_IMAGE_CREATE_
	CUBE_COMPATIBLE_BIT
		viewType = VIEW_TYPE_CUBE
		baseArrayLayer >= 0
		arrayLayers = 6

Table 11.1: (continued)

`CUBE, 1, 0`	imageType = IMAGE_TYPE_2D
	width >= 1
	height = width
	depth = 1
	arrayLayers >= 6*N
	samples = 1
	flags include VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT
		viewType = VIEW_TYPE_CUBE_ARRAY
		baseArrayLayer >= 0
		arrayLayers = 6*N

`3D, 0, 0`	imageType = IMAGE_TYPE_3D
	width >= 1
	height >= 1
	depth >= 1
	arrayLayers = 1
	samples = 1
		viewType = VIEW_TYPE_3D
		baseArrayLayer = 0
		arrayLayers = 1

The *subresourceRange* member is of type VkImageSubresourceRange and is defined as:

```
typedef struct VkImageSubresourceRange {
    VkImageAspectFlags                                aspectMask;
    uint32_t                                          baseMipLevel;
    uint32_t                                          levelCount;
    uint32_t                                          baseArrayLayer;
    uint32_t                                          layerCount;
} VkImageSubresourceRange;
```

- *aspectMask* is a bitmask indicating which aspect(s) of the image are included in the view. See `VkImageAspectFlagBits`.
- *baseMipLevel* is the first mipmap level accessible to the view.
- *levelCount* is the number of mipmap levels (starting from *baseMipLevel*) accessible to the view.
- *baseArrayLayer* is the first array layer accessible to the view.
- *layerCount* is the number of array layers (starting from *baseArrayLayer*) accessible to the view.

Valid Usage

- *aspectMask* must be a valid combination of `VkImageAspectFlagBits` values
- *aspectMask* must not be `0`

The number of mip-map levels and array layers must be a subset of the subresources in the image. If an application wants to use all mip-levels or layers in an image after the *baseMipLevel* or *baseArrayLayer*, it can set *levelCount* and *layerCount* to the special values VK_REMAINING_MIP_LEVELS and VK_REMAINING_ARRAY_LAYERS without knowing the exact number of mip-levels or layers.

For cube and cube array image views, the layers of the image view starting at *baseArrayLayer* correspond to faces in the order +X, -X, +Y, -Y, +Z, -Z. For cube arrays, each set of six sequential layers is a single cube, so the number of cube maps in a cube map array view is `layerCount` */ 6*, and image array layer `baseArrayLayer` + *i* is face index *i mod 6* of cube *i / 6*. If the number of layers in the view, whether set explicitly in *layerCount* or implied by VK_REMAINING_ARRAY_LAYERS, is not a multiple of 6, behavior when indexing the last cube is undefined.

aspectMask is a bitmask indicating the format being used. Bits which may be set include:

```
typedef enum VkImageAspectFlagBits {
    VK_IMAGE_ASPECT_COLOR_BIT = 0x00000001,
    VK_IMAGE_ASPECT_DEPTH_BIT = 0x00000002,
    VK_IMAGE_ASPECT_STENCIL_BIT = 0x00000004,
    VK_IMAGE_ASPECT_METADATA_BIT = 0x00000008,
} VkImageAspectFlagBits;
```

The mask must be only VK_IMAGE_ASPECT_COLOR_BIT, VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT if *format* is a color, depth-only or stencil-only format, respectively. If using a depth/stencil format, *aspectMask* must include at least one of VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT, and can include both.

When using an imageView of a depth/stencil image to populate a descriptor set (e.g. for sampling in the shader, or for use as an input attachment), the *aspectMask* must only include one bit and selects whether the imageView is used for depth reads (i.e. using a floating-point sampler or input attachment in the shader) or stencil reads (i.e. using an unsigned integer sampler or input attachment in the shader). When an imageView of a depth/stencil image is used as a depth/stencil framebuffer attachment, the *aspectMask* is ignored and both depth and stencil subresources are used.

The *components* member is defined as follows:

```
typedef struct VkComponentMapping {
    VkComponentSwizzle                          r;
    VkComponentSwizzle                          g;
    VkComponentSwizzle                          b;
    VkComponentSwizzle                          a;
} VkComponentMapping;
```

and describes a remapping from components of the image to components of the vector returned by shader image instructions. This remapping must be identity for storage image descriptors, input attachment descriptors, and framebuffer attachments. The *r*, *g*, *b*, and *a* members of *components* are the values placed in the corresponding components of the output vector:

```
typedef enum VkComponentSwizzle {
    VK_COMPONENT_SWIZZLE_IDENTITY = 0,
    VK_COMPONENT_SWIZZLE_ZERO = 1,
    VK_COMPONENT_SWIZZLE_ONE = 2,
    VK_COMPONENT_SWIZZLE_R = 3,
    VK_COMPONENT_SWIZZLE_G = 4,
    VK_COMPONENT_SWIZZLE_B = 5,
    VK_COMPONENT_SWIZZLE_A = 6,
} VkComponentSwizzle;
```

- VK_COMPONENT_SWIZZLE_IDENTITY: the component is set to the identity swizzle.
- VK_COMPONENT_SWIZZLE_ZERO: the component is set to zero.
- VK_COMPONENT_SWIZZLE_ONE: the component is set to either 1 or 1.0 depending on whether the type of the image view format is integer or floating-point respectively, as determined by the Format Definition section for each `VkFormat`.

- VK_COMPONENT_SWIZZLE_R: the component is set to the value of the R component of the image.
- VK_COMPONENT_SWIZZLE_G: the component is set to the value of the G component of the image.
- VK_COMPONENT_SWIZZLE_B: the component is set to the value of the B component of the image.
- VK_COMPONENT_SWIZZLE_A: the component is set to the value of the A component of the image.

Valid Usage

- `r` must be a valid `VkComponentSwizzle` value
- `g` must be a valid `VkComponentSwizzle` value
- `b` must be a valid `VkComponentSwizzle` value
- `a` must be a valid `VkComponentSwizzle` value

Setting the identity swizzle on a component is equivalent to setting the identity mapping on that component. That is:

Table 11.2: Component Mappings Equivalent To VK_COMPONENT_SWIZZLE_IDENTITY

Component	**Identity Mapping**
`components.r`	VK_COMPONENT_SWIZZLE_R
`components.g`	VK_COMPONENT_SWIZZLE_G
`components.b`	VK_COMPONENT_SWIZZLE_B
`components.a`	VK_COMPONENT_SWIZZLE_A

To destroy an image view, call:

```
void vkDestroyImageView(
    VkDevice                                    device,
    VkImageView                                 imageView,
    const VkAllocationCallbacks*                pAllocator);
```

`device` is the device to be used to destroy the image view. `imageView` is the handle of the image view to destroy. `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- `device` must be a valid VkDevice handle

- If *imageView* is not VK_NULL_HANDLE, *imageView* must be a valid VkImageView handle
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *imageView* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *imageView* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to *imageView* must have completed execution
- If VkAllocationCallbacks were provided when *imageView* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *imageView* was created, *pAllocator* must be `NULL`

Host Synchronization

- Host access to *imageView* must be externally synchronized

11.6 Resource Memory Association

Resources are initially created as *virtual allocations* with no backing memory. Device memory is allocated separately (see Section 10.2) and then associated with the resource. This association is done differently for sparse and non-sparse resources.

Resources created with any of the sparse creation flags are considered sparse resources. Resources created without these flags are non-sparse. The details on resource memory association for sparse resources is described in Chapter 27.

Non-sparse resources must be bound completely and contiguously to a single `VkDeviceMemory` object before the resource is referenced by any of the following operations:

- creating image or buffer views
- updating descriptor sets
- recording commands in a command buffer

Once bound, the memory binding is immutable for the lifetime of the resource.

To determine the memory requirements for a non-sparse Vulkan resource, use the **`vkGetBufferMemoryRequirements`** and **`vkGetImageMemoryRequirements`** commands, whose prototypes are:

```
void vkGetBufferMemoryRequirements(
    VkDevice                                    device,
    VkBuffer                                    buffer,
    VkMemoryRequirements*                       pMemoryRequirements);
```

Valid Usage

- *device* must be a valid VkDevice handle
- *buffer* must be a valid VkBuffer handle
- *pMemoryRequirements* must be a pointer to a VkMemoryRequirements structure
- *buffer* must have been created, allocated or retrieved from *device*
- Each of *device* and *buffer* must have been created, allocated or retrieved from the same VkPhysicalDevice

```
void vkGetImageMemoryRequirements(
    VkDevice                                    device,
    VkImage                                     image,
    VkMemoryRequirements*                       pMemoryRequirements);
```

Valid Usage

- *device* must be a valid VkDevice handle
- *image* must be a valid VkImage handle
- *pMemoryRequirements* must be a pointer to a VkMemoryRequirements structure
- *image* must have been created, allocated or retrieved from *device*
- Each of *device* and *image* must have been created, allocated or retrieved from the same VkPhysicalDevice

device is the handle to the device that owns the resource, whose handle is specified in *buffer* or *image*. *pMemoryRequirements* points to a VkMemoryRequirements structure, whose contents will be overwritten with the memory requirements of the resource object, and whose definition is:

```
typedef struct VkMemoryRequirements {
    VkDeviceSize                                size;
    VkDeviceSize                                alignment;
    uint32_t                                    memoryTypeBits;
} VkMemoryRequirements;
```

The members of the VkMemoryRequirements structure have the following meanings:

- *size* is the size, in bytes, of the memory allocation required for the resource.
- *alignment* is the alignment, in bytes, of the offset within the allocation required for the resource.
- *memoryTypeBits* is a bitfield and contains one bit set for every supported memory type for the resource. Bit `i` is set if and only if the memory type `i` in the VkPhysicalDeviceMemoryProperties structure for the physical device is supported for the resource.

The implementation guarantees certain properties about the returned values:

- The *memoryTypeBits* member always contains at least one bit set.
- If *buffer* is a VkBuffer, or if *image* is a VkImage that was created with a VK_IMAGE_TILING_LINEAR value in the *tiling* member of the VkImageCreateInfo structure passed to **vkCreateImage**, then the *memoryTypeBits* member always contains at least one bit set corresponding to a VkMemoryType with a *propertyFlags* that has both the VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT bit and the VK_ MEMORY_PROPERTY_HOST_COHERENT_BIT bit set. In other words, mappable coherent memory can always be attached to these objects.
- The value of the *memoryTypeBits* member is identical for all VkBuffer objects created with the same value for the *flags* and *usage* members in the VkBufferCreateInfo structure passed to **vkCreateBuffer**. Further, if **usage1** and **usage2** of type `VkBufferUsageFlags` are such that **usage2** contains a subset of the bits set in **usage1** and they have the same value of *flags*, then the bits set in the value of *memoryTypeBits* returned for **usage1** must be a subset of the bits set in the value of *memoryTypeBits* returned for **usage2**, for all values of *flags*.
- The value of the *alignment* member is identical for all VkBuffer objects created with the same combination of values for the *usage* and *flags* members in the VkBufferCreateInfo structure passed to **vkCreateBuffer**.
- The value of the *memoryTypeBits* member is identical for all VkImage objects created with the same combination of values for the *tiling* member and the VK_IMAGE_CREATE_SPARSE_BINDING_BIT bit of the *flags* member and the VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT of the *usage* member in the VkImageCreateInfo structure passed to **vkCreateImage**.
- The *memoryTypeBits* member must not refer to a VkMemoryType with a *propertyFlags* that has the VK_ MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set if the VkImage does not have VK_IMAGE_ USAGE_TRANSIENT_ATTACHMENT_BIT bit set in the *usage* member of the VkImageCreateInfo structure passed to **vkCreateImage**.

To attach memory to a buffer object, call **vkBindBufferMemory**, whose prototype is:

```
VkResult vkBindBufferMemory(
    VkDevice                                    device,
    VkBuffer                                    buffer,
    VkDeviceMemory                              memory,
    VkDeviceSize                                memoryOffset);
```

Valid Usage

- `device` must be a valid VkDevice handle
- `buffer` must be a valid VkBuffer handle
- `memory` must be a valid VkDeviceMemory handle
- `buffer` must have been created, allocated or retrieved from `device`
- `memory` must have been created, allocated or retrieved from `device`
- Each of `device`, `buffer` and `memory` must have been created, allocated or retrieved from the same VkPhysicalDevice
- `buffer` must not already be backed by a memory object
- `buffer` must not have been created with any sparse memory binding flags
- `memoryOffset` must be less than the size of `memory`
- If `buffer` was created with the VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT or VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT, `memoryOffset` must be a multiple of the value of VkPhysicalDeviceLimits::`minTexelBufferOffsetAlignment`
- If `buffer` was created with the VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT, `memoryOffset` must be a multiple of the value of VkPhysicalDeviceLimits::`minUniformBufferOffsetAlignment`
- If `buffer` was created with the VK_BUFFER_USAGE_STORAGE_BUFFER_BIT, `memoryOffset` must be a multiple of the value of VkPhysicalDeviceLimits::`minStorageBufferOffsetAlignment`
- `memory` must have been allocated using one of the memory types allowed in the `memoryTypeBits` member of the VkMemoryRequirements structure returned from a call to **`vkGetBufferMemoryRequirements`** with `buffer`
- The sum of `memoryOffset` and the size of `buffer` must be less than or equal to the size of `memory`
- `memoryOffset` must be an integer multiple of the `alignment` member of the VkMemoryRequirements structure returned from a call to **`vkGetBufferMemoryRequirements`** with `buffer`

Host Synchronization

- Host access to `buffer` must be externally synchronized

To attach memory to a image object, call **`vkBindImageMemory`**, whose prototype is:

```
VkResult vkBindImageMemory(
    VkDevice                                    device,
    VkImage                                     image,
    VkDeviceMemory                              memory,
    VkDeviceSize                                memoryOffset);
```

Valid Usage

- *device* must be a valid VkDevice handle
- *image* must be a valid VkImage handle
- *memory* must be a valid VkDeviceMemory handle
- *image* must have been created, allocated or retrieved from *device*
- *memory* must have been created, allocated or retrieved from *device*
- Each of *device*, *image* and *memory* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *image* must not already be backed by a memory object
- *image* must not have been created with any sparse memory binding flags
- *memoryOffset* must be less than the size of *memory*
- *memory* must have been allocated using one of the memory types allowed in the *memoryTypeBits* member of the VkMemoryRequirements structure returned from a call to **vkGetImageMemoryRequirements** with *image*
- *memoryOffset* must be an integer multiple of the *alignment* member of the VkMemoryRequirements structure returned from a call to **vkGetImageMemoryRequirements** with *image*
- *memory* must have storage from *memoryOffset* onwards equal to or greater than the *size* member of the VkMemoryRequirements structure returned from a call to **vkGetImageMemoryRequirements** with *image*

Host Synchronization

- Host access to *image* must be externally synchronized

The *device* parameter specifies the device used to allocate the object and the memory. The *memory* parameter specifies the memory object to bind. *memoryOffset* is the start offset of the region of *memory* which is to be bound to the object. The number of bytes returned in the *size* member of the VkMemoryRequirements structure in *memory* starting from *memoryOffset* bytes will be bound to the specified buffer or image.

Buffer-Image Granularity

Additionally there is an implementation-dependent limit, *bufferImageGranularity*, which specifies a page-like granularity at which buffer and image resources must be placed in adjacent memory locations for simultaneous usage. If two resources do not satisfy this granularity, then the resources alias. The *bufferImageGranularity* is specified in bytes, and must be a power of two. Implementations which do not require such an additional granularity would report a value of one.

Given resourceA at the lower memory offset and resourceB at the higher memory offset, where one of the resources is a buffer and the other is an image, and the following:

```
resourceA.end       = resourceA.memoryOffset + resourceA.size - 1
resourceA.endPage   = resourceA.end & ~(bufferImageGranularity-1)
resourceB.start     = resourceB.memoryOffset
resourceB.startPage = resourceB.start & ~(bufferImageGranularity-1)
```

The following property must hold:

```
resourceA.endPage < resourceB.startPage
```

That is, the end of the first resource (A) and the beginning of the second resource (B) must be on separate *pages* of size *bufferImageGranularity*. *bufferImageGranularity* may be different than the physical page size of the memory heap. This restriction is only needed for adjacent image and buffer memory locations which will be used simultaneously. Adjacent buffers' or adjacent images' memory ranges can be closer than *bufferImageGranularity*, provided they meet the *alignment* requirement for the objects in question.

Sparse memory block sizes and sparse image and buffer memory alignments must all be multiples of the *bufferImageGranularity*. Therefore, memory bound to sparse resources naturally satisfies the *bufferImageGranularity*.

11.7 Resource Sharing Mode

Buffer and image objects are created with a *sharing mode* controlling how they can be accessed from queues. The supported sharing modes are:

```
typedef enum VkSharingMode {
    VK_SHARING_MODE_EXCLUSIVE = 0,
    VK_SHARING_MODE_CONCURRENT = 1,
} VkSharingMode;
```

- VK_SHARING_MODE_EXCLUSIVE specifies that access to any range or subresource of the object will be exclusive to a single queue family at a time.
- VK_SHARING_MODE_CONCURRENT specifies that concurrent access to any range or subresource of the object from multiple queue families is supported.

Note
VK_SHARING_MODE_CONCURRENT may result in lower performance access to the buffer or image than VK_SHARING_MODE_EXCLUSIVE.

Ranges of buffers and subresources of image objects created using VK_SHARING_MODE_EXCLUSIVE must only be accessed by queues in the same queue family at any given time. In order for a different queue family to be able to interpret the memory contents of a range or subresource, the application must transfer exclusive ownership of the range or subresource between the source and destination queue families with the following sequence of operations:

1. Release exclusive ownership from the source queue family to the destination queue family.
2. Use semaphores to ensure proper execution control for the ownership transfer.
3. Acquire exclusive ownership for the destination queue family from the source queue family.

To release exclusive ownership of a range of a buffer or subresource of an image object, the application must execute a buffer or image memory barrier, respectively (see `VkBufferMemoryBarrier` and `VkImageMemoryBarrier`) on a queue from the source queue family. The *`srcQueueFamilyIndex`* parameter of the barrier must be set to the source queue family index, and the *`dstQueueFamilyIndex`* parameter to the destination queue family index.

To acquire exclusive ownership, the application must execute the same buffer or image memory barrier on a queue from the destination queue family.

Upon creation, resources using VK_SHARING_MODE_EXCLUSIVE are not owned by any queue family. A buffer or image memory barrier is not required to acquire ownership when no queue family owns the resource - it is implicitly acquired upon first use within a queue. However, images still require a layout transition from VK_IMAGE_LAYOUT_UNDEFINED or VK_IMAGE_LAYOUT_PREINITIALIZED before being used on the first queue. This layout transition can either be accomplished by an image memory barrier or by use in a render pass instance.

Once a queue family has used a range or subresource of an VK_SHARING_MODE_EXCLUSIVE resource, its contents are undefined to other queue families unless ownership is transferred. The contents may also become undefined for other reasons, e.g. as a result of writes to an image subresource that aliases the same memory. A queue family can take ownership of a range or subresource without an ownership transfer in the same way as for a resource that was just created, however doing so means any contents written by other queue families or via incompatible aliases are undefined.

11.8 Memory Aliasing

A range of a VkDeviceMemory allocation is *aliased* if it is bound to multiple resources simultaneously, via `vkBindImageMemory`, `vkBindBufferMemory`, or via sparse memory bindings. A memory range aliased between two images or two buffers is defined to be the intersection of the memory ranges bound to the two resources. A memory range aliased between an image and a buffer is defined to be the intersection of the memory ranges bound to the two resources, where each range is first bloated to be aligned to the *`bufferImageGranularity`*. Applications can alias memory, but use of multiple aliases is subject to several constraints.

Note
Memory aliasing can be useful to reduce the total device memory footprint of an application, if some large resources are used for disjoint periods of time.

When an opaque, non-VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT image is bound to an aliased range, all subresources of the image *overlap* the range. When a linear image is bound to an aliased range, the subresources that (according to the image's advertised layout) include bytes from the aliased range overlap the range. When a VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT image has blocks bound to an aliased range, only subresources including those blocks overlap the range, and when the memory bound to the image's miptail overlaps an aliased range all subresources in the miptail overlap the range.

Buffers, and linear image subresources in either the VK_IMAGE_LAYOUT_PREINITIALIZED or VK_IMAGE_LAYOUT_GENERAL layouts, are *host-accessible subresources*. That is, the host has a well-defined addressing scheme to interpret the contents, and thus the layout of the data in memory can be consistently interpreted across aliases if each of those aliases is a host-accessible subresource. Opaque images and linear image subresources in other layouts are not host-accessible.

If two aliases are both host-accessible, then they interpret the contents of the memory in consistent ways, and data written to one alias can be read by the other alias.

If either of two aliases is not host-accessible, then the aliases interpret the contents of the memory differently, and writes via one alias make the contents of memory partially or completely undefined to the other alias. If the first alias is a host-accessible subresource, then the bytes affected are those written by the memory operations according to its addressing scheme. If the first alias is not host-accessible, then the bytes affected are those overlapped by the image subresources that were written. If the second alias is a host-accessible subresource, the affected bytes become undefined. If the second alias is a not host-accessible, all sparse blocks (for sparse residency images) or all subresources (for non-sparse residency images) that overlap the affected bytes become undefined.

If any subresources are made undefined due to writes to an alias, then each of those subresources must have its layout transitioned from VK_IMAGE_LAYOUT_UNDEFINED to a valid layout before it is used, or from VK_IMAGE_LAYOUT_PREINITIALIZED if the memory has been written by the host. If any blocks of a sparse image have been made undefined, then only the subresources containing them must be transitioned.

Use of an overlapping range by two aliases must be separated by a memory dependency using the appropriate access types if at least one of those uses performs writes, whether the aliases interpret memory consistently or not. If buffer or image memory barriers are used, the scope of the barrier must contain the entire range and/or set of subresources that overlap.

If two aliasing image views are used in the same framebuffer, then the render pass must declare the attachments using the VK_ATTACHMENT_DESCRIPTION_MAY_ALIAS_BIT, and follow the other rules listed in that section.

Access to resources which alias memory from shaders using variables decorated with **`Coherent`** are not automatically coherent with each other.

Note
Memory recycled via an application suballocator (i.e. without freeing and reallocating the memory objects) is not substantially different from memory aliasing. However, a suballocator usually waits on a fence before recycling a region of memory, and signalling a fence involves enough implicit ordering that the above requirements are all satisfied.

Chapter 12

Samplers

VkSampler objects encapsulate the state of an image sampler which is used by the implementation to read image data and apply filtering and other transformations for the shader.

To create a VkSampler object, call:

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

The VkSamplerCreateInfo struct specifies the state of the VkSampler object. `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- `device` must be a valid VkDevice handle
- `pCreateInfo` must be a pointer to a valid VkSamplerCreateInfo structure
- If `pAllocator` is not `NULL`, `pAllocator` must be a pointer to a valid VkAllocationCallbacks structure
- `pSampler` must be a pointer to a VkSampler handle

The VkSamplerCreateInfo structure is defined as follows:

```
typedef struct 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;
```

The members of VkSamplerCreateInfo are described as follows:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `magFilter` is the magnification filter to apply to lookups, and is of type:

  ```
  typedef enum VkFilter {
      VK_FILTER_NEAREST = 0,
      VK_FILTER_LINEAR = 1,
  } VkFilter;
  ```

- `minFilter` is the minification filter to apply to lookups, and is of type `VkFilter`.
- `mipmapMode` is the mipmap filter to apply to lookups as described in the Texel Filtering section, and is of type:

  ```
  typedef enum VkSamplerMipmapMode {
      VK_SAMPLER_MIPMAP_MODE_BASE = 0,
      VK_SAMPLER_MIPMAP_MODE_NEAREST = 1,
      VK_SAMPLER_MIPMAP_MODE_LINEAR = 2,
  } VkSamplerMipmapMode;
  ```

- `addressModeU` is the addressing mode for outside [0..1] range for U coordinate. See `VkSamplerAddressMode`.
- `addressModeV` is the addressing mode for outside [0..1] range for V coordinate. See `VkSamplerAddressMode`.
- `addressModeW` is the addressing mode for outside [0..1] range for W coordinate. See `VkSamplerAddressMode`.
- `mipLodBias` is the bias to be added to mipmap LOD calculation and bias provided by image sampling functions in SPIR-V, as described in the Level-of-Detail Operation section.
- `anisotropyEnable` is VK_TRUE to enable anisotropic filtering, as described in the Anisotropic Texel Selection section, or VK_FALSE otherwise.
- `maxAnisotropy` is the anisotropy value clamp.
- `compareEnable` is VK_TRUE to enable comparison against a reference value during lookups, or VK_FALSE otherwise.

  - Note: Some implementations will default to shader state if this member does not match.

- *compareOp* is the comparison function to apply to fetched data before filtering as described in the Depth Compare Operation section. See `VkCompareOp`.
- *minLod* and *maxLod* are the values used to clamp the computed level-of-detail value, as described in the Level-of-Detail Operation section. *maxLod* must be greater than or equal to *minLod*.
- *borderColor* is the predefined border color to use, as described in the Texel Replacement section, and is of type:

```
typedef enum VkBorderColor {
    VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK = 0,
    VK_BORDER_COLOR_INT_TRANSPARENT_BLACK = 1,
    VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK = 2,
    VK_BORDER_COLOR_INT_OPAQUE_BLACK = 3,
    VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE = 4,
    VK_BORDER_COLOR_INT_OPAQUE_WHITE = 5,
} VkBorderColor;
```

- *unnormalizedCoordinates* controls whether to use unnormalized or normalized texel coordinates to address texels of the image. When set to VK_TRUE, the range of the image coordinates used to lookup the texel is in the range of zero to the image dimensions for x, y and z. When set to VK_FALSE the range of image coordinates is zero to one. When *unnormalizedCoordinates* is VK_TRUE, samplers have the following requirements:
  - *minFilter* and *magFilter* must be equal.
  - *mipmapMode* must be VK_SAMPLER_MIPMAP_MODE_BASE.
  - *addressModeU* and *addressModeV* must each be either VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE or VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER.
  - *anisotropyEnable* must be VK_FALSE.
  - *compareEnable* must be VK_FALSE.
- When *unnormalizedCoordinates* is VK_TRUE, images the sampler is used with in the shader have the following requirements:
  - The *viewType* must be either VK_IMAGE_VIEW_TYPE_1D or VK_IMAGE_VIEW_TYPE_2D.
  - The image view must have a single layer and a single mip level.
- When *unnormalizedCoordinates* is VK_TRUE, image built-in functions in the shader that use the sampler have the following requirements:
  - The functions must not use projection.
  - The functions must not use offsets.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO
- *pNext* must be `NULL`

- *flags* must be 0
- *magFilter* must be a valid VkFilter value
- *minFilter* must be a valid VkFilter value
- *mipmapMode* must be a valid VkSamplerMipmapMode value
- *addressModeU* must be a valid VkSamplerAddressMode value
- *addressModeV* must be a valid VkSamplerAddressMode value
- *addressModeW* must be a valid VkSamplerAddressMode value
- The absolute value of *mipLodBias* must be less than or equal to VkPhysicalDeviceLimits::*maxSamplerLodBias*
- If the anisotropic sampling feature is not enabled, *anisotropyEnable* must be VK_FALSE
- If *anisotropyEnable* is VK_TRUE, the value of *maxAnisotropy* must be between 1.0 and VkPhysicalDeviceLimits::*maxSamplerAnisotropy*, inclusive
- If *unnormalizedCoordinates* is VK_TRUE, *minFilter* and *magFilter* must be equal
- If *unnormalizedCoordinates* is VK_TRUE, *mipmapMode* must be VK_SAMPLER_MIPMAP_MODE_BASE
- If *unnormalizedCoordinates* is VK_TRUE, *addressModeU* and *addressModeV* must each be either VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE or VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER
- If *unnormalizedCoordinates* is VK_TRUE, *anisotropyEnable* must be VK_FALSE
- If *unnormalizedCoordinates* is VK_TRUE, *compareEnable* must be VK_FALSE
- If any of *addressModeU*, *addressModeV* or *addressModeW* are VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER, *borderColor* must be a valid VkBorderColor value
- If *compareEnable* is VK_TRUE, *compareOp* must be a valid VkCompareOp value

addressModeU, *addressModeV*, and *addressModeW* must each have one of the following values:

```
typedef enum VkSamplerAddressMode {
    VK_SAMPLER_ADDRESS_MODE_REPEAT = 0,
    VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT = 1,
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE = 2,
    VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER = 3,
    VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE = 4,
} VkSamplerAddressMode;
```

These values control the behavior of sampling with coordinates outside the range [0,1] for the respective u, v, or w coordinate as defined in the Wrapping Operation section.

- VK_SAMPLER_ADDRESS_MODE_REPEAT indicates that the repeat wrap mode will be used.
- VK_SAMPLER_ADDRESS_MODE_MIRRORED_REPEAT indicates that the mirrored repeat wrap mode will be used.

- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE indicates that the clamp to edge wrap mode will be used.
- VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER indicates that the clamp to border wrap mode will be used.
- VK_SAMPLER_ADDRESS_MODE_MIRROR_CLAMP_TO_EDGE indicates that the mirror clamp to edge wrap mode will be used.

The maximum number of sampler objects which can be simultaneously created on a device is implementation-dependent and specified by the maxSamplerAllocationCount member of the VkPhysicalDeviceLimits structure. If *maxSamplerAllocationCount* is exceeded, **vkCreateSampler** will return VK_ERROR_TOO_MANY_OBJECTS.

Since VkSampler is a non-dispatchable handle type, implementations may return the same handle for sampler state vectors that are identical. In such cases, all such objects would only count once against the *maxSamplerAllocationCount* limit.

To destroy a sampler, call:

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

device is the device to be used to destroy the sampler. *sampler* is the handle of the sampler to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *sampler* is not VK_NULL_HANDLE, *sampler* must be a valid VkSampler handle
- If *pAllocator* is not NULL, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *sampler* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *sampler* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to *sampler* must have completed execution
- If VkAllocationCallbacks were provided when *sampler* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *sampler* was created, *pAllocator* must be NULL

Host Synchronization

- Host access to *`sampler`* must be externally synchronized

Chapter 13

Resource Descriptors

Shaders access buffer and image resources by using special shader variables which are indirectly bound to buffer and image views via the API. These variables are organized into sets, where each set of bindings is represented by a *descriptor set* object in the API and a descriptor set is bound all at once. A *descriptor* is an opaque data structure representing a shader resource such as a buffer view, image view, sampler, or combined image sampler. The content of each set is determined by its *descriptor set layout* and the sequence of sets that are referenced by a pipeline is specified in a *pipeline layout*.

Each shader can use up to `maxBoundDescriptorSets` (see Limits) descriptor sets, and each descriptor set can reference all descriptor types. Each shader resource variable is assigned a tuple of (set number, binding number, array element) that defines its location within a descriptor set layout. In GLSL, the set number and binding number are assigned via layout qualifiers, and the array element is implicitly assigned consecutively starting with index equal to zero for the first element of an array (and array element is zero for non-array variables):

GLSL example

```
// Assign set number = M, binding number = N, array element = 0
layout (set=m, binding=n) uniform sampler2D variableName;

// Assign set number = M, binding number = N for all array elements, and
// array element = i for the i'th member of the array.
layout (set=m, binding=n) uniform sampler2D variableNameArray[L];
```

SPIR-V example

```
// Assign set number = M, binding number = N, array element = 0
               ...
          %1 = OpExtInstImport "GLSL.std.450"
               ...
               OpName %10 "variableName"
               OpDecorate %10 DescriptorSet m
               OpDecorate %10 Binding n
          %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
               ...
```

```
// Assign set number = M, binding number = N for all array elements, and
// array element = i for the i'th member of the array.
               ...
          %1 = OpExtInstImport "GLSL.std.450"
               ...
               OpName %13 "variableNameArray"
               OpDecorate %13 DescriptorSet m
               OpDecorate %13 Binding n
          %2 = OpTypeVoid
          %3 = OpTypeFunction %2
          %6 = OpTypeFloat 32
          %7 = OpTypeImage %6 2D 0 0 0 1 Unknown
          %8 = OpTypeSampledImage %7
          %9 = OpTypeInt 32 0
         %10 = OpConstant %9 L
         %11 = OpTypeArray %8 %10
         %12 = OpTypePointer UniformConstant %11
         %13 = OpVariable %12 UniformConstant
               ...
```

13.1 Descriptor Types

The following sections outline the various descriptor types supported by Vulkan. Each section defines a descriptor type, and each descriptor type has a manifestation in the shading language and SPIR-V as well as in descriptor sets. There is mostly a one-to-one correspondence between descriptor types and classes of opaque types in the shading language, where the opaque types in the shading language must refer to a descriptor in the pipeline layout of the corresponding descriptor type. But there is an exception to this rule as described in Combined Image Sampler.

13.1.1 Storage Image

A *storage image* (VK_DESCRIPTOR_TYPE_STORAGE_IMAGE) is a descriptor type that is used for load, store, and atomic operations on image memory from within shaders bound to pipelines.

Loads from storage images do not use samplers and are unfiltered and do not support coordinate wrapping or clamping. Loads are supported in all shader stages for image formats which report support for the VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT feature bit via `vkGetPhysicalDeviceFormatProperties`.

Stores to storage images are supported in compute shaders for image formats which report support for the VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT feature.

Storage images also support atomic operations in compute shaders for image formats which report support for the VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT feature.

Load and store operations on storage images can only be done on images in VK_IMAGE_LAYOUT_GENERAL layout.

When the fragmentStoresAndAtomics feature is enabled, stores and atomic operations are also supported for storage images in fragment shaders with the same set of image formats as supported in compute shaders. When the vertexPipelineStoresAndAtomics feature is enabled, stores and atomic operations are also supported in vertex, tessellation, and geometry shaders with the same set of image formats as supported in compute shaders.

Storage image declarations must specify the image format in the shader if the variable is used for atomic operations.

If the shaderStorageImageReadWithoutFormat feature is not enabled, storage image declarations must specify the image format in the shader if the variable is used for load operations.

If the shaderStorageImageWriteWithoutFormat feature is not enabled, storage image declarations must specify the image format in the shader if the variable is used for store operations.

Storage images are declared in GLSL shader source using uniform "image" variables of the appropriate dimensionality as well as a format layout qualifier (if necessary):

GLSL example

```
layout (set=m, binding=n, r32f) uniform image2D myStorageImage;
```

SPIR-V example

```
          ...
     %1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %9 "myStorageImage"
          OpDecorate %9 DescriptorSet m
          OpDecorate %9 Binding n
     %2 = OpTypeVoid
     %3 = OpTypeFunction %2
     %6 = OpTypeFloat 32
     %7 = OpTypeImage %6 2D 0 0 0 2 R32f
     %8 = OpTypePointer UniformConstant %7
     %9 = OpVariable %8 UniformConstant
          ...
```

13.1.2 Sampler

A *sampler* (VK_DESCRIPTOR_TYPE_SAMPLER) represents a set of parameters which control address calculations, filtering behavior, and other properties, that can be used to perform filtered loads from *sampled images* (see Sampled Image).

Samplers are declared in GLSL shader source using uniform "sampler" variables, where the sampler type has no associated texture dimensionality:

GLSL Example

```
layout (set=m, binding=n) uniform sampler mySampler;
```

SPIR-V Example

```
          ...
     %1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %8 "mySampler"
          OpDecorate %8 DescriptorSet m
          OpDecorate %8 Binding n
     %2 = OpTypeVoid
     %3 = OpTypeFunction %2
     %6 = OpTypeSampler
     %7 = OpTypePointer UniformConstant %6
     %8 = OpVariable %7 UniformConstant
          ...
```

13.1.3 Sampled Image

A *sampled image* (VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE) can be used (usually in conjunction with a sampler) to retrieve sampled image data. Shaders use a sampled image handle and a sampler handle to sample data, where the image handle generally defines the shape and format of the memory and the sampler generally defines how coordinate addressing is performed. The same sampler can be used to sample from multiple images, and it is possible to sample from the same sampled image with multiple samplers, each containing a different set of sampling parameters.

Sampled images are declared in GLSL shader source using uniform "texture" variables of the appropriate dimensionality:

GLSL example

```
layout (set=m, binding=n) uniform texture2D mySampledImage;
```

SPIR-V example

```
          ...
          %1 = OpExtInstImport "GLSL.std.450"
               ...
               OpName %9 "mySampledImage"
               OpDecorate %9 DescriptorSet m
               OpDecorate %9 Binding n
          %2 = OpTypeVoid
          %3 = OpTypeFunction %2
          %6 = OpTypeFloat 32
          %7 = OpTypeImage %6 2D 0 0 0 1 Unknown
          %8 = OpTypePointer UniformConstant %7
          %9 = OpVariable %8 UniformConstant
               ...
```

13.1.4 Combined Image Sampler

A *combined image sampler* (VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER) represents a sampled image along with a set of sampling parameters. It is logically considered a sampled image and a sampler bound together.

Note
On some implementations, it may be more efficient to sample from an image using a combination of sampler and sampled image that are stored together in the descriptor set in a combined descriptor.

Combined image samplers are declared in GLSL shader source using uniform "sampler" variables of the appropriate dimensionality:

GLSL example

```
layout (set=m, binding=n) uniform sampler2D myCombinedImageSampler;
```

SPIR-V example

```
          ...
    %1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %10 "myCombinedImageSampler"
          OpDecorate %10 DescriptorSet m
          OpDecorate %10 Binding n
    %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
          ...
```

VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER descriptor set entries can also be accessed via separate sampler and sampled image shader variables. Such variables refer exclusively to the corresponding half of the descriptor, and can be combined in the shader with samplers or sampled images that can come from the same descriptor or from other combined or separate descriptor types. There are no additional restrictions on how a separate sampler or sampled image variable is used due to it originating from a combined descriptor.

13.1.5 Uniform Texel Buffer

A *uniform texel buffer* (VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER) represents a tightly packed array of homogeneous formatted data that is stored in a buffer and is made accessible to shaders. Uniform texel buffers are read-only.

Uniform texel buffers are declared in GLSL shader source using uniform "samplerBuffer" variables:

GLSL example

```
layout (set=m, binding=n) uniform samplerBuffer myUniformTexelBuffer;
```

SPIR-V example

```
          ...
    %1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %10 "myUniformTexelBuffer"
          OpDecorate %10 DescriptorSet m
          OpDecorate %10 Binding n
    %2 = OpTypeVoid
    %3 = OpTypeFunction %2
    %6 = OpTypeFloat 32
    %7 = OpTypeImage %6 Buffer 0 0 0 1 Unknown
    %8 = OpTypeSampledImage %7
    %9 = OpTypePointer UniformConstant %8
   %10 = OpVariable %9 UniformConstant
          ...
```

13.1.6 Storage Texel Buffer

A *storage texel buffer* (VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER) represents a tightly packed array of homogeneous formatted data that is stored in a buffer and is made accessible to shaders. Storage texel buffers differ from uniform texel buffers in that they support stores and atomic operations in shaders, may support a different maximum length, and may have different performance characteristics.

Storage texel buffers are declared in GLSL shader source using uniform "imageBuffer" variables:

GLSL example

```
layout (set=m, binding=n, r32f) uniform imageBuffer myStorageTexelBuffer;
```

SPIR-V example

```
          ...
     %1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %9 "myStorageTexelBuffer"
          OpDecorate %9 DescriptorSet m
          OpDecorate %9 Binding n
     %2 = OpTypeVoid
     %3 = OpTypeFunction %2
     %6 = OpTypeFloat 32
     %7 = OpTypeImage %6 Buffer 0 0 0 2 R32f
     %8 = OpTypePointer UniformConstant %7
     %9 = OpVariable %8 UniformConstant
          ...
```

13.1.7 Uniform Buffer

A *uniform buffer* (VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER) is a region of structured storage that is made accessible for read-only access to shaders. It is typically used to store medium sized arrays of constants such as shader parameters, matrices and other related data.

Uniform buffers are declared in GLSL shader source using the uniform storage qualifier and block syntax:

GLSL example

```
layout (set=m, binding=n) uniform myUniformBuffer
{
    vec4 myElement[32];
};
```

SPIR-V example

```
          ...
     %1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %11 "myUniformBuffer"
          OpMemberName %11 0 "myElement"
          OpName %13 ""
          OpDecorate %10 ArrayStride 16
          OpMemberDecorate %11 0 Offset 0
          OpDecorate %11 Block
          OpDecorate %13 DescriptorSet m
          OpDecorate %13 Binding n
     %2 = OpTypeVoid
     %3 = OpTypeFunction %2
     %6 = OpTypeFloat 32
     %7 = OpTypeVector %6 4
```

```
          %8 = OpTypeInt 32 0
          %9 = OpConstant %8 32
         %10 = OpTypeArray %7 %9
         %11 = OpTypeStruct %10
         %12 = OpTypePointer Uniform %11
         %13 = OpVariable %12 Uniform
               ...
```

13.1.8 Storage Buffer

A *storage buffer* (VK_DESCRIPTOR_TYPE_STORAGE_BUFFER) is a region of structured storage that supports both read and write access for shaders. In addition to general read and write operations, some members of storage buffers can be used as the target of atomic operations. In general, atomic operations are only supported on members that have unsigned integer formats.

Storage buffers are declared in GLSL shader source using buffer storage qualifier and block syntax:

GLSL example

```
layout (set=m, binding=n) buffer myStorageBuffer
{
    vec4 myElement[];
};
```

SPIR-V example

```
               ...
          %1 = OpExtInstImport "GLSL.std.450"
               ...
               OpName %9 "myStorageBuffer"
               OpMemberName %9 0 "myElement"
               OpName %11 ""
               OpDecorate %8 ArrayStride 16
               OpMemberDecorate %9 0 Offset 0
               OpDecorate %9 BufferBlock
               OpDecorate %11 DescriptorSet m
               OpDecorate %11 Binding n
          %2 = OpTypeVoid
          %3 = OpTypeFunction %2
          %6 = OpTypeFloat 32
          %7 = OpTypeVector %6 4
          %8 = OpTypeRuntimeArray %7
          %9 = OpTypeStruct %8
         %10 = OpTypePointer Uniform %9
         %11 = OpVariable %10 Uniform
               ...
```

13.1.9 Dynamic Uniform Buffer

A *dynamic uniform buffer* (VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC) differs from a uniform buffer only in how its address and length are specified. Uniform buffers bind a buffer address and length that is specified in the descriptor set update by a buffer handle, offset and range (see Descriptor Set Updates). With dynamic uniform buffers the buffer handle, offset and range specified in the descriptor set define the base address and length.

The dynamic offset which is relative to this base address is taken from the *pDynamicOffsets* parameter to `vkCmdBindDescriptorSets` (see Descriptor Set Binding). The address used for a dynamic uniform buffer is the sum of the buffer base address and the relative offset. The length is unmodified and remains the range as specified in the descriptor update. The shader syntax is identical for uniform buffers and dynamic uniform buffers.

13.1.10 Dynamic Storage Buffer

A *dynamic storage buffer* (VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC) differs from a storage buffer only in how its address and length are specified. The difference is identical to the difference between uniform buffers and dynamic uniform buffers (see Dynamic Uniform Buffer). The shader syntax is identical for storage buffers and dynamic storage buffers.

13.1.11 Input Attachment

An *input attachment* (VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT) is an image view that can be used for pixel local load operations from within fragment shaders bound to pipelines. Loads from input attachments are unfiltered. All image formats that are supported for color attachments (VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT) or depth/stencil attachments (VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT) for a given image tiling mode are also supported for input attachments.

In the shader, input attachments must be referenced both by their attachment index as well as via descriptor set and binding numbers.

GLSL example

```
layout (input_attachment_index=i, set=m, binding=n) uniform subpass  ↩
    myInputAttachment;
```

SPIR-V example

```
          ...
     %1 = OpExtInstImport "GLSL.std.450"
          ...
          OpName %9 "myInputAttachment"
          OpDecorate %9 DescriptorSet m
          OpDecorate %9 Binding n
     %2 = OpTypeVoid
     %3 = OpTypeFunction %2
     %6 = OpTypeFloat 32
     %7 = OpTypeImage %6 SubpassData 0 0 0 2 Unknown
     %8 = OpTypePointer UniformConstant %7
     %9 = OpVariable %8 UniformConstant
          ...
```

13.2 Descriptor Sets

Descriptors are grouped together into descriptor set objects. A descriptor set object is an opaque object that contains storage for a set of descriptors, where the types and number of descriptors is defined by a descriptor set layout. The layout object may be used to define the association of each descriptor binding with memory or other hardware resources. The layout is used both for determining the resources that need to be associated with the descriptor set, and determining the interface between shader stages and shader resources.

13.2.1 Descriptor Set Layout

A descriptor set layout object is defined by an array of zero or more descriptor bindings. Each individual descriptor binding is specified by a descriptor type, a count (array size) of the number of descriptors in the binding, a set of shader stages that can access the binding, and (if using immutable samplers) an array of sampler descriptors.

Descriptor set layouts are represented by VkDescriptorSetLayout objects which are created by calling:

```
VkResult vkCreateDescriptorSetLayout(
    VkDevice                                    device,
    const VkDescriptorSetLayoutCreateInfo*      pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorSetLayout*                      pSetLayout);
```

The *device* parameter specifies the device that the descriptor set layout will be created on.

pAllocator controls host memory allocation as described in the Memory Allocation chapter.

The created descriptor set layout's handle will be returned in *pSetLayout*.

Valid Usage

- *device* must be a valid VkDevice handle
- *pCreateInfo* must be a pointer to a valid VkDescriptorSetLayoutCreateInfo structure
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pSetLayout* must be a pointer to a VkDescriptorSetLayout handle

Information about the descriptor set layout is passed in an instance of the VkDescriptorSetLayoutCreateInfo structure:

```
typedef struct VkDescriptorSetLayoutCreateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkDescriptorSetLayoutCreateFlags            flags;
    uint32_t                                    bindingCount;
    const VkDescriptorSetLayoutBinding*         pBindings;
} VkDescriptorSetLayoutCreateInfo;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *bindingCount* is the number of elements in *pBindings*.
- *pBindings* is a pointer to an array of `VkDescriptorSetLayoutBinding` structures.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`
- If *bindingCount* is not `0`, *pBindings* must be a pointer to an array of *bindingCount* valid VkDescriptorSetLayoutBinding structures

The definition of the VkDescriptorSetLayoutBinding structure is:

```
typedef struct VkDescriptorSetLayoutBinding {
    uint32_t                                        binding;
    VkDescriptorType                                descriptorType;
    uint32_t                                        descriptorCount;
    VkShaderStageFlags                              stageFlags;
    const VkSampler*                                pImmutableSamplers;
} VkDescriptorSetLayoutBinding;
```

- *binding* is the binding number of this entry and corresponds to a resource of the same binding number in the shader stages.
- *descriptorType* is an `VkDescriptorType` specifying which type of resource descriptors are used for this binding.
- *descriptorCount* is the number of descriptors contained in the binding, accessed in a shader as an array. If *descriptorCount* is zero this binding entry is reserved and the resource must not be accessed from any stage via this binding within any pipeline using the set layout.
- *stageFlags* member is a bitfield of `VkShaderStageFlagBits` specifying which pipeline shader stages can access a resource for this binding. VK_SHADER_STAGE_ALL is a shorthand specifying that all defined shader stages, including any additional stages defined by extensions, can access the resource.

  If a shader stage is not included in *stageFlags*, then a resource must not be accessed from that stage via this binding within any pipeline using the set layout. There are no limitations on what combinations of stages can be used by a descriptor binding, and in particular a binding can be used by both graphics stages and the compute stage.
- *pImmutableSamplers* affects initialization of samplers. If *descriptorType* specifies a VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER type descriptor, then *pImmutableSamplers* can be used to initialize a set of *immutable samplers*. Immutable samplers are permanently bound into the set layout; later binding a sampler into an immutable sampler slot in a descriptor set is not allowed. If *pImmutableSamplers* is not `NULL`, then it is considered to be a pointer to an array of sampler handles that will be consumed by the set layout and used for the corresponding binding. If *pImmutableSamplers* is `NULL`, then the sampler slots are dynamic and sampler handles must be bound into descriptor sets using this layout. If *descriptorType* is not one of these descriptor types, then *pImmutableSamplers* is ignored.

The above layout definition allows the descriptor bindings to be specified sparsely such that not all binding numbers between 0 and the maximum binding number need to be specified in the *pBindings* array. However, all binding numbers between 0 and the maximum binding number may consume memory in the descriptor set layout even if not all descriptor bindings are used, though it should not consume additional memory from the descriptor pool.

Note
The maximum binding number specified should be as compact as possible to avoid wasted memory.

Valid Usage

- `descriptorType` must be a valid `VkDescriptorType` value
- If `descriptorType` is VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and `descriptorCount` is not `0` and `pImmutableSamplers` is not `NULL`, `pImmutableSamplers` must be a pointer to an array of `descriptorCount` valid VkSampler handles
- If `descriptorCount` is not `0`, `stageFlags` must be a valid combination of `VkShaderStageFlagBits` values

The following examples show a shader snippet using two descriptor sets, and application code that creates corresponding descriptor set layouts.

GLSL example

```
//
// binding to a single sampled image descriptor in set 0
//
layout (set=0, binding=0) uniform texture2D mySampledImage;

//
// binding to an array of sampled image descriptors in set 0
//
layout (set=0, binding=1) uniform texture2D myArrayOfSampledImages[12];

//
// binding to a single uniform buffer descriptor in set 1
//
layout (set=1, binding=0) uniform myUniformBuffer
{
    vec4 myElement[32];
};
```

SPIR-V example

```
               ...
          %1 = OpExtInstImport "GLSL.std.450"
               ...
               OpName %9 "mySampledImage"
               OpName %14 "myArrayOfSampledImages"
               OpName %18 "myUniformBuffer"
               OpMemberName %18 0 "myElement"
               OpName %20 ""
               OpDecorate %9 DescriptorSet 0
               OpDecorate %9 Binding 0
               OpDecorate %14 DescriptorSet 0
               OpDecorate %14 Binding 1
               OpDecorate %17 ArrayStride 16
               OpMemberDecorate %18 0 Offset 0
               OpDecorate %18 Block
               OpDecorate %20 DescriptorSet 1
               OpDecorate %20 Binding 0
          %2 = OpTypeVoid
          %3 = OpTypeFunction %2
          %6 = OpTypeFloat 32
          %7 = OpTypeImage %6 2D 0 0 0 1 Unknown
          %8 = OpTypePointer UniformConstant %7
          %9 = OpVariable %8 UniformConstant
         %10 = OpTypeInt 32 0
         %11 = OpConstant %10 12
         %12 = OpTypeArray %7 %11
         %13 = OpTypePointer UniformConstant %12
         %14 = OpVariable %13 UniformConstant
         %15 = OpTypeVector %6 4
         %16 = OpConstant %10 32
         %17 = OpTypeArray %15 %16
         %18 = OpTypeStruct %17
         %19 = OpTypePointer Uniform %18
         %20 = OpVariable %19 Uniform
               ...
```

API example

```
VkResult myResult;

const VkDescriptorSetLayoutBinding myDescriptorSetLayoutBinding[] =
{
    // binding to a single image descriptor
    {
        0,                                      // binding
        VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,       // descriptorType
        1,                                      // descriptorCount
        VK_SHADER_STAGE_FRAGMENT_BIT,           // stageFlags
        NULL                                    // pImmutableSamplers
    },

    // binding to an array of image descriptors
    {
        1,                                      // binding
        VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,       // descriptorType
        12,                                     // descriptorCount
        VK_SHADER_STAGE_FRAGMENT_BIT,           // stageFlags
        NULL                                    // pImmutableSamplers
    },

    // binding to a single uniform buffer descriptor
    {
        0,                                      // binding
        VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,      // descriptorType
        1,                                      // descriptorCount
        VK_SHADER_STAGE_FRAGMENT_BIT,           // stageFlags
        NULL                                    // pImmutableSamplers
    }
};

const VkDescriptorSetLayoutCreateInfo myDescriptorSetLayoutCreateInfo[] =
{
    // Create info for first descriptor set with two descriptor bindings
    {
        VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,    // sType
        NULL,                                                   // pNext
        0,                                                      // flags
        2,                                                      // bindingCount
        &myDescriptorSetLayoutBinding[0]                        // pBindings
    },

    // Create info for second descriptor set with one descriptor binding
    {
        VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO,    // sType
        NULL,                                                   // pNext
        0,                                                      // flags
        1,                                                      // bindingCount
        &myDescriptorSetLayoutBinding[2]                        // pBindings
    }
};

VkDescriptorSetLayout myDescriptorSetLayout[2];

//
// Create first descriptor set layout
//
myResult = vkCreateDescriptorSetLayout(
    myDevice,
    &myDescriptorSetLayoutCreateInfo[0],
    &myDescriptorSetLayout[0]);

//
// Create second descriptor set layout
//
myResult = vkCreateDescriptorSetLayout(
    myDevice,
    &myDescriptorSetLayoutCreateInfo[1],
    &myDescriptorSetLayout[1]);
```

To destroy a descriptor set layout, call:

```
void vkDestroyDescriptorSetLayout(
    VkDevice                                    device,
    VkDescriptorSetLayout                       descriptorSetLayout,
    const VkAllocationCallbacks*                pAllocator);
```

device is the device to be used to destroy the descriptor set layout. *descriptorSetLayout* is the handle of the descriptor set layout to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *descriptorSetLayout* is not VK_NULL_HANDLE, *descriptorSetLayout* must be a valid VkDescriptorSetLayout handle
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *descriptorSetLayout* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *descriptorSetLayout* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- If VkAllocationCallbacks were provided when *descriptorSetLayout* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *descriptorSetLayout* was created, *pAllocator* must be `NULL`

Host Synchronization

- Host access to *descriptorSetLayout* must be externally synchronized

13.2.2 Pipeline Layouts

Access to descriptor sets from a pipeline is accomplished through a *pipeline layout*. Zero or more descriptor set layouts and zero or more push constant ranges are combined to form a pipeline layout object which describes the complete set of resources that can be accessed by a pipeline. The pipeline layout represents a sequence of descriptor sets with each having a specific layout. This sequence of layouts is used to determine the interface between shader stages and shader resources. Each pipeline is created using a pipeline layout.

A pipeline layout is created by calling:

```
VkResult vkCreatePipelineLayout(
    VkDevice                                    device,
    const VkPipelineLayoutCreateInfo*           pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkPipelineLayout*                           pPipelineLayout);
```

The device specified in *device* is used to create the pipeline layout, and upon success, a handle to the new layout object is placed in the variable pointed to by *pPipelineLayout*. The `VkPipelineLayoutCreateInfo` structure pointed to by *pCreateInfo* is used to construct the pipeline layout. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- `device` must be a valid VkDevice handle
- `pCreateInfo` must be a pointer to a valid VkPipelineLayoutCreateInfo structure
- If `pAllocator` is not `NULL`, `pAllocator` must be a pointer to a valid VkAllocationCallbacks structure
- `pPipelineLayout` must be a pointer to a VkPipelineLayout handle

The definition of the `VkPipelineLayoutCreateInfo` structure is:

```
typedef struct VkPipelineLayoutCreateInfo {
    VkStructureType                                sType;
    const void*                                    pNext;
    VkPipelineLayoutCreateFlags                    flags;
    uint32_t                                       setLayoutCount;
    const VkDescriptorSetLayout*                   pSetLayouts;
    uint32_t                                       pushConstantRangeCount;
    const VkPushConstantRange*                     pPushConstantRanges;
} VkPipelineLayoutCreateInfo;
```

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `setLayoutCount` is the number of descriptor sets included in the pipeline layout.
- `pSetLayouts` is a pointer to an array of VkDescriptorSetLayout objects.
- `pushConstantRangeCount` is the number of push constant ranges included in the pipeline layout.
- `pPushConstantRanges` is a pointer to an array of VkPushConstantRange structures defining a set of push constant ranges for use in a single pipeline layout. In addition to descriptor set layouts, a pipeline layout also describes how many push constants can be accessed by each stage of the pipeline.

Note
Push constants represent a high speed path to modify constant data in pipelines that is expected to outperform memory-backed resource updates.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`
- If *setLayoutCount* is not `0`, *pSetLayouts* must be a pointer to an array of *setLayoutCount* valid VkDescriptorSetLayout handles
- If *pushConstantRangeCount* is not `0`, *pPushConstantRanges* must be a pointer to an array of *pushConstantRangeCount* valid VkPushConstantRange structures
- *setLayoutCount* must be less than or equal to VkPhysicalDeviceLimits::*maxBoundDescriptorSets*
- The total number of descriptors of the type VK_DESCRIPTOR_TYPE_SAMPLER and VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, referenced across all elements of *pSetLayouts* for any given shader stage, must be less than or equal to the value of VkPhysicalDeviceLimits::*maxPerStageDescriptorSamplers*
- The total number of descriptors of the type VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER and VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, referenced across all elements of *pSetLayouts* for any given shader stage, must be less than or equal to the value of VkPhysicalDeviceLimits::*maxPerStageDescriptorUniformBuffers*
- The total number of descriptors of the type VK_DESCRIPTOR_TYPE_STORAGE_BUFFER and VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, referenced across all elements of *pSetLayouts* for any given shader stage, must be less than or equal to the value of VkPhysicalDeviceLimits::*maxPerStageDescriptorStorageBuffers*
- The total number of descriptors of the type VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, and VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, referenced across all elements of *pSetLayouts* for any given shader stage, must be less than or equal to the value of VkPhysicalDeviceLimits::*maxPerStageDescriptorSampledImages*
- The total number of descriptors of the type VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, and VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, referenced across all elements of *pSetLayouts* for any given shader stage, must be less than or equal to the value of VkPhysicalDeviceLimits::*maxPerStageDescriptorStorageImages*

The definition of VkPushConstantRange is:

```
typedef struct VkPushConstantRange {
    VkShaderStageFlags                        stageFlags;
    uint32_t                                  offset;
    uint32_t                                  size;
} VkPushConstantRange;
```

- *stageFlags* is a set of stage flags describing the shader stages that will access a range of push constants. If a particular stage is not included in the range, then accessing members of that range of push constants from the corresponding shader stage will result in undefined data being read.

- `offset` and `size` are the start offset and size, respectively, consumed by the range. Both `offset` and `size` are in units of bytes and must be a multiple of 4. The layout of the push constant variables is specified in the shader.

Valid Usage

- `stageFlags` must be a valid combination of `VkShaderStageFlagBits` values
- `stageFlags` must not be `0`
- The sum of `offset` and `size` must be less than or equal to the value of VkPhysicalDeviceLimits::`maxPushConstantsSize`
- The value of `size` must be greater than `0`
- The value of `size` must be a multiple of `4`

Once created, pipeline layouts are used as part of pipeline creation (see Pipelines), as part of binding descriptor sets (see Descriptor Set Binding), and as part of setting push constants (see Push Constant Updates). Pipeline creation accepts a pipeline layout as input, and the layout may be used to map (set, binding, arrayElement) tuples to hardware resources or memory locations within a descriptor set. The assignment of hardware resources depends only on the bindings defined in the descriptor sets that comprise the pipeline layout, and not on any shader source.

All resource variables statically used in all shaders in a pipeline must be declared with a (set,binding,arrayElement) that exists in the corresponding descriptor set layout and is of an appropriate descriptor type and includes the set of shader stages it is used by in `stageFlags`. The pipeline layout can include entries that are not used by a particular pipeline, or that are dead-code eliminated from any of the shaders. The pipeline layout allows the application to provide a consistent set of bindings across multiple pipeline compiles, which enables those pipelines to be compiled in a way that the implementation may cheaply switch pipelines without reprogramming the bindings.

Similarly, the push constant block declared in each shader (if present) must only place variables at offsets that are each included in a push constant range with `stageFlags` including the bit corresponding to the shader stage that uses it. The pipeline layout can include ranges or portions of ranges that are not used by a particular pipeline, or for which the variables have been dead-code eliminated from any of the shaders.

There is a limit on the total number of resources of each type that can be referenced by a pipeline layout as shown in Pipeline Layout Resource Limits. The "Total Resources Available" column gives the limit on the number of each type of resource that can be referenced from all descriptor sets in the pipeline layout. Some resource types count against multiple limits. Additionally, there are limits on the total number of each type of resource that can be used in any pipeline stage as described in Shader Resource Limits.

Table 13.1: Pipeline Layout Resource Limits

Total Resources Available	**Resource Types**
maxDescriptorSetSamplers	sampler
	combined image sampler
maxDescriptorSetSampledImages	sampled image
	combined image sampler
	uniform texel buffer
maxDescriptorSetStorageImages	storage image

Table 13.1: (continued)

Total Resources Available	**Resource Types**
	storage texel buffer
maxDescriptorSetUniformBuffers	uniform buffer
	uniform buffer dynamic
maxDescriptorSetUniformBuffersDynamic	uniform buffer dynamic
maxDescriptorSetStorageBuffers	storage buffer
	storage buffer dynamic
maxDescriptorSetStorageBuffersDynamic	storage buffer dynamic
maxDescriptorSetInputAttachments	input attachment

To destroy a pipeline layout, call:

```
void vkDestroyPipelineLayout(
    VkDevice                                    device,
    VkPipelineLayout                            pipelineLayout,
    const VkAllocationCallbacks*                pAllocator);
```

device is the device to be used to destroy the pipeline layout. *pipelineLayout* is the handle of the pipeline layout to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *pipelineLayout* is not VK_NULL_HANDLE, *pipelineLayout* must be a valid VkPipelineLayout handle
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *pipelineLayout* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *pipelineLayout* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- If VkAllocationCallbacks were provided when *pipelineLayout* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *pipelineLayout* was created, *pAllocator* must be `NULL`

Host Synchronization

- Host access to *pipelineLayout* must be externally synchronized

13.2.2.1 Pipeline Layout Compatibility

Two pipeline layouts are defined to be "compatible for push constants" if they were created with identical push constant ranges. Two pipeline layouts are defined to be "compatible for set N" if they were created with matching (the same, or identically defined) descriptor set layouts for sets zero through N, and if they were created with identical push constant ranges.

When binding a descriptor set (see Descriptor Set Binding) to set number N, if the previously bound descriptor sets for sets zero through N-1 were all bound using compatible pipeline layouts, then performing this binding does not disturb any of the lower numbered sets. If, additionally, the previous bound descriptor set for set N was bound using a pipeline layout compatible for set N, then the bindings in sets numbered greater than N are also not disturbed.

Similarly, when binding a pipeline, the pipeline can correctly access any previously bound descriptor sets which were bound with compatible pipeline layouts, as long as all lower numbered sets were also bound with compatible layouts.

Layout compatibility means that descriptor sets can be bound without reference to a specific pipeline used to create them, and without having bound a particular pipeline first. It also means that descriptor sets can remain valid across a pipeline change, and the same resources will be available to the newly bound pipeline.

Implementor's Note

A consequence of layout compatibility is that when the implementation compiles a pipeline layout and assigns hardware units to resources, the mechanism to assign hardware units for set N should only be a function of sets [0..N].

Note
Place the least frequently changing descriptor sets near the start of the pipeline layout, and place the descriptor sets representing the most frequently changing resources near the end. When pipelines are switched, only the descriptor set bindings that have been invalidated will need to be updated and the remainder of the descriptor set bindings will remain in place.

The maximum number of descriptor sets that can be bound to a pipeline layout is queried from physical device properties (see *maxBoundDescriptorSets* in Limits).

API example

```
const VkDescriptorSetLayout layouts = { layout1, layout2 };

const VkPushConstantRange ranges[] =
{
    {
        VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,     // stageFlags
        0,                                      // offset
        4                                       // size
    },

    {
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,  // stageFlags
        4,                                      // offset
        4                                       // size
    },
};

const VkPipelineLayoutCreateInfo createInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO,  // sType
    NULL,                                           // pNext
    2,                                              // setLayoutCount
    layouts,                                        // pSetLayouts
    2,                                              // pushConstantRangeCount
    ranges                                          // pPushConstantRanges
};

VkPipelineLayout myPipelineLayout;
myResult = vkCreatePipelineLayout(
    myDevice,
    &createInfo,
    &myPipelineLayout);
```

13.2.3 Allocation of Descriptor Sets

Descriptor sets are allocated from *descriptor pool* objects. A descriptor pool maintains a pool of descriptors, from which sets are allocated. Descriptor pools are externally synchronized, meaning that the application must not allocate and/or free descriptor sets from the same pool in multiple threads simultaneously.

Descriptor pools are created by calling:

```
VkResult vkCreateDescriptorPool(
    VkDevice                                    device,
    const VkDescriptorPoolCreateInfo*           pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkDescriptorPool*                           pDescriptorPool);
```

device is the device that *pDescriptorPool* will be created on.

pAllocator controls host memory allocation as described in the Memory Allocation chapter.

The created descriptor pool is returned in *pDescriptorPool*.

Valid Usage

- *device* must be a valid VkDevice handle

- *pCreateInfo* must be a pointer to a valid VkDescriptorPoolCreateInfo structure
- If *pAllocator* is not NULL, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- *pDescriptorPool* must be a pointer to a VkDescriptorPool handle

Additional information about the pool is passed in an instance of the VkDescriptorPoolCreateInfo structure:

```
typedef struct VkDescriptorPoolCreateInfo {
    VkStructureType                              sType;
    const void*                                  pNext;
    VkDescriptorPoolCreateFlags                  flags;
    uint32_t                                     maxSets;
    uint32_t                                     poolSizeCount;
    const VkDescriptorPoolSize*                  pPoolSizes;
} VkDescriptorPoolCreateInfo;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to an extension-specific structure.
- *flags* specifies certain supported operations on the pool, with possible values defined below.
- *maxSets* is the maximum number of descriptor sets that can be allocated from the pool.
- *poolSizeCount* is the number of elements in *pPoolSizes*.
- *pPoolSizes* is a pointer to an array of VkDescriptorPoolSize structures, each containing a descriptor type and number of descriptors of that type to be allocated in the pool.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO
- *pNext* must be NULL
- *flags* must be a valid combination of VkDescriptorPoolCreateFlagBits values
- *pPoolSizes* must be a pointer to an array of *poolSizeCount* valid VkDescriptorPoolSize structures
- The value of *poolSizeCount* must be greater than 0
- The value of *maxSets* must be greater than 0

If multiple VkDescriptorPoolSize structures appear in the *pPoolSizes* array then the pool will be created with enough storage for the total number of descriptors of each type.

Fragmentation of a descriptor pool is possible and may lead to descriptor set allocation failures. A failure due to fragmentation is defined as failing a descriptor set allocation despite the sum of all outstanding descriptor set allocations from the pool plus the requested allocation requiring no more than the total number of descriptors requested at pool creation. Implementations provide certain guarantees of when fragmentation must not cause allocation failure, as described below.

If a descriptor pool has not had any descriptor sets freed since it was created or most recently reset then fragmentation must not cause an allocation failure (note that this is always the case for a pool created without the VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT bit set). Additionally, if all sets allocated from the pool since it was created or most recently reset use the same number of descriptors (of each type) and the requested allocation also uses that same number of descriptors (of each type), then fragmentation must not cause an allocation failure.

If an allocation failure occurs due to fragmentation, an application can create an additional descriptor pool to perform further descriptor set allocations.

The *`flags`* member of VkDescriptorPoolCreateInfo can include the following values:

```
typedef enum VkDescriptorPoolCreateFlagBits {
    VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT = 0x00000001,
} VkDescriptorPoolCreateFlagBits;
```

If *`flags`* includes VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT, then descriptor sets can return their individual allocations to the pool, i.e. all of **`vkAllocateDescriptorSets`**, **`vkFreeDescriptorSets`**, and **`vkResetDescriptorPool`** are allowed. Otherwise, descriptor sets allocated from the pool must not be individually freed back to the pool, i.e. only **`vkAllocateDescriptorSets`** and **`vkResetDescriptorPool`** are allowed.

The definition of the VkDescriptorPoolSize structure is:

```
typedef struct VkDescriptorPoolSize {
    VkDescriptorType                               type;
    uint32_t                                       descriptorCount;
} VkDescriptorPoolSize;
```

- *`type`* is the type of descriptor.
- *`descriptorCount`* is the number of descriptors of that type to allocate.

Valid Usage

- *`type`* must be a valid `VkDescriptorType` value
- The value of *`descriptorCount`* must be greater than `0`

To destroy a descriptor pool, call:

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

`device` is the device to be used to destroy the descriptor pool. `descriptorPool` is the handle of the descriptor pool to destroy. `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- `device` must be a valid VkDevice handle
- If `descriptorPool` is not VK_NULL_HANDLE, `descriptorPool` must be a valid VkDescriptorPool handle
- If `pAllocator` is not `NULL`, `pAllocator` must be a pointer to a valid VkAllocationCallbacks structure
- If `descriptorPool` is a valid handle, it must have been created, allocated or retrieved from `device`
- Each of `device` and `descriptorPool` that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to `descriptorPool` (via any allocated descriptor sets) must have completed execution
- If VkAllocationCallbacks were provided when `descriptorPool` was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when `descriptorPool` was created, `pAllocator` must be `NULL`

Host Synchronization

- Host access to `descriptorPool` must be externally synchronized

When a pool is destroyed, all descriptor sets allocated from the pool are implicitly freed and become invalid. Descriptor sets allocated from a given pool do not need to be freed before destroying that descriptor pool.

Descriptor sets are allocated from a descriptor pool by calling:

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

The `device` parameter specifies the device that the descriptor pool is owned by. `pAllocateInfo` describes the parameters of the allocation.

Valid Usage

- *device* must be a valid VkDevice handle
- *pAllocateInfo* must be a pointer to a valid VkDescriptorSetAllocateInfo structure
- *pDescriptorSets* must be a pointer to an array of *pAllocateInfo*→count VkDescriptorSet handles

Host Synchronization

- Host access to *pAllocateInfo*→descriptorPool must be externally synchronized

The VkDescriptorSetAllocateInfo structure is defined as:

```
typedef struct VkDescriptorSetAllocateInfo {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkDescriptorPool                                descriptorPool;
    uint32_t                                        setLayoutCount;
    const VkDescriptorSetLayout*                    pSetLayouts;
} VkDescriptorSetAllocateInfo;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *descriptorPool* is the pool which the sets will be allocated from.
- *setLayoutCount* determines the number descriptor sets to be allocated from the pool.
- *pSetLayouts* is an array of descriptor set layouts, with each member specifying how the corresponding descriptor set is allocated.

The allocated descriptor sets are returned in *pDescriptorSets*.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO
- *pNext* must be `NULL`
- *descriptorPool* must be a valid VkDescriptorPool handle

- *pSetLayouts* must be a pointer to an array of *setLayoutCount* valid VkDescriptorSetLayout handles
- The value of *setLayoutCount* must be greater than 0
- Each of *descriptorPool* and the elements of *pSetLayouts* must have been created, allocated or retrieved from the same VkDevice
- The value of *setLayoutCount* must not be greater than the number of sets that are currently available for allocation in *descriptorPool*

When a descriptor set is allocated, the initial state is largely uninitialized and all entries reference undefined descriptors. However, the descriptor set can be bound in a command buffer without causing errors or exceptions. Entries must be populated before they are accessed by a pipeline, but leaving uninitialized entries that are not accessed by a pipeline will produce well-defined results. This means applications need not populate unused entries with dummy descriptors.

Allocated descriptor sets are freed by calling:

```
VkResult vkFreeDescriptorSets(
    VkDevice                                    device,
    VkDescriptorPool                            descriptorPool,
    uint32_t                                    descriptorSetCount,
    const VkDescriptorSet*                      pDescriptorSets);
```

The *device* parameter specifies the device that the *descriptorPool* is owned by.

In order to free individual descriptor sets, *descriptorPool* must have been created with the VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT flag.

The descriptor sets to be freed are specified in *pDescriptorSets*. All elements of *pDescriptorSets* must have been allocated from *descriptorPool*.

Valid Usage

- *device* must be a valid VkDevice handle
- *descriptorPool* must be a valid VkDescriptorPool handle
- The value of *descriptorSetCount* must be greater than 0
- *descriptorPool* must have been created, allocated or retrieved from *device*
- Each element of *pDescriptorSets* that is a valid handle must have been created, allocated or retrieved from *descriptorPool*
- Each of *device*, *descriptorPool* and the elements of *pDescriptorSets* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to any element of *pDesciptorSets* must have completed execution
- *pDescriptorSets* must be a pointer to an array of *descriptorSetCount* VkDescriptorSet handles, each element of which must either be a valid handle or VK_NULL_HANDLE

- *descriptorPool* must have been created with the VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT flag

Host Synchronization

- Host access to *descriptorPool* must be externally synchronized
- Host access to each member of *pDescriptorSets* must be externally synchronized

After a successful call to **vkFreeDescriptorSets**, all descriptor sets in *pDescriptorSets* are invalid.

Rather than freeing individual descriptor sets, all descriptor sets allocated from a given pool can be returned to the pool by calling:

```
VkResult vkResetDescriptorPool(
    VkDevice                                    device,
    VkDescriptorPool                            descriptorPool,
    VkDescriptorPoolResetFlags                  flags);
```

The *device* parameter specifies the device that the *descriptorPool* is owned by. *descriptorPool* is the pool to be reset. The *flags* parameter is currently unused.

Valid Usage

- *device* must be a valid VkDevice handle
- *descriptorPool* must be a valid VkDescriptorPool handle
- *flags* must be `0`
- *descriptorPool* must have been created, allocated or retrieved from *device*
- Each of *device* and *descriptorPool* must have been created, allocated or retrieved from the same VkPhysicalDevice
- All uses of *descriptorPool* (via any allocated descriptor sets) must have completed execution

Host Synchronization

- Host access to *descriptorPool* must be externally synchronized
- Host access to any VkDescriptorSet objects allocated from *descriptorPool* must be externally synchronized

Resetting a descriptor pool recycles all of the resources from all of the descriptor sets allocated from the descriptor pool back to the descriptor pool, and the descriptor sets are implicitly freed.

13.2.4 Descriptor Set Updates

Once allocated, descriptor sets are updated by passing an array of VkWriteDescriptorSet and an array of VkCopyDescriptorSet structures to the **vkUpdateDescriptorSets** command. Its prototype is:

```
void vkUpdateDescriptorSets(
    VkDevice                                    device,
    uint32_t                                    descriptorWriteCount,
    const VkWriteDescriptorSet*                 pDescriptorWrites,
    uint32_t                                    descriptorCopyCount,
    const VkCopyDescriptorSet*                  pDescriptorCopies);
```

The *device* parameter specifies the device that is to carry out the updates to the specified descriptor sets. *pDescriptorWrites* and *pDescriptorCopies* parameters are pointers to arrays of VkWriteDescriptorSet and VkCopyDescriptorSet structures. These arrays contain *writeCount* and *copyCount* elements, respectively.

Valid Usage

- *device* must be a valid VkDevice handle
- If *descriptorWriteCount* is not 0, *pDescriptorWrites* must be a pointer to an array of *descriptorWriteCount* valid VkWriteDescriptorSet structures
- If *descriptorCopyCount* is not 0, *pDescriptorCopies* must be a pointer to an array of *descriptorCopyCount* valid VkCopyDescriptorSet structures

Host Synchronization

- Host access to *pDescriptorWrites*[].dstSet must be externally synchronized
- Host access to *pDescriptorCopies*[].dstSet must be externally synchronized

Each element in the *pDescriptorWrites* array describes an operation updating the descriptor set using descriptors for resources specified in the structure.

The definition of VkWriteDescriptorSet is:

```
typedef struct VkWriteDescriptorSet {
    VkStructureType                                 sType;
    const void*                                     pNext;
    VkDescriptorSet                                 dstSet;
    uint32_t                                        dstBinding;
    uint32_t                                        dstArrayElement;
    uint32_t                                        descriptorCount;
    VkDescriptorType                                descriptorType;
    const VkDescriptorImageInfo*                    pImageInfo;
    const VkDescriptorBufferInfo*                   pBufferInfo;
    const VkBufferView*                             pTexelBufferView;
} VkWriteDescriptorSet;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *dstSet* is the destination descriptor set to update.
- *dstBinding* is the descriptor binding within that set.
- *dstArrayElement* is the starting element in that array.
- *descriptorCount* is the number of descriptors to update (the number of elements in *pImageInfo*, *pBufferInfo*, or *pTexelBufferView*).
- *descriptorType* is the type of each descriptor in *pImageInfo*, *pBufferInfo*, or *pTexelBufferView*, and must be the same type as what was specified in VkDescriptorSetLayoutBinding for *dstSet* at *dstBinding*. The type of the descriptor also controls which array the descriptors are taken from. *descriptorType* can take on values including:

```
typedef enum VkDescriptorType {
    VK_DESCRIPTOR_TYPE_SAMPLER = 0,
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER = 1,
    VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE = 2,
    VK_DESCRIPTOR_TYPE_STORAGE_IMAGE = 3,
    VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER = 4,
    VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER = 5,
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER = 6,
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER = 7,
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC = 8,
    VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC = 9,
    VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT = 10,
} VkDescriptorType;
```

- *pImageInfo* points to an array of VkDescriptorImageInfo structures or is ignored, as described below.
- *pBufferInfo* points to an array of VkDescriptorBufferInfo structures or is ignored, as described below.
- *pTexelBufferView* points to an array of VkBufferView handles or is ignored, as described below.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET
- *pNext* must be `NULL`
- *dstSet* must be a valid VkDescriptorSet handle
- *descriptorType* must be a valid `VkDescriptorType` value
- The value of *descriptorCount* must be greater than `0`
- Each of *dstSet* and the elements of *pTexelBufferView* that are valid handles must have been created, allocated or retrieved from the same VkDevice
- *dstBinding* must be a valid binding point within *dstSet*
- *descriptorType* must match the type of *dstBinding* within *dstSet*
- The sum of *dstArrayElement* and *descriptorCount* must be less than or equal to the number of array elements in the descriptor set binding specified by *dstBinding*, and all applicable consecutive bindings, as described by consecutive binding updates
- If *descriptorType* is VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE or VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, *pImageInfo* must be a pointer to an array of *descriptorCount* valid VkDescriptorImageInfo structures
- If *descriptorType* is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, *pTexelBufferView* must be a pointer to an array of *descriptorCount* valid VkBufferView handles
- If *descriptorType* is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, *pBufferInfo* must be a pointer to an array of *descriptorCount* valid VkDescriptorBufferInfo structures
- If *descriptorType* is VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and *dstSet* was not created with a layout that included immutable samplers for *dstBinding* with *descriptorType*, the *sampler* member of any given element of *pImageInfo* must be a valid VkSampler object
- If *descriptorType* is VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE or VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, the *imageView* and *imageLayout* members of any given element of *pImageInfo* must be a valid VkImageView and `VkImageLayout`, respectively
- If *descriptorType* is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, the *offset* member of any given element of *pBufferInfo* must be a multiple of the value of VkPhysicalDeviceLimits::*minUniformBufferOffsetAlignment*
- If *descriptorType* is VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, the *offset* member of any given element of *pBufferInfo* must be a multiple of the value of VkPhysicalDeviceLimits::*minStorageBufferOffsetAlignment*

- If *descriptorType* is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, the *buffer* member of any given element of *pBufferInfo* must have been created with VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT set
- If *descriptorType* is VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, the *buffer* member of any given element of *pBufferInfo* must have been created with VK_BUFFER_USAGE_STORAGE_BUFFER_BIT set
- If *descriptorType* is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, the *range* member of any given element of *pBufferInfo* must be less than or equal to the value of VkPhysicalDeviceLimits::*maxUniformBufferRange*
- If *descriptorType* is VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, the *range* member of any given element of *pBufferInfo* must be less than or equal to the value of VkPhysicalDeviceLimits::*maxStorageBufferRange*
- If *descriptorType* is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, the VkBuffer that *pTexelBufferView* references must have been created with VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT set
- If *descriptorType* is VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, the VkBuffer that *pTexelBufferView* references must have been created with VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT set
- If *descriptorType* is VK_DESCRIPTOR_TYPE_STORAGE_IMAGE or VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, the *imageView* must have been created with identity swizzle

Only one of *pImageInfo*, *pBufferInfo*, or *pTexelBufferView* members is used according to the descriptor type specified in the *descriptorType* member of the containing VkWriteDescriptorSet structure, as specified below.

If *descriptorType* is VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC, the *pBufferInfo* array will be used to update the descriptors, and other arrays will be ignored. Each entry is of type VkDescriptorBufferInfo and is defined as:

```
typedef struct VkDescriptorBufferInfo {
    VkBuffer                                      buffer;
    VkDeviceSize                                  offset;
    VkDeviceSize                                  range;
} VkDescriptorBufferInfo;
```

- *buffer* is the buffer resource.
- *offset* is the offset in bytes from the start of *buffer*. Access to buffer memory via this descriptor uses addressing that is relative to this starting offset.
- *range* is the size in bytes that is used for this descriptor update, or VK_WHOLE_SIZE to use the range from *offset* to the end of the buffer.

Valid Usage

- `buffer` must be a valid VkBuffer handle
- If `range` is not equal to VK_WHOLE_SIZE, the sum of `offset` and `range` must be less than or equal to the size of `buffer`

For VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC and VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC descriptor types, `offset` is the base offset from which the dynamic offset is applied and `range` is the static size used for all dynamic offsets.

If `descriptorType` is VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, the `pTexelBufferView` array will be used to update the descriptors, and other arrays will be ignored.

If `descriptorType` is VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, or VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT, the members in `pImageInfo` array will be used to update the descriptors, and other arrays will be ignored. `imageInfo` is of type VkDescriptorImageInfo and is defined as:

```
typedef struct VkDescriptorImageInfo {
    VkSampler                                            sampler;
    VkImageView                                          imageView;
    VkImageLayout                                        imageLayout;
} VkDescriptorImageInfo;
```

- `sampler` is a sampler handle, and is used in descriptor updates for types VK_DESCRIPTOR_TYPE_SAMPLER and VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER if the binding being updated does not use immutable samplers.
- `imageView` is an image view handle, and is used in descriptor updates for types VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT.
- `imageLayout` is the layout that the image will be in at the time this descriptor is accessed. `imageLayout` is used in descriptor updates for types VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, and VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT.

Valid Usage

- Each of `sampler` and `imageView` that are valid handles must have been created, allocated or retrieved from the same VkDevice

Members of VkDescriptorImageInfo that are not used in an update (as described above) are ignored.

If the *dstBinding* has fewer than *descriptorCount* array elements remaining starting from *dstArrayElement*, then the remainder will be used to update the subsequent binding - *dstBinding*+1 starting at array element zero. This behavior applies recursively, with the update affecting consecutive bindings as needed to update all *descriptorCount* descriptors. All consecutive bindings updated via a single VkWriteDescriptorSet structure must have identical *descriptorType* and *stageFlags*, and must all either use immutable samplers or must all not use immutable samplers.

Each element in the *pDescriptorCopies* array in a VkCopyDescriptorSet structure describing an operation copying descriptors between sets. The definition of VkCopyDescriptorSet is:

```
typedef struct VkCopyDescriptorSet {
    VkStructureType                                  sType;
    const void*                                      pNext;
    VkDescriptorSet                                  srcSet;
    uint32_t                                         srcBinding;
    uint32_t                                         srcArrayElement;
    VkDescriptorSet                                  dstSet;
    uint32_t                                         dstBinding;
    uint32_t                                         dstArrayElement;
    uint32_t                                         descriptorCount;
} VkCopyDescriptorSet;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *srcSet*, *srcBinding*, and *srcArrayElement* are the source set, binding, and array element, respectively.
- *dstSet*, *dstBinding*, and *dstArrayElement* are the destination set, binding, and array element, respectively.
- *descriptorCount* is the number of descriptors to copy from the source to destination. If *descriptorCount* is greater than the number of remaining array elements in the source or destination binding, those affect consecutive bindings in a manner similar to `VkWriteDescriptorSet` above.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_COPY_DESCRIPTOR_SET
- *pNext* must be `NULL`
- *srcSet* must be a valid VkDescriptorSet handle
- *dstSet* must be a valid VkDescriptorSet handle
- Each of *srcSet* and *dstSet* must have been created, allocated or retrieved from the same VkDevice
- *srcBinding* must be a valid binding within *srcSet*
- The sum of *srcArrayElement* and *descriptorCount* must be less than or equal to the number of array elements in the descriptor set binding specified by *srcBinding*, and all applicable consecutive bindings, as described by consecutive binding updates

- `dstBinding` must be a valid binding within `dstSet`
- The sum of `dstArrayElement` and `descriptorCount` must be less than or equal to the number of array elements in the descriptor set binding specified by `dstBinding`, and all applicable consecutive bindings, as described by consecutive binding updates

13.2.5 Descriptor Set Binding

Once descriptor sets have been allocated, one or more descriptor sets can be bound to the command buffer by calling:

```
void vkCmdBindDescriptorSets(
    VkCommandBuffer                             commandBuffer,
    VkPipelineBindPoint                         pipelineBindPoint,
    VkPipelineLayout                            layout,
    uint32_t                                    firstSet,
    uint32_t                                    descriptorSetCount,
    const VkDescriptorSet*                      pDescriptorSets,
    uint32_t                                    dynamicOffsetCount,
    const uint32_t*                             pDynamicOffsets);
```

`commandbuffer` is the command buffer that the descriptor sets will be bound to, and *`pipelineBindPoint`* is a `VkPipelineBindPoint` indicating whether the descriptors will be used by graphics pipelines or compute pipelines. There is a separate set of bind points for each of graphics and compute, so binding one does not disturb the other.

`descriptorSetCount` is the number of descriptor sets being bound and is the number of elements in the *`pDescriptorSets`* array, and *`firstSet`* is the set number of the first set to be bound. Element i of *`pDescriptorSets`* is bound to set number firstSet+i.

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`pipelineBindPoint`* must be a valid `VkPipelineBindPoint` value
- *`layout`* must be a valid VkPipelineLayout handle
- *`pDescriptorSets`* must be a pointer to an array of *`descriptorSetCount`* valid VkDescriptorSet handles
- If *`dynamicOffsetCount`* is not `0`, *`pDynamicOffsets`* must be a pointer to an array of *`dynamicOffsetCount`* uint32_t values
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support graphics or compute operations
- The value of *`descriptorSetCount`* must be greater than `0`
- Each of *`commandBuffer`*, *`layout`* and the elements of *`pDescriptorSets`* must have been created, allocated or retrieved from the same VkDevice

- Any given element of *pDescriptorSets* must have been created with a VkDescriptorSetLayout that matches the VkDescriptorSetLayout at set *n* in *layout*, where *n* is the sum of the index into *pDescriptorSets* and *firstSet*
- *dynamicOffsetCount* must be equal to the total number of dynamic descriptors in *pDescriptorSets*
- *pipelineBindPoint* must be supported by the *commandBuffer*'s parent VkCommandPool's queue family

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

vkCmdBindDescriptorSets causes the sets numbered [*firstSet*, *firstSet*+*descriptorSetCount*) to use the bindings stored in *pDescriptorSets*[0..*descriptorSetCount*-1] for subsequent rendering commands (either compute or graphics, according to the *pipelineBindPoint*). Any bindings that were previously applied via these sets are no longer valid.

Once bound, a descriptor set affects rendering of subsequent graphics or compute commands in the command buffer until a different set is bound to the same set number, or else the set is disturbed as described in Pipeline Layout Compatibility.

A compatible descriptor set must be bound for all set numbers that any shaders in a pipeline access, at the time that a draw or dispatch command is recorded to execute using that pipeline. However, if a pipeline does not reference a particular set number, then no descriptor set need be bound for that set number, even if the pipeline layout included that set.

If any of the sets being bound include dynamic uniform or storage buffers, then *pDynamicOffsets* includes one element for each array element in each dynamic descriptor type binding in each set. Values are taken from *pDynamicOffsets* in an order such that all entries for set N come before set N+1, and within a set entries are ordered by the binding numbers in the decriptor set layouts, and within a binding array elements are in order. *dynamicOffsetCount* is the total number of dynamic offsets provided, and must equal the total number of dynamic descriptors in the sets being bound.

The effective offset used for dynamic uniform and storage buffer bindings is the sum of the relative offset taken from *pDynamicOffsets*, and the base address of the buffer plus base offset in the descriptor set. The length of the dynamic uniform and storage buffer bindings is the buffer range as specified in the descriptor set.

The *layout* is a pipeline layout used to program the bindings, and each of the *pDescriptorSets* must be compatible with the pipeline layout. The pipeline layout used to program the bindings must also be compatible with the pipeline used in subsequent graphics or compute commands, as defined in the Pipeline Layout Compatibility section.

The descriptor set contents referenced by a **vkCmdBindDescriptorSets** command may be consumed during host execution of the command, or during shader execution of the resulting draws, or any time in between. Thus, the contents must not be altered (overwritten by an Update command, or freed) between when the command is recorded and when the command completes executing on the queue. The contents of *pDynamicOffsets* are consumed immediately during execution of **vkCmdBindDescriptorSets**. Once all pending uses have completed, it is legal to update and reuse a descriptor set.

13.2.6 Push Constant Updates

As described above in section Pipeline Layouts, the pipeline layout defines shader push constants which are updated via Vulkan commands rather than via writes to memory or copy commands.

Note
Push constants represent a high speed path to modify constant data in pipelines that is expected to outperform memory-backed resource updates.

The contents of the push constants are undefined at the start of a command buffer. Push constants are updated by calling:

```
void vkCmdPushConstants(
    VkCommandBuffer                             commandBuffer,
    VkPipelineLayout                            layout,
    VkShaderStageFlags                          stageFlags,
    uint32_t                                    offset,
    uint32_t                                    size,
    const void*                                 pValues);
```

commandbuffer is the command buffer in which the push constant update will be recorded. *layout* is the pipeline layout used to program the push constant updates. *stageFlags* specifies the shader stages that will use the push constants in the updated range. *offset* is the start offset of the push constant range to update and *size* is the size of the range to update. Both *offset* and *size* are in units of bytes and must be a multiple of 4. *pValues* is an array of *size* bytes containing the new push constant values.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *layout* must be a valid VkPipelineLayout handle
- *stageFlags* must be a valid combination of `VkShaderStageFlagBits` values
- *stageFlags* must not be `0`
- *pValues* must be a pointer to an array of *size* bytes
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics or compute operations
- The value of *size* must be greater than `0`
- Each of *commandBuffer* and *layout* must have been created, allocated or retrieved from the same VkDevice
- *stageFlags* must match exactly the shader stages used in *layout* for the range specified by *offset* and *size*
- *offset* must be a multiple of `4`
- *size* must be a multiple of `4`

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Chapter 14

Image Operations

14.1 Image Operations Overview

Image Operations are steps performed by SPIR-V image instructions, where those instructions which take an **`OpTypeImage`** (representing a VkImageView) or **`OpTypeImageSampler`** (representing a (VkImageView, VkSampler) pair) and texel coordinates as operands, and return a value based on one or more neighboring texture elements (*texels*) in the image.

Note
Texel is a term which is a combination of the words texture and element. Early interactive computer graphics supported texture operations on textures, a small subset of the image operations on images described here. The discrete samples remain essentially equivalent, however, so we retain the historical term texel to refer to them.

SPIR-V Image Instructions include the following functionality:

- **`OpImageSample`*** and **`OpImageSparseSample`*** read one or more neighboring texels of the image, and filter the texel values based on the state of the sampler.
  - Instructions with **`ImplicitLod`** in the name determine the level of detail used in the sampling operation based on the coordinates used in neighboring fragments.
  - Instructions with **`ExplicitLod`** in the name determine the level of detail used in the sampling operation based on additional coordinates.
  - Instructions with **`Proj`** in the name apply homogeneous projection to the coordinates.
- **`OpImageFetch`** and **`OpImageSparseFetch`** returns a single texel of the image. No sampler is used.
- **`OpImage`*****`Gather`** and **`OpImageSparse`*****`Gather`** read neighboring texels and returns a single component of each.
- **`OpImageRead`** (and **`OpImageSparseRead`**) and **`OpImageWrite`** read and write, respectively, a texel in the image. No sampler is used.
- Instructions with **`Dref`** in the name apply depth comparison on the texel values.
- Instructions with **`Sparse`** in the name additionally return a sparse residency code.

14.1.1 Texel Coordinate Systems

Images are addressed by *texel coordinates*. There are three *texel coordinate systems*:

- normalized texel coordinates (coordinates ranging from 0 to 1 span the image),
- unnormalized texel coordinates (floating point coordinates ranging from 0 to width/height/depth span the image), and
- integer texel coordinates (integer coordinates ranging from 0 to width-1/height-1/depth-1 address the texels within the image).

SPIR-V **`OpImageFetch`**, **`OpImageSparseFetch`**, **`OpImageRead`**, **`OpImageSparseRead`**, and **`OpImageWrite`** instructions use integer texel coordinates. Other image instructions can use either normalized or unnormalized texel coordinates (selected by the *`unnormalizedCoordinates`* state of the sampler used in the instruction), but there are limitations on what operations, image state, and sampler state is supported. Normalized coordinates are logically converted to unnormalized as part of image operations, and certain steps are only performed on normalized coordinates. The array layer coordinate is always treated as unnormalized even when other coordinates are normalized.

Normalized texel coordinates are referred to as (s,t,r,q,a), with the coordinates having the following meanings:

- s: Coordinate in the first dimension of an image.
- t: Coordinate in the second dimension of an image.
- r: Coordinate in the third dimension of an image.
  - (s,t,r) are interpreted as a direction vector for Cube images.
- q: Fourth coordinate, for homogeneous (projective) coordinates.
- a: Coordinate for array layer.

The coordinates are extracted from the SPIR-V operand based on the dimensionality of the image variable and type of instruction. For **`Proj`** instructions, the components are in order (s, [t,] [r,] q) with t and r being conditionally present based on the **`Dim`** of the image. For non-**`Proj`** instructions, the coordinates are (s [,t] [,r] [,a]), with t and r being conditionally present based on the **`Dim`** of the image and a being conditionally present based on the **`Arrayed`** property of the image. Projective image instructions are not supported on **`Arrayed`** images.

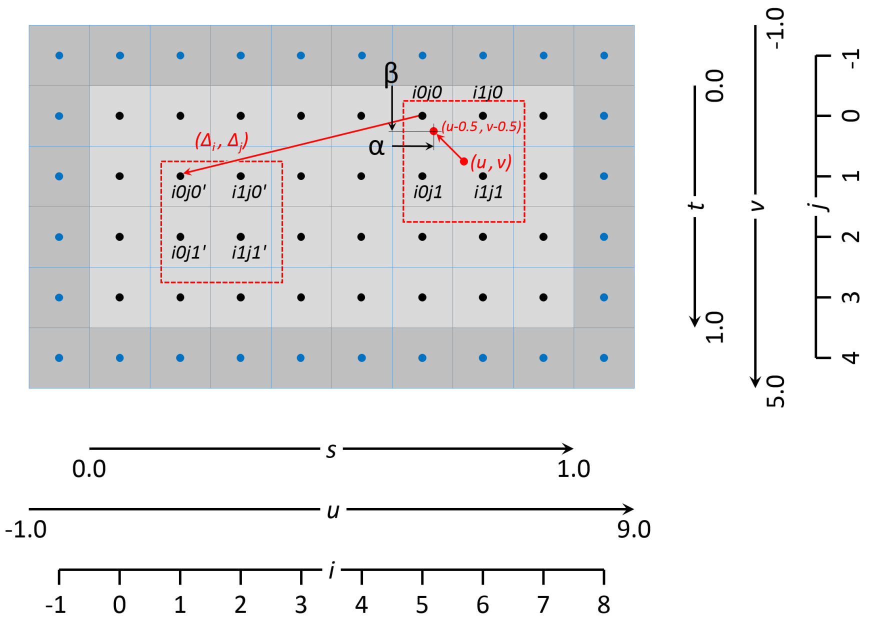
Unnormalized texel coordinates are referred to as (u,v,w,a), with the coordinates having the following meanings:

- u: Coordinate in the first dimension of an image.
- v: Coordinate in the second dimension of an image.
- w: Coordinate in the third dimension of an image.
- a: Coordinate for array layer.

Only the u and v coordinates are directly extracted from the SPIR-V operand, because only 1D and 2D (non-**`Arrayed`**) dimensionalities support unnormalized coordinates. The components are in order (u [,v]), with v being conditionally present when the dimensionality is 2D. When normalized coordinates are converted to unnormalized coordinates, all four coordinates are used.

Integer texel coordinates are referred to as (i,j,k,l,n), and the first four in that order have the same meanings as unnormalized texel coordinates. They are extracted from the SPIR-V operand in order (i, [,j], [,k], [,l]), with j and k conditionally present based on the **`Dim`** of the image, and l conditionally present based on the **`Arrayed`** property of the image. n is the sample index and is taken from the **`Sample`** image operand.

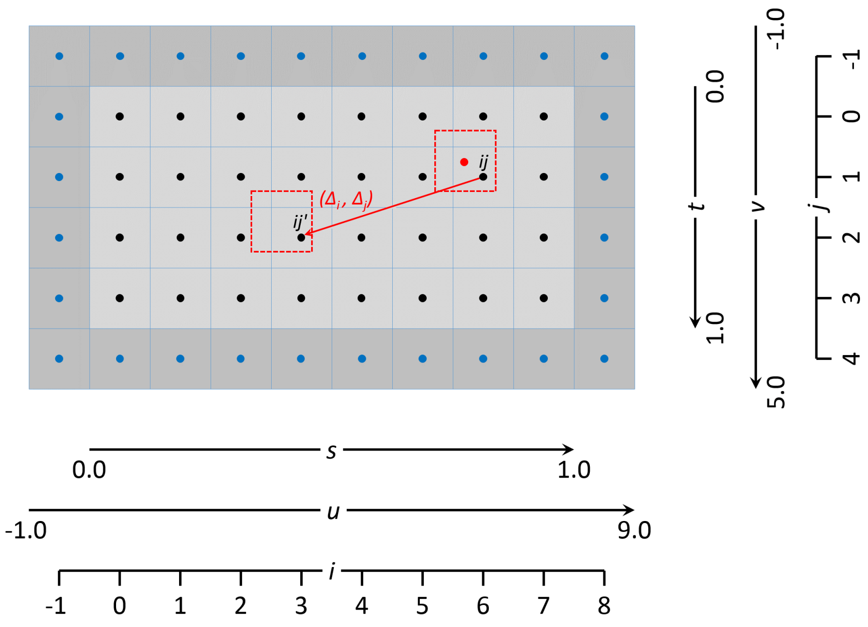
For all coordinate types, unused coordinates are assigned a value of zero.



The Texel Coordinate Systems - For the example shown of an 8x4 texel two dimensional image.

- Normalized texel coordinates:
  - The s coordinate goes from 0.0 to 1.0, left to right.
  - The t coordinate goes from 0.0 to 1.0, top to bottom.
- Unnormalized texel coordinates:
  - The u coordinate goes from -1.0 to 9.0, left to right. The u coordinate within the range 0.0 to 8.0 is within the image, otherwise it is within the border.
  - The v coordinate goes from -1.0 to 5.0, top to bottom. The v coordinate within the range 0.0 to 4.0 is within the image, otherwise it is within the border.
- Integer texel coordinates:
  - The i coordinate goes from -1 to 8, left to right. The i coordinate within the range 0 to 7 addresses texels within the image, otherwise it addresses a border texel.
  - The j coordinate goes from -1 to 5, top to bottom. The j coordinate within the range 0 to 3 addresses texels within the image, otherwise it addresses a border texel.
- Also shown for linear filtering:

- Given the unnormalized coordinates (u,v), the four texels selected are i0j0, i1j0, i0j1 and i1j1.
- The weights α and β.
- Given the offset Δ_i and Δ_j, the four texels selected by the offset are i0j0', i1j0', i0j1' and i1j1'.



The Texel Coordinate Systems - For the example shown of an 8x4 texel two dimensional image.

- Texel coordinates as above. Also shown for nearest filtering:
  - Given the unnormalized coordinates (u,v), the texel selected is ij.
  - Given the offset Δ_i and Δ_j, the texel selected by the offset is ij'.

14.2 Conversion Formulas

14.2.1 sRGB to Linear Conversion

An sRGB non-linear color c_{sRGB} is converted to a linear color space value c_{RGB} as follows:

$$c_{RGB} = \begin{cases} \frac{c_{sRGB}}{12.92} & \text{for } c_{sRGB} \leq 0.04045 \\ \left(\frac{c_{sRGB}+0.055}{1.055}\right)^{2.4} & \text{for } c_{sRGB} > 0.04045 \end{cases}$$

14.2.2 Linear to sRGB Conversion

An RGB linear color c_{RGB} is converted to an sRGB non-linear color space value c_{sRGB} as follows:

$$c_{sRGB} = \begin{cases} 0.0 & \text{for } c_{RGB} \leq 0 \\ 12.92 \times c_{RGB} & \text{for } 0 < c_{RGB} < 0.0031308 \\ 1.055 \times (c_{RGB})^{0.41666} - 0.055 & \text{for } 0.0031308 \leq c_{RGB} < 1 \\ 1.0 & \text{for } c_{RGB} \geq 1 \end{cases}$$

14.2.3 RGB to Shared Exponent Conversion

An RGB color $(red, green, blue)$ is transformed to a shared exponent color $(red_{shared}, green_{shared}, blue_{shared}, exp_{shared})$ as follows:

First, the components $(red, green, blue)$ are clamped to $(red_{clamped}, green_{clamped}, blue_{clamped})$ as:

$$\begin{aligned} red_{clamped} &= \max(0, min(sharedexp_{max}, red)) \\ green_{clamped} &= \max(0, min(sharedexp_{max}, green)) \\ blue_{clamped} &= \max(0, min(sharedexp_{max}, blue)) \end{aligned}$$

Where:

$N = 9$	number of mantissa bits per component
$B = 15$	exponent bias
$E_{max} = 31$	maximum possible biased exponent value
$sharedexp_{max} = \frac{(2^N - 1)}{2^N} \times 2^{(E_{max} - B)}$	

Note
NaN, if supported, is handled as in IEEE 754-2008 minNum() and maxNum(). That is the result is a NaN is mapped to zero.

The largest clamped component, $max_{clamped}$ is determined:

$$max_{clamped} = \max(red_{clamped}, green_{clamped}, blue_{clamped})$$

A preliminary shared exponent exp' is computed:

$$exp' = \max(-B - 1, \lfloor \log_2(max_{clamped} + 1 + B) \rfloor)$$

The shared exponent exp_{shared} is computed:

$$max_{shared} = \left\lfloor \frac{max_{clamped}}{2^{(exp' - B - N)}} + \frac{1}{2} \right\rfloor$$

$$exp_{shared} = \begin{cases} exp' & \text{for } 0 \leq max_{shared} < 2^N \\ exp' + 1 & \text{for } max_{shared} = 2^N \end{cases}$$

Finally, three integer values in the range 0 to 2^N are computed:

$$red_{shared} = \left\lfloor \frac{red_{clamped}}{2^{(exp_{shared}-B-N)}} + \frac{1}{2} \right\rfloor$$
$$green_{shared} = \left\lfloor \frac{green_{clamped}}{2^{(exp_{shared}-B-N)}} + \frac{1}{2} \right\rfloor$$
$$blue_{shared} = \left\lfloor \frac{blue_{clamped}}{2^{(exp_{shared}-B-N)}} + \frac{1}{2} \right\rfloor$$

14.2.4 Shared Exponent to RGB

A shared exponent color $(red_{shared}, green_{shared}, blue_{shared}, exp_{shared})$ is transformed to an RGB color $(red, green, blue)$ as follows:

$$red = red_{shared} \times 2^{(exp_{shared}-B-N)}$$
$$green = green_{shared} \times 2^{(exp_{shared}-B-N)}$$
$$blue = blue_{shared} \times 2^{(exp_{shared}-B-N)}$$

Where:

$N = 9$	number of mantissa bits per component
$B = 15$	exponent bias

14.3 Texel Input Operations

Texel input instructions are SPIR-V image instructions that read from an image. *Texel input operations* are a set of steps that are performed on state, coordinates, and texel values while processing a texel input instruction, and which are common to some or all texel input instructions. They include the following steps, which are performed in the listed order:

- Validation operations
  - Instruction/Sampler/Image validation
  - Coordinate validation
  - Sparse validation
- Format conversion
- Texel replacement
- Depth comparison
- Conversion to RGBA
- Component swizzle

For texel input instructions involving multiple texels (for sampling or gathering), these steps are applied for each texel that is used in the instruction. Depending on the type of image instruction, other steps are conditionally performed between these steps or involving multiple coordinate or texel values.

14.3.1 Texel Input Validation Operations

Texel input validation operations inspect instruction/image/sampler state or coordinates, and in certain circumstances cause the texel value to be replaced or become undefined. There are a series of validations that the texel undergoes.

14.3.1.1 Instruction/Sampler/Image Validation

There are a number of cases where a SPIR-V instruction can mismatch with the sampler, the image, or both. There are a number of cases where the sampler can mismatch with the image. In such cases the value of the texel returned is undefined.

These cases include:

- The sampler *`borderColor`* is an integer type and the image *`format`* is not one of the `VkFormat` integer types or a stencil aspect of a stencil or depth/stencil format.
- The sampler *`borderColor`* is a float type and the image *`format`* is not one of the `VkFormat` float types or a depth aspect of a depth or depth/stencil format.
- The sampler *`borderColor`* is one of the opaque black colors (VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK or VK_BORDER_COLOR_INT_OPAQUE_BLACK) and the image `VkComponentSwizzle` for any of the `VkComponentMapping` components is not VK_COMPONENT_SWIZZLE_IDENTITY.
- If the instruction is **`OpImageRead`** or **`OpImageSparseRead`** and the *`shaderStorageImageReadWithoutFormat`* feature is not enabled, or the instruction is **`OpImageWrite`** and the *`shaderStorageImageWriteWithoutFormat`* feature is not enabled, then the SPIR-V Image Format must be compatible with the image view's *`format`*.
- The SPIR-V instruction is one of the **`ImageSample`*****`ExplicitLod`** instructions and the sampler *`mipmapMode`* is VK_SAMPLER_MIPMAP_MODE_BASE.
- The sampler *`unnormalizedCoordinates`* is VK_TRUE and any of the limitations of unnormalized coordinates are violated.
- The SPIR-V instruction is one of the **`OpImage`*****`Dref`*** instructions and the sampler *`compareEnable`* is VK_FALSE
- The SPIR-V instruction is not one of the **`OpImage`*****`Dref`*** instructions and the sampler *`compareEnable`* is VK_TRUE
- The SPIR-V instruction is one of the **`OpImage`*****`Dref`*** instructions and the image *`format`* is not one of the depth or depth/stencil formats, or the image aspect is not VK_IMAGE_ASPECT_DEPTH_BIT.
- The SPIR-V instruction's image variable's properties are not compatible with the image view:
  - Rules for *`viewType`*:
    - VK_IMAGE_VIEW_TYPE_1D must have **`Dim`** = 1D, **`Arrayed`** = 0, **`MS`** = 0.
    - VK_IMAGE_VIEW_TYPE_2D must have **`Dim`** = 2D, **`Arrayed`** = 0.
    - VK_IMAGE_VIEW_TYPE_3D must have **`Dim`** = 3D, **`Arrayed`** = 0, **`MS`** = 0.
    - VK_IMAGE_VIEW_TYPE_CUBE must have **`Dim`** = Cube, **`Arrayed`** = 0, **`MS`** = 0.
    - VK_IMAGE_VIEW_TYPE_1D_ARRAY must have **`Dim`** = 1D, **`Arrayed`** = 1, **`MS`** = 0.
    - VK_IMAGE_VIEW_TYPE_2D_ARRAY must have **`Dim`** = 2D, **`Arrayed`** = 1.
    - VK_IMAGE_VIEW_TYPE_CUBE_ARRAY must have **`Dim`** = Cube, **`Arrayed`** = 1, **`MS`** = 0.
  - If the image's *`samples`* is not equal to VK_SAMPLE_COUNT_1_BIT, the instruction must have **`MS`** = 1.

14.3.1.2 Integer Texel Coordinate Validation

Integer texel coordinates are validated against the size of the image level, and the number of layers and number of samples in the image. For SPIR-V instructions that use integer texel coordinates, this is performed directly on the integer coordinates. For instructions that use normalized or unnormalized texel coordinates, this is performed on the coordinates that result after conversion to integer texel coordinates.

If the integer texel coordinates satifies any of the conditions

$$\begin{aligned} i &< 0 & i &\geq w_s \\ j &< 0 & j &\geq h_s \\ k &< 0 & k &\geq d_s \\ l &< 0 & l &\geq layers \\ n &< 0 & n &\geq samples \end{aligned}$$

where:

$$\begin{aligned} w_s &= \text{width of the image level} \\ h_s &= \text{height of the image level} \\ d_s &= \text{depth of the image level} \\ layers &= \text{number of layers in the image} \\ samples &= \text{number of samples per texel in the image} \end{aligned}$$

then the texel fails integer texel coordinate validation.

There are four cases to consider:

- Valid Texel Coordinates
  - If the texel coordinates pass validation (that is, the coordinates lie within the image),
    then the texel value comes from the value in image memory.
- Border Texel
  - If the texel coordinates fail validation, and
  - If the read is the result of an image sample instruction or image gather instruction, and
  - If the image is not a cube image,
    then the texel is a border texel and texel replacement is performed.
- Invalid Texel
  - If the texel coordinates fail validation, and
  - If the read is the result of an image fetch instruction, image read instruction, or atomic instruction,
    then the texel is an invalid texel and texel replacement is performed.
- Cube Map Edge or Corner
  - Otherwise the texel coordinates lie on the borders along the edges and corners of a cube map image, and Cube map edge handling is performed.

14.3.1.3 Cube Map Edge Handling

If the texel coordinates lie on the borders along the edges and corners of a cube map image, the following steps are performed. Note that this only occurs when using VK_FILTER_LINEAR filtering within a miplevel, since VK_FILTER_NEAREST is treated as using VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE.

- Cube Map Edge Texel
  - If the texel lies along the border in either only i or only j
    then the texel lies along an edge, so the coordinates (i, j) and the array layer l are transformed to select the adjacent texel from the appropriate neighboring face.
- Cube Map Corner Texel
  - If the texel lies along the border in both i and j
    then the texel lies at the corner and there is no unique neighboring face from which to read that texel. The texel should be replaced by the average of the three values of the adjacent texels in each incident face. However, implementations may replace the cube map corner texel by other methods, subject to the constraint that if the three available samples have the same value, the replacement texel also has that value.

14.3.1.4 Sparse Validation

If the texel reads from an unbound region of a sparse image, the texel is a sparse unbound texel, and processing continues with texel replacement.

14.3.2 Format Conversion

Texels undergo a format conversion from the `VkFormat` of the image view to a vector of either floating point or signed or unsigned integer components, with the number of components based on the number of components present in the format.

- Color formats have one, two, three, or four components, according to the format.
- Depth, stencil, and depth/stencil formats are one component. For depth/stencil formats, the depth or stencil component is selected by the *`aspectMask`* of the image view.

Each component is converted based on its type and size (as defined in the Format Definition section for each `VkFormat`), using the appropriate equations in 16-Bit Floating-Point Numbers, Unsigned 11-Bit Floating-Point Numbers, Unsigned 10-Bit Floating-Point Numbers, Fixed-Point Data Conversion, and Shared Exponent to RGB.

If the image format is sRGB, the color components are first converted as if they are UNORM, and then sRGB to linear conversion is applied to the R, G, and B components. The A component, if present, is unchanged.

If the image view format is block-compressed, then the texel value is first decoded, then converted based on the type and number of components defined by the compressed format.

14.3.3 Texel Replacement

A texel is replaced if it is one (and only one) of:

- a border texel, or

- an invalid texel, or
- a sparse unbound texel.

Border texels are replaced with a value based on the image format and the `borderColor` of the sampler. The border color is:

Table 14.1: Border Color B

Sampler **`borderColor`**	Corresponding Border Color
VK_BORDER_COLOR_FLOAT_TRANSPARENT_BLACK	$B = (0.0, 0.0, 0.0, 0.0)$
VK_BORDER_COLOR_FLOAT_OPAQUE_BLACK	$B = (0.0, 0.0, 0.0, 1.0)$
VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE	$B = (1.0, 1.0, 1.0, 1.0)$
VK_BORDER_COLOR_INT_TRANSPARENT_BLACK	$B = (0, 0, 0, 0)$
VK_BORDER_COLOR_INT_OPAQUE_BLACK	$B = (0, 0, 0, 1)$
VK_BORDER_COLOR_INT_OPAQUE_WHITE	$B = (1, 1, 1, 1)$

This is substituted for the texel value by replacing the number of components in the image format

Table 14.2: Border Texel Components After Replacement

Texel Aspect or Format	Component Assignment
Depth aspect	$D = (B_r)$
Stencil aspect	$S = (B_r)$
One component color format	$C_r = (B_r)$
Two component color format	$C_{rg} = (B_r, B_g)$
Three component color format	$C_{rgb} = (B_r, B_g, B_b)$
Four component color format	$C_{rgba} = (B_r, B_g, B_b, B_a)$

If the read operation is from a buffer resource, and robustBufferAccess is enabled, the texel is replaced with zero values (as though $B = (zero, zero, zero, zero)$ in the replacement table above, where $zero = 0.0f$ for floating-point color formats and depth aspects, and $zero = 0$ for integer color formats and stencil aspects.

If the read operation is not from a buffer resource, or robustBufferAccess is disabled, the texel value is undefined.

If the VkPhysicalDeviceSparseProperties property `residencyNonResidentStrict` is true, a sparse unbound texel is replaced with zero values in the same fashion as described for reads from buffer resources above.

If `residencyNonResidentStrict` is false, the read must be safe, but the value of the sparse unbound texel is undefined.

14.3.4 Depth Compare Operation

If the image view's format is depth and the operation is a **`Dref`** instruction, a depth comparison is performed. The initial value of the result r is 0.0, which is replaced with 1.0 if the result of the compare operation is $true$. The compare operation is selected by the `compareOp` member of the sampler.

$$r = 0.0 \qquad \text{initial value}$$

$$r = 1.0 \begin{cases} D_{ref} \leq D_t & \text{for LEQUAL} \\ D_{ref} \geq D_t & \text{for GEQUAL} \\ D_{ref} < D_t & \text{for LESS} \\ D_{ref} > D_t & \text{for GREATER} \\ D_{ref} = D_t & \text{for EQUAL} \\ D_{ref} \neq D_t & \text{for NOTEQUAL} \\ true & \text{for ALWAYS} \\ false & \text{for NEVER} \end{cases}$$

where:

$$D_{ref} = shaderOp.D_{ref} \qquad \text{(from optional SPIR-V operand)}$$
$$D_t \qquad \text{texel depth value}$$

14.3.5 Conversion to RGBA

The texel is expanded from one, two, or three to four components based on the image base color:

Table 14.3: Border Texel Components After Replacement

Texel Aspect or Format	**RGBA Color**
Depth aspect	$C_{rgba} = (D, 0, 0, one)$
Stencil aspect	$C_{rgba} = (S, 0, 0, one)$
One component color format	$C_{rgba} = (C_r, 0, 0, one)$
Two component color format	$C_{rgba} = (C_{rg}, 0, one)$
Three component color format	$C_{rgba} = (C_{rgb}, one)$
Four component color format	$C_{rgba} = C_{rgba}$

where $one = 1.0f$ for floating-point formats and depth aspects, and $one = 1$ for integer formats and stencil aspects.

14.3.6 Component Swizzle

All texel input instructions apply a *swizzle* based on the `VkComponentSwizzle` enums in the `components` member of the `VkImageViewCreateInfo` structure for the image being read. The swizzle can rearrange the components of the texel, or substitute zero and one for any components. It is defined as follows for the R component, and operates similarly for the other components.

$$C'_{rgba}[R] = \begin{cases} C_{rgba}[R] & \text{for RED swizzle} \\ C_{rgba}[G] & \text{for GREEN swizzle} \\ C_{rgba}[B] & \text{for BLUE swizzle} \\ C_{rgba}[A] & \text{for ALPHA swizzle} \\ 0 & \text{for ZERO swizzle} \\ one & \text{for ONE swizzle} \\ C_{rgba}[R] & \text{for IDENTITY swizzle} \end{cases}$$

where:

$$\begin{aligned} C_{rgba}[R] & \text{is the RED component} \\ C_{rgba}[G] & \text{is the GREEN component} \\ C_{rgba}[B] & \text{is the BLUE component} \\ C_{rgba}[A] & \text{is the ALPHA component} \\ one = 1.0\text{f} & \qquad \text{for floating point components} \\ one = 1 & \qquad \text{for integer components} \end{aligned}$$

For each component this is applied to, the VK_COMPONENT_SWIZZLE_IDENTITY swizzle selects the corresponding component from C_{rgba}.

If the border color is one of the VK_BORDER_COLOR_*_OPAQUE_BLACK enums and the `VkComponentSwizzle` is not VK_COMPONENT_SWIZZLE_IDENTITY for all components (or the equivalent identity mapping), the value of the texel after swizzle is undefined.

14.4 Texel Output Operations

Texel output instructions are SPIR-V image instructions that write to an image. *Texel output operations* are a set of steps that are performed on state, coordinates, and texel values while processing a texel output instruction, and which are common to some or all texel ouitput instructions. They include the following steps, which are performed in the listed order:

- Validation operations
  - Format validation
  - Coordinate validation
  - Sparse validation
- Texel output format conversion

14.4.1 Texel Output Validation Operations

Texel output validation operations inspect instruction/image state or coordinates, and in certain circumstances cause the write to have no effect. There are a series of validations that the texel undergoes.

14.4.1.1 Texel Format Validation

If the image format of the *`OpTypeImage`* is not compatible with the VkImageView's *`format`*, the effect of the write on the image view's memory is undefined, but the write must not access memory outside of the image view.

14.4.2 Integer Texel Coordinate Validation

The integer texel coordinates are validated according to the same rules as for texel input coordinate validation.

If the texel fails integer texel coordinate validation, then the write has no effect.

14.4.3 Sparse Texel Operation

If the texel attempts to write to an unbound region of a sparse image, the texel is a sparse unbound texel. In such a case, if the VkPhysicalDeviceSparseProperties property *`residencyNonResidentStrict`* is VK_TRUE, the sparse unbound texel write has no effect. If *`residencyNonResidentStrict`* is VK_FALSE, the effect of the write is undefined but must be safe. In addition, the write may have a side effect that is visible to other image instructions, but must not be written to any device memory allocation.

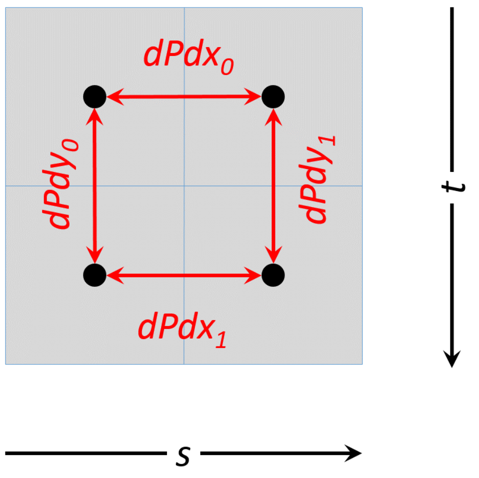
14.4.4 Texel Output Format Conversion

Texels undergo a format conversion from the floating point, signed, or unsigned integer type of the texel data to the `VkFormat` of the image view. Any unused components are ignored.

Each component is converted based on its type and size (as defined in the Format Definition section for each `VkFormat`), using the appropriate equations in 16-Bit Floating-Point Numbers and Fixed-Point Data Conversion.

14.5 Derivative Operations

SPIR-V derivative instructions include **`OpDPdx`**, **`OpDPdy`**, **`OpDPdxFine`**, **`OpDPdyFine`**, **`OpDPdxCoarse`**, and **`OpDPdyCoarse`**. Derivative instructions are only available in a fragment shader.



Derivatives are computed as if there is a 2x2 neighborhood of fragments for each fragment shader invocation. These neighboring fragments are used to compute derivatives with the assumption that the values of P in the neighborhood are piecewise linear. It is further assumed that the values of P in the neighborhood are locally continuous, therefore derivatives in non-uniform control flow are undefined.

$$\begin{aligned} dPdx_{i_1,j_0} = dPdx_{i_0,j_0} &\qquad = P_{i_1,j_0} - P_{i_0,j_0} \\ dPdx_{i_1,j_1} = dPdx_{i_0,j_1} &\qquad = P_{i_1,j_1} - P_{i_0,j_1} \end{aligned}$$

$$\begin{aligned} dPdy_{i_0,j_1} = dPdy_{i_0,j_0} &\qquad = P_{i_0,j_1} - P_{i_0,j_0} \\ dPdy_{i_1,j_1} = dPdy_{i_1,j_0} &\qquad = P_{i_1,j_1} - P_{i_1,j_0} \end{aligned}$$

The **`Fine`** derivative instructions must return the values above, for a group of fragments in a 2x2 neighborhood. Coarse derivatives may return only two values. In this case, the values should be:

$$dPdx = \begin{cases} dPdx_{i_0,j_0} & \text{preferred} \\ dPdx_{i_0,j_1} \end{cases}$$

$$dPdy = \begin{cases} dPdy_{i_0,j_0} & \text{preferred} \\ dPdy_{i_1,j_0} \end{cases}$$

`OpDPdx` and **`OpDPdy`** must return: the same result as either **`OpDPdxFine`** or **`OpDPdxCoarse`** and either **`OpDPdyFine`** or **`OpDPdyCoarse`**, respectively. Implementations must make the same choice of either coarse or fine for both **`OpDPdx`** and **`OpDPdy`**, and implementations should make the choice that is more efficient to compute.

14.6 Normalized Texel Coordinate Operations

If the image sampler instruction provides normalized texel coordinates, some of the following operations are performed.

14.6.1 Projection Operation

For **`Proj`** image operations, the normalized texel coordinates (s,t,r,q,a) and (if present) the D_{ref} coordinate are transformed as follows:

$$s = \frac{s}{q}, \qquad \text{for 1D, 2D, or 3D image}$$

$$t = \frac{t}{q}, \qquad \text{for 2D or 3D image}$$

$$r = \frac{r}{q}, \qquad \text{for 3D image}$$

$$D_{ref} = \frac{D_{ref}}{q}, \qquad \text{if provided}$$

14.6.2 Derivative Image Operations

Derivatives are used for level-of-detail selection. These derivatives are either implicit (in an **`ImplicitLod`** image instruction in a fragment shader) or explicit (provided explicitly by shader to the image instruction in any shader).

For implicit derivatives image instructions, the derivatives of texel coordinates are calculated in the same manner as derivative operations above. That is:

$$\begin{aligned} \partial s/\partial x &= dPdx(s), & \partial s/\partial y &= dPdy(s), & &\text{for 1D, 2D, Cube, or 3D image} \\ \partial t/\partial x &= dPdx(t), & \partial t/\partial y &= dPdy(t), & &\text{for 2D, Cube, or 3D image} \\ \partial u/\partial x &= dPdx(u), & \partial u/\partial y &= dPdy(u), & &\text{for Cube or 3D image} \end{aligned}$$

Partial derivatives not defined above for certain image dimensionalities are set to zero.

For explicit level-of-detail image instructions, if the optional SPIR-V operand *Grad* is provided, then the operand values are used for the derivatives. The number of components present in each derivative for a given image dimensionality matches the number of partial derivatives computed above.

If the optional SPIR-V operand *Lod* is provided, then derivatives are set to zero, the cube map derivative transformation is skipped, and the scale factor operation is skipped. Instead, the floating point scalar coordinate is directly assigned to λ_{base} as described in Level-of-Detail Operation.

14.6.3 Cube Map Face Selection and Transformations

For cube map image instructions, the (s,t,r) coordinates are treated as a direction vector (r_x, r_y, r_z). The direction vector is used to select a cube map face. The direction vector is transformed to a per-face texel coordinate system (s_{face}, t_{face}). The direction vector is also used to transform the derivatives to per-face derivatives.

14.6.4 Cube Map Face Selection

The direction vector selects one of the cube maps face's layers based on the largest magnitude coordinate direction (the major axis direction). Since two or more coordinates can have identical magnitude, the implementation must have rules to disambiguate this situation.

The rules should have as the first rule that r_z wins over r_y and r_x, and the second rule that r_y wins over r_x. An implementation may choose other rules, but the rules must be deterministic and depend only on (r_x, r_y, r_z).

The layer number (corresponding to a cube map face), the coordinate selections for s_c, t_c, r_c, and the selection of derivatives, are determined by the major axis direction as specified in the following two tables.

Table 14.4: Cube map face and coordinate selection

Major Axis Direction	**Layer Number**	**Cube Map Face**	s_c	t_c	r_c
$+r_x$	0	*PositiveX*	$-r_z$	$-r_y$	r_x
$-r_x$	1	*NegativeX*	$+r_z$	$-r_y$	r_x
$+r_y$	2	*PositiveY*	$+r_x$	$+r_z$	r_y
$-r_y$	3	*NegativeY*	$+r_x$	$-r_z$	r_y
$+r_z$	4	*PositiveZ*	$+r_x$	$-r_y$	r_z
$-r_z$	5	*NegativeZ*	$-r_x$	$-r_y$	r_z

Table 14.5: Cube map derivative selection

Major Axis Direction	$\partial s_c/\partial x$	$\partial s_c/\partial y$	$\partial t_c/\partial x$	$\partial t_c/\partial y$	$\partial r_c/\partial x$	$\partial r_c/\partial y$
$+r_x$	$-\partial r_z/\partial x$	$-\partial r_z/\partial y$	$-\partial r_y/\partial x$	$-\partial r_y/\partial y$	$+\partial r_x/\partial x$	$+\partial r_x/\partial y$
$-r_x$	$+\partial r_z/\partial x$	$+\partial r_z/\partial y$	$-\partial r_y/\partial x$	$-\partial r_y/\partial y$	$-\partial r_x/\partial x$	$-\partial r_x/\partial y$
$+r_y$	$+\partial r_x/\partial x$	$+\partial r_x/\partial y$	$+\partial r_z/\partial x$	$+\partial r_z/\partial y$	$+\partial r_y/\partial x$	$+\partial r_y/\partial y$
$-r_y$	$+\partial r_x/\partial x$	$+\partial r_x/\partial y$	$-\partial r_z/\partial x$	$-\partial r_z/\partial y$	$-\partial r_y/\partial x$	$-\partial r_y/\partial y$
$+r_z$	$+\partial r_x/\partial x$	$+\partial r_x/\partial y$	$-\partial r_y/\partial x$	$-\partial r_y/\partial y$	$+\partial r_z/\partial x$	$+\partial r_z/\partial y$
$-r_z$	$-\partial r_x/\partial x$	$-\partial r_x/\partial y$	$-\partial r_y/\partial x$	$-\partial r_y/\partial y$	$-\partial r_z/\partial x$	$-\partial r_z/\partial y$

14.6.5 Cube Map Coordinate Transformation

$$s_{face} = \frac{1}{2} \times \frac{s_c}{|r_c|} + \frac{1}{2}$$
$$t_{face} = \frac{1}{2} \times \frac{t_c}{|r_c|} + \frac{1}{2}$$

14.6.6 Cube Map Derivative Transformation

$$\frac{\partial s_{face}}{\partial x} = \frac{\partial}{\partial x}\left(\frac{1}{2} \times \frac{s_c}{|r_c|} + \frac{1}{2}\right)$$
$$\frac{\partial s_{face}}{\partial x} = \frac{1}{2} \times \frac{\partial}{\partial x}\left(\frac{s_c}{|r_c|}\right)$$
$$\frac{\partial s_{face}}{\partial x} = \frac{1}{2} \times \left(\frac{|r_c| \times \partial s_c/\partial x - s_c \times \partial r_c/\partial x}{(r_c)^2}\right)$$

$$\frac{\partial s_{face}}{\partial y} = \frac{1}{2} \times \left(\frac{|r_c| \times \partial s_c/\partial y - s_c \times \partial r_c/\partial y}{(r_c)^2}\right)$$
$$\frac{\partial t_{face}}{\partial x} = \frac{1}{2} \times \left(\frac{|r_c| \times \partial t_c/\partial x - t_c \times \partial r_c/\partial x}{(r_c)^2}\right)$$
$$\frac{\partial t_{face}}{\partial y} = \frac{1}{2} \times \left(\frac{|r_c| \times \partial t_c/\partial y - t_c \times \partial r_c/\partial y}{(r_c)^2}\right)$$

14.6.7 Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection

Level-of-detail selection can be either explicit (provided explicitly by the image instruction) or implicit (determined from a scale factor calculated from the derivatives).

14.6.7.1 Scale Factor Operation

The magnitude of the derivatives are calculated by:

$$\begin{aligned} m_{ux} &= |\partial s/\partial x| \times w_{base} \\ m_{vx} &= |\partial t/\partial x| \times h_{base} \\ m_{wx} &= |\partial r/\partial x| \times d_{base} \\ \\ m_{uy} &= |\partial s/\partial y| \times w_{base} \\ m_{vy} &= |\partial t/\partial y| \times h_{base} \\ m_{wy} &= |\partial r/\partial y| \times d_{base} \end{aligned}$$

where:

$$\begin{aligned} \partial t/\partial x &= \partial t/\partial y = 0 && \text{(for 1D image)} \\ \partial r/\partial x &= \partial r/\partial y = 0 && \text{(for 1D, 2D or Cube image)} \\ \\ w_{base} &= image.w \\ h_{base} &= image.h \\ d_{base} &= image.d \\ &\text{of the } baseMipLevel && \text{(from image descriptor)} \end{aligned}$$

The *scale factors* $(\rho_x, \rho y)$ should be calculated by:

$$\begin{aligned} \rho_x &= \sqrt{m_{ux}^2 + m_{vx}^2 + m_{wx}^2} \\ \rho_y &= \sqrt{m_{uy}^2 + m_{vy}^2 + m_{wy}^2} \end{aligned}$$

The ideal functions ρ_x and ρ_y may be approximated with functions f_x and f_y, subject to the following constraints:

$$f_x \text{ is continuous and monotonically increasing in each of} m_{ux}, m_{vx}, \text{and} m_{wx}$$
$$f_y \text{ is continuous and monotonically increasing in each of} m_{uy}, m_{vy}, \text{and} m_{wy}$$

$$\max(|m_{ux}|, |m_{vx}|, |m_{wx}|) \le f_x \le |m_{ux}| + |m_{vx}| + |m_{wx}|$$
$$\max(|m_{uy}|, |m_{vy}|, |m_{wy}|) \le f_y \le |m_{uy}| + |m_{vy}| + |m_{wy}|$$

The minimum and maximum scale factors (ρ_{min}, ρ_{max}) are determined by:

$$\begin{aligned} \rho_{max} &= \max(\rho_x, \rho_y) \\ \rho_{min} &= \min(\rho_x, \rho_y) \end{aligned}$$

The sampling rate is determined by:

$$N = \min\left(\left\lceil \frac{\rho_{max}}{\rho_{min}} \right\rceil, max_{Aniso}\right)$$

where:

$$\begin{aligned} sampler.max_{Aniso} &= maxAnisotropy && \text{(from sampler descriptor)} \\ limits.max_{Aniso} &= maxSamplerAnisotropy && \text{(from physical device limits)} \\ max_{Aniso} &= \min(sampler.max_{Aniso}, limits.max_{Aniso}) \end{aligned}$$

An implementation may round N up to the nearest supported sampling rate.

If $N = 1$, sampling is isotropic. If $N > 1$, sampling is anistropic.

14.6.7.2 Level-of-Detail Operation

The *level-of-detail* parameter λ is computed as follows:

$$\lambda_{base}(x,y) = \begin{cases} shaderOp.Lod & \text{(from optional SPIR-V operand)} \\ \log_2\left(\frac{\rho_{max}}{N}\right) & \text{otherwise} \end{cases}$$

$$\lambda'(x,y) = \lambda_{base} + \text{clamp}(sampler.bias + shaderOp.bias)$$

$$\lambda = \begin{cases} lod_{max}, & \lambda' > lod_{max} \\ \lambda', & lod_{min} \leq \lambda' \leq lod_{max} \\ lod_{min}, & \lambda' < lod_{min} \\ undefined, & lod_{min} > lod_{max} \end{cases}$$

where:

$$sampler.bias = mipLodBias \qquad \text{(from sampler descriptor)}$$

$$shaderOp.bias = \begin{cases} Bias & \text{(from optional SPIR-V operand)} \\ 0 & \text{otherwise} \end{cases}$$

$$sampler.lod_{min} = minLod \qquad \text{(from sampler descriptor)}$$

$$shaderOp.lod_{min} = \begin{cases} MinLod & \text{(from optional SPIR-V operand)} \\ 0 & \text{otherwise} \end{cases}$$

$$lod_{min} = \max(sampler.lod_{min}, shaderOp.lod_{min})$$

$$lod_{max} = maxLod \qquad \text{(from sampler descriptor)}$$

14.6.7.3 Image Level(s) Selection

The image level(s) $d, d_{hi},$ and d_{lo} which texels are read from are selected based on the level-of-detail parameter, as follows. If the sampler's `mipmapMode` is VK_SAMPLER_MIPMAP_MODE_NEAREST, then level d is used:

$$d = \begin{cases} level_{base}, & \lambda \leq \frac{1}{2} \\ nearest(\lambda), & \lambda > \frac{1}{2}, level_{base} + \lambda \leq q + \frac{1}{2} \\ q, & \lambda > \frac{1}{2}, level_{base} + \lambda > q + \frac{1}{2} \end{cases}$$

where:

$$nearest(\lambda) = \begin{cases} \left\lceil level_{base} + \lambda + \frac{1}{2} \right\rceil - 1, & \text{preferred} \\ \left\lfloor level_{base} + \lambda + \frac{1}{2} \right\rfloor, & \text{alternative} \end{cases}$$

and where q is the `levelCount` from the `subresourceRange` of the image view.

If the sampler's `mipmapMode` is VK_SAMPLER_MIPMAP_MODE_LINEAR, two neighboring levels are selected:

$$d_{hi} = \begin{cases} q, & level_{base} + \lambda \geq q \\ \left\lfloor level_{base} + \lambda \right\rfloor, & \text{otherwise} \end{cases}$$

$$d_{lo} = \begin{cases} q, & level_{base} + \lambda \geq q \\ d_{hi} + 1, & \text{otherwise} \end{cases}$$

δ is the fractional value used for linear filtering between levels.

$$\delta = \text{frac}(\lambda)$$

If the sampler's `mipmapMode` is VK_SAMPLER_MIPMAP_MODE_BASE, the base level is used:

$$d = level_{base}$$

14.6.8 (s,t,r,q,a) to (u,v,w,a) Transformation

The normalized texel coordinates are scaled by the image level dimensions and the array layer is selected. This transformation is performed once for each level (d or d_{hi} and d_{lo}) used in filtering.

$$u(x,y) = s(x,y) \times width_{level}$$
$$v(x,y) = \begin{cases} 0 & \text{for 1D images} \\ t(x,y) \times height_{level} & \text{otherwise} \end{cases}$$
$$w(x,y) = \begin{cases} 0 & \text{for 2D or Cube images} \\ r(x,y) \times depth_{level} & \text{otherwise} \end{cases}$$

$$a(x,y) = \begin{cases} a(x,y) & \text{for array images} \\ 0 & \text{otherwise} \end{cases}$$

Operations then proceed to Unnormalized Texel Coordinate Operations.

14.7 Unnormalized Texel Coordinate Operations

14.7.1 (u,v,w,a) to (i,j,k,l,n) Transformation And Array Layer Selection

The unnormalized texel coordinates are transformed to integer texel coordinates relative to the selected mipmap level.

The layer index l is computed as:

$$l = \text{clamp}(\text{RNE}(a), 0, layerCount - 1) + baseArrayLayer$$

where `layerCount` is the number of layers in the subresource range of the image view, `baseArrayLayer` is the first layer from the subresource range, and where:

$$\text{RNE}(a) = \begin{cases} \text{roundTiesToEven}(a) & \text{preferred, from IEEE Std 754-2008 Floating-Point Arithmetic} \\ \lfloor a + \frac{1}{2} \rfloor & \text{alternative} \end{cases}$$

The sample index n is assigned the value zero.

Nearest filtering (VK_FILTER_NEAREST) computes the integer texel coordinates that the unnormalized coordinates lie within:

$$i = \lfloor u \rfloor$$
$$j = \lfloor v \rfloor$$
$$k = \lfloor w \rfloor$$

Linear filtering (VK_FILTER_LINEAR) computes a set of neighboring coordinates which bound the unnormalized coordinates. The integer texel coordinates are combinations of i_0 or i_1, j_0 or j_1, k_0 or k_1, as well as weights $\alpha, \beta, and \gamma$.

$$i_0 = \left\lfloor u - \frac{1}{2} \right\rfloor \qquad i_1 = i_0 + 1$$
$$j_0 = \left\lfloor v - \frac{1}{2} \right\rfloor \qquad j_1 = j_0 + 1$$
$$k_0 = \left\lfloor w - \frac{1}{2} \right\rfloor \qquad k_1 = k_0 + 1$$

$$\alpha = \operatorname{frac}\left(u - \frac{1}{2}\right)$$
$$\beta = \operatorname{frac}\left(v - \frac{1}{2}\right)$$
$$\gamma = \operatorname{frac}\left(w - \frac{1}{2}\right)$$

If the image instruction includes a $ConstOffset$ operand, the constant offsets $(\Delta_i, \Delta_j, \Delta_k)$ are added to (i, j, k) components of the integer texel coordinates.

14.8 Image Sample Operations

14.8.1 Wrapping Operation

`Cube` images ignore the wrap modes specified in the sampler. Instead, if VK_FILTER_NEAREST is used within a miplevel then VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE is used, and if VK_FILTER_LINEAR is used within a miplevel then sampling at the edges is performed as described earlier in the Cube map edge handling section.

The first integer texel coordinate i is transformed based on the *`addressModeU`* parameter of the sampler.

$$i = \begin{cases} i \bmod size & \text{for repeat} \\ (size - 1) - \operatorname{mirror}((i \bmod (2 \times size)) - size) & \text{for mirrored repeat} \\ \operatorname{clamp}(i, 0, size - 1) & \text{for clamp to edge} \\ \operatorname{clamp}(i, -1, size) & \text{for clamp to border} \\ \operatorname{clamp}(\operatorname{mirror}(i), 0, size - 1) & \text{for mirror clamp to edge} \end{cases}$$

where:

$$\operatorname{mirror}(n) = \begin{cases} n & \text{for } n \geq 0 \\ -(1 + n) & \text{otherwise} \end{cases}$$

j (for 2D and Cube image) and k (for 3D image) are similarly transformed based on the *`addressModeV`* and *`addressModeW`* parameters of the sampler, respectively.

14.8.2 Texel Gathering

SPIR-V instructions with **`Gather`** in the name return a vector derived from a 2x2 block of texels in the base level of the image view. The rules for the LINEAR minification filter are applied to identify the four selected texels. Each texel is then converted to an RGBA value according to conversion to RGBA and then swizzled. A four-component vector is then assembled by taking the component indicated by the **`Component`** value in the instruction from the swizzled color value of the four texels:

$$\begin{aligned}
\tau[R] &= \tau_{i0j1}[level_{base}][comp] \\
\tau[G] &= \tau_{i1j1}[level_{base}][comp] \\
\tau[B] &= \tau_{i1j0}[level_{base}][comp] \\
\tau[A] &= \tau_{i0j0}[level_{base}][comp]
\end{aligned}$$

where:

$$\tau[level_{base}][comp] = \begin{cases} \tau[level_{base}][R], & \text{for } comp = 0 \\ \tau[level_{base}][G], & \text{for } comp = 1 \\ \tau[level_{base}][B], & \text{for } comp = 2 \\ \tau[level_{base}][A], & \text{for } comp = 3 \end{cases}$$

$$comp \text{ from SPIR-V operand Component}$$

14.8.3 Texel Filtering

If λ is less than or equal to zero, the texture is said to be *magnified*, and the filter mode within a mip level is selected by the `magFilter` in the sampler. If λ is greater than zero, the texture is said to be *minified*, and the filter mode within a mip level is selected by the `minFilter` in the sampler.

Within a miplevel, NEAREST filtering selects a single value using the (i, j, k) texel coordinates, with all texels taken from layer l.

$$\tau[level] = \begin{cases} \tau_{ijk}[level], & \text{for 3D image} \\ \tau_{ij}[level], & \text{for 2D or Cube image} \\ \tau_{i}[level], & \text{for 1D image} \end{cases}$$

Within a miplevel, LINEAR filtering computes a weighted average of 8 (for 3D), 4 (for 2D or ube), or 2 (for 1D) texel values, using the weights computed earlier:

$$\begin{aligned}
\tau_{3D}[level] = {} & (1-\alpha)(1-\beta)(1-\gamma)\tau_{i0j0k0}[level] \\
&+ (\alpha)(1-\beta)(1-\gamma)\tau_{i1j0k0}[level] \\
&+ (1-\alpha)(\beta)(1-\gamma)\tau_{i0j1k0}[level] \\
&+ (\alpha)(\beta)(1-\gamma)\tau_{i1j1k0}[level] \\
&+ (1-\alpha)(1-\beta)(\gamma)\tau_{i0j0k1}[level] \\
&+ (\alpha)(1-\beta)(\gamma)\tau_{i1j0k1}[level] \\
&+ (1-\alpha)(\beta)(\gamma)\tau_{i0j1k1}[level] \\
&+ (\alpha)(\beta)(\gamma)\tau_{i1j1k1}[level]
\end{aligned}$$

$$\begin{aligned}
\tau_{2D}[level] = {} & (1-\alpha)(1-\beta)\tau_{i0j0}[level] \\
&+ (\alpha)(1-\beta)\tau_{i1j0}[level] \\
&+ (1-\alpha)(\beta)\tau_{i0j1}[level] \\
&+ (\alpha)(\beta)\tau_{i1j1}[level]
\end{aligned}$$

$$\begin{aligned}
\tau_{1D}[level] = {} & (1-\alpha)\tau_{i0}[level] \\
&+ (\alpha)\tau_{i1}[level]
\end{aligned}$$

$$\tau[level] = \begin{cases} \tau_{3D}[level], & \text{for 3D image} \\ \tau_{2D}[level], & \text{for 2D or Cube image} \\ \tau_{1D}[level], & \text{for 1D image} \end{cases}$$

Finally, mipmap filtering either selects a value from one miplevel or computes a weighted average between neighboring miplevels:

$$\tau = \begin{cases} \tau[d], & \text{for mipmode BASE or NEAREST} \\ (1-\delta)\tau[d_{hi}] + \delta\tau[d_{lo}], & \text{for mipmode LINEAR} \end{cases}$$

14.8.4 Texel Anisotropic Filtering

Anisotropic filtering is enabled by the *`anisotropyEnable`* in the sampler. When enabled, the image filtering scheme accounts for a degree of anisotropy.

The particular scheme for anisotropic texture filtering is implementation dependent. Implementations should consider the *`magFilter`*, *`minFilter`* and *`mipmapMode`* of the sampler to control the specifics of the anisotropic filtering scheme used. In addition, implementations should consider *`minLod`* and *`maxLod`* of the sampler.

The following describes one particular approach to implementing anisotropic filtering for the 2D Image case, implementations may choose other methods:

Given a *`magFilter`*, *`minFilter`* of LINEAR and a *`mipmapMode`* of NEAREST,

Instead of a single isotropic sample, N isotropic samples are be sampled within the image footprint of the image level d to approximate an anisotropic filter. The sum $\tau_{2Daniso}$ is defined using the single isotropic τ_{2D}(u,v) at level d.

$$\tau_{2Daniso} = \frac{1}{N}\sum_{i=1}^{N} \tau_{2D}\left(u\left(x-\frac{1}{2}+\frac{i}{N+1}, y\right), \left(v\left(x-\frac{1}{2}+\frac{i}{N+1}\right), y\right)\right), \qquad \text{when } \rho_x > \rho_y$$

$$\tau_{2Daniso} = \frac{1}{N}\sum_{i=1}^{N} \tau_{2D}\left(u\left(x, y-\frac{1}{2}+\frac{i}{N+1}\right), \left(v\left(x, y-\frac{1}{2}+\frac{i}{N+1}\right)\right)\right), \qquad \text{when } \rho_y \geq \rho_x$$

14.9 Image Operation Steps

Each step described in this chapter is performed by a subset of the image instructions:

- Texel Input Validation Operations, Format Conversion, Texel Replacement, Conversion to RGBA, and Component Swizzle: Performed by all instructions except **`OpImageWrite`**.
- Depth Comparison: Performed by **`OpImage`*****`Dref`** instructions.
- All Texel output operations: Performed by **`OpImageWrite`**.
- Projection: Performed by all **`OpImage`*****`Proj`** instructions.
- Derivative Image Operations, Cube Map Operations, Scale Factor Operation, Level-of-Detail Operation and Image Level(s) Selection, and Texel Anisotropic Filtering: Performed by all **`OpImageSample`*** and **`OpImageSparseSample`*** instructions.
- (s,t,r,q,a) to (u,v,w,a) Transformation, Wrapping, and (u,v,w,a) to (i,j,k,l,n) Transformation And Array Layer Selection: Performed by all **`OpImageSample`**, **`OpImageSparseSample`**, and **`OpImage`*****`Gather`** instructions.
- Texel Gathering: Performed by **`OpImage`*****`Gather`** instructions.
- Texel Filtering: Performed by all **`OpImageSample`*** and **`OpImageSparseSample`*** instructions.

Chapter 15

Queries

Queries provide a mechanism to return information about the processing of a sequence of Vulkan commands. Queries are asynchronous, and as such, their results are not returned immediately. Instead, their results, and their availability status, are stored in Query Pool entries. The state in these query entries can be read back on the host or copied to a buffer object on the device.

The supported query types are Occlusion Queries, Pipeline Statistics Queries, and Timestamp Queries.

15.1 Query Pools

Queries are managed using *query pool* objects. Each query pool is a collection of a specific number of queries of a particular type.

To create a query pool, call:

```
VkResult vkCreateQueryPool(
    VkDevice                                    device,
    const VkQueryPoolCreateInfo*                pCreateInfo,
    const VkAllocationCallbacks*                pAllocator,
    VkQueryPool*                                pQueryPool);
```

`device` is the device to be used to create the query pool. The created pool is returned in `pQueryPool`. `pCreateInfo` is a pointer to an instance of the VkQueryPoolCreateInfo structure containing the number and type of queries to be managed by the pool. `pAllocator` controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- `device` must be a valid VkDevice handle
- `pCreateInfo` must be a pointer to a valid VkQueryPoolCreateInfo structure
- If `pAllocator` is not `NULL`, `pAllocator` must be a pointer to a valid VkAllocationCallbacks structure
- `pQueryPool` must be a pointer to a VkQueryPool handle

The definition of VkQueryPoolCreateInfo is:

```
typedef struct VkQueryPoolCreateInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    VkQueryPoolCreateFlags                     flags;
    VkQueryType                                queryType;
    uint32_t                                   entryCount;
    VkQueryPipelineStatisticFlags              pipelineStatistics;
} VkQueryPoolCreateInfo;
```

The members of VkQueryPoolCreateInfo have the following meanings:

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *queryType* is the type of queries managed by the pool, and must be one of the values

  ```
  typedef enum VkQueryType {
      VK_QUERY_TYPE_OCCLUSION = 0,
      VK_QUERY_TYPE_PIPELINE_STATISTICS = 1,
      VK_QUERY_TYPE_TIMESTAMP = 2,
  } VkQueryType;
  ```

- *entryCount* is the number of query entries managed by the pool.
- *pipelineStatistics* is a bitmask indicating which counters will be returned in queries on the new pool, as described below in Section 15.4. *pipelineStatistics* is ignored if *queryType* is not VK_QUERY_TYPE_PIPELINE_STATISTICS.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_QUERY_POOL_CREATE_INFO
- *pNext* must be NULL
- *flags* must be 0
- *queryType* must be a valid VkQueryType value
- If the pipeline statistics queries feature is not enabled, *queryType* must not be VK_QUERY_TYPE_PIPELINE_STATISTICS
- If *queryType* is VK_QUERY_TYPE_PIPELINE_STATISTICS, *pipelineStatistics* must be a valid combination of VkQueryPipelineStatisticFlagBits values

To destroy a query pool, call:

```
void vkDestroyQueryPool(
    VkDevice                                    device,
    VkQueryPool                                 queryPool,
    const VkAllocationCallbacks*                pAllocator);
```

device is the device to be used to destroy the query pool. *queryPool* is the handle of the query pool to destroy. *pAllocator* controls host memory allocation as described in the Memory Allocation chapter.

Valid Usage

- *device* must be a valid VkDevice handle
- If *queryPool* is not VK_NULL_HANDLE, *queryPool* must be a valid VkQueryPool handle
- If *pAllocator* is not `NULL`, *pAllocator* must be a pointer to a valid VkAllocationCallbacks structure
- If *queryPool* is a valid handle, it must have been created, allocated or retrieved from *device*
- Each of *device* and *queryPool* that are valid handles must have been created, allocated or retrieved from the same VkPhysicalDevice
- All submitted commands that refer to *queryPool* must have completed execution
- If VkAllocationCallbacks were provided when *queryPool* was created, a compatible set of callbacks must be provided here
- If no VkAllocationCallbacks were provided when *queryPool* was created, *pAllocator* must be `NULL`

Host Synchronization

- Host access to *queryPool* must be externally synchronized

15.2 Query Operation

The operation of queries is controlled by the commands `vkCmdBeginQuery`, `vkCmdEndQuery`, `vkCmdResetQueryPool`, `vkCmdCopyQueryPoolResults`, and `vkCmdWriteTimestamp`.

In order for a VkCommandBuffer to record query management commands, the queue family for which its VkCommandPool was created must support the appropriate type of operations (graphics, compute) suitable for the query type of a given query pool.

Each entry in a query pool has a status that is either *unavailable* or *available*, and also has state to store the numerical results of a query of the type requested when the query pool was created. Resetting a query entry via `vkCmdResetQueryPool` sets the status to *unavailable* and makes the numerical results undefined. Performing a query with `vkCmdBeginQuery` and `vkCmdEndQuery` changes the status to *available* when the query completes, and updates the numerical results. Both the availability status and numerical results are retrieved by calling either `vkGetQueryPoolResults` or `vkCmdCopyQueryPoolResults`.

All query commands execute in order and are guaranteed to see the effects of each other's memory accesses, with one significant exception: **`vkCmdCopyQueryPoolResults`** may execute before the results of **`vkCmdEndQuery`** are *available*. However, if VK_QUERY_RESULT_WAIT_BIT is used, then **`vkCmdCopyQueryPoolResults`** must reflect the result of any previously executed queries. Other sequences of commands, such as **`vkCmdResetQueryPool`** followed by **`vkCmdBeginQuery`**, must make the effects of the first command visible to the second command.

After query pool creation, each entry is in an undefined state and must be reset prior to use. Entries must also be reset between uses. Executing a query on a entry that has not been reset will result in undefined behavior.

To reset a range of entries in a query pool, call:

```
void vkCmdResetQueryPool(
    VkCommandBuffer                             commandBuffer,
    VkQueryPool                                 queryPool,
    uint32_t                                    firstQuery,
    uint32_t                                    queryCount);
```

`commandBuffer` is the handle of the command buffer into which this command will be recorded. *`queryPool`* is the handle of the query pool managing the query entries being reset. *`firstQuery`* and *`queryCount`* are the initial entry index and number of entries to reset, respectively. When executed on a queue, this command sets the status of each indicated entry to *unavailable*.

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`queryPool`* must be a valid VkQueryPool handle
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support graphics or compute operations
- This command must only be called outside of a render pass instance
- Each of *`commandBuffer`* and *`queryPool`* must have been created, allocated or retrieved from the same VkDevice
- *`firstQuery`* must be less than the number of query entries in *`queryPool`*
- The sum of *`firstQuery`* and *`queryCount`* must be less than or equal to the number of query entries in *`queryPool`*

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Once query entries are reset and ready for use, query commands can be issued to a command buffer. Occlusion queries and pipeline statistics queries count events - drawn samples and pipeline stage invocations, respectively - resulting from commands that are recorded between a **vkCmdBeginQuery** command and a **vkCmdEndQuery** command within a specified command buffer, effectively scoping a set of drawing and/or compute commands. Timestamp queries write timestamps to query pool entries.

A query must begin and end in the same command buffer, although if it is a primary command buffer, and the inherited queries feature is enabled, it can execute secondary command buffers during the query. For a secondary command buffer to be executed while a query is active, it must set the *occlusionQueryEnable*, *queryFlags*, and/or *pipelineStatistics* members of `VkCommandBufferBeginInfo` to conservative values, as described in that section. A query must either begin and end inside the same subpass of a render pass instance, or must both begin and end outside of a render pass instance (i.e. contain entire render pass instances).

Begin a query by calling:

```
void vkCmdBeginQuery(
    VkCommandBuffer                             commandBuffer,
    VkQueryPool                                 queryPool,
    uint32_t                                    entry,
    VkQueryControlFlags                         flags);
```

with *commandBuffer* set to the VkCommandBuffer handle of the command buffer into which this command will be recorded.

flags is a bitmask indicating constraints on the types of queries that can be performed. Valid bits in *flags* include:

```
typedef enum VkQueryControlFlagBits {
    VK_QUERY_CONTROL_PRECISE_BIT = 0x00000001,
} VkQueryControlFlagBits;
```

queryPool is the VkQueryPool handle of the query pool that will manage the results of the query, and *entry* is the query index within the query pool that will contain the results. If the *queryType* of the pool is VK_QUERY_TYPE_OCCLUSION and *flags* contains VK_QUERY_CONTROL_PRECISE_BIT, an implementation must use precise methods to collect results. This is described in more detail in Occlusion Queries.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *queryPool* must be a valid VkQueryPool handle
- *flags* must be a valid combination of `VkQueryControlFlagBits` values
- *commandBuffer* must be in the recording state

- The VkCommandPool that *commandBuffer* was allocated from must support graphics or compute operations
- Each of *commandBuffer* and *queryPool* must have been created, allocated or retrieved from the same VkDevice
- The query identified by *queryPool* and *entry* must currently not be active
- The query identified by *queryPool* and *entry* must be unavailable
- If the precise occlusion queries feature is not enabled, or the *queryType* used to create *queryPool* was not VK_QUERY_TYPE_OCCLUSION, *flags* must not contain VK_QUERY_CONTROL_PRECISE_BIT
- *queryPool* must have been created with a *queryType* that differs from that of any other queries that have been made active, and are currently still active within *commandBuffer*
- *entry* must be less than the number of entries in *queryPool*
- If the *queryType* used to create *queryPool* was VK_QUERY_TYPE_OCCLUSION, the VkCommandPool that *commandBuffer* was created from must support graphics operations
- If the *queryType* used to create *queryPool* was VK_QUERY_TYPE_PIPELINE_STATISTICS and any of the *pipelineStatistics* indicate graphics operations, the VkCommandPool that *commandBuffer* was created from must support graphics operations
- If the *queryType* used to create *queryPool* was VK_QUERY_TYPE_PIPELINE_STATISTICS and any of the *pipelineStatistics* indicate compute operations, the VkCommandPool that *commandBuffer* was created from must support compute operations

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

After beginning a query, that query is considered active within the command buffer it was called in until that same query is ended. Queries active in a primary command buffer when secondary command buffers are executed are considered active for those secondary command buffers.

After the set of desired draw or dispatch commands, end a query by calling:

```
void vkCmdEndQuery(
    VkCommandBuffer                             commandBuffer,
    VkQueryPool                                 queryPool,
    uint32_t                                    entry);
```

with *commandBuffer* set to the VkCommandBuffer handle of the command buffer into which this command will be recorded. *queryPool* is the VkQueryPool handle of the query pool that is managing the results of the query, and *entry* is the entry index where the result is stored.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *queryPool* must be a valid VkQueryPool handle
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics or compute operations
- Each of *commandBuffer* and *queryPool* must have been created, allocated or retrieved from the same VkDevice
- The query identified by *queryPool* and *entry* must currently be active
- *entry* must be less than the number of entries in *queryPool*

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

When a query ends, the query entry's status is set to *available*, and the numerical results can be retrieved by `vkGetQueryPoolResults` and `vkCmdCopyQueryPoolResults`.

An application can retrieve results either by requesting they be written into application-provided memory, or by requesting they be copied into a VkBuffer. In either case, the layout in memory is defined as follows:

- The first query entry's result is written starting at the first byte requested by the command, and each subsequent entry's result begins *stride* bytes later.
- Each query entry's result is a tightly packed array of unsigned integers, either 32- or 64-bits as requested by the command, storing the numerical results and, if requested, the availability status.
- If VK_QUERY_RESULT_WITH_AVAILABILITY_BIT is used, the final element of each entry's result is an integer indicating whether the entry's result is *available*, with any non-zero value indicating that it is *available*.
- Occlusion queries write one integer value - the number of samples passed. Pipeline statistics queries write one integer value for each bit that is enabled in the *pipelineStatistics* when the pool is created, and the statistics values are written in bit order starting from the least significant bit. Timestamps write one integer value.
- If more than one query is retrieved and *stride* is not at least as large as the size of the array of integers, the values written to memory are undefined.

To retrieve status and results for a set of query entries, call:

```
VkResult vkGetQueryPoolResults(
    VkDevice                                    device,
    VkQueryPool                                 queryPool,
    uint32_t                                    firstQuery,
    uint32_t                                    queryCount,
    size_t                                      dataSize,
    void*                                       pData,
    VkDeviceSize                                stride,
    VkQueryResultFlags                          flags);
```

device is the device on which *queryPool* was created. *queryPool* is the query pool handle managing the query entries containing the desired results. *firstQuery* and *queryCount* are the initial query entry index and number of query entries defining a range of queries, respectively. *pData* is a pointer to user-allocated memory where the results will be written, and is of size *dataSize* bytes. *flags* is a bitfield specifying how and when results are returned.

Valid Usage

- *device* must be a valid VkDevice handle
- *queryPool* must be a valid VkQueryPool handle
- *pData* must be a pointer to an array of *dataSize* bytes
- *flags* must be a valid combination of `VkQueryResultFlagBits` values
- The value of *dataSize* must be greater than `0`
- *queryPool* must have been created, allocated or retrieved from *device*
- Each of *device* and *queryPool* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *firstQuery* must be less than the number of query entries in *queryPool*
- The sum of *firstQuery* and *queryCount* must be less than or equal to the number of query entries in *queryPool*
- *dataSize* must be large enough to contain the result of each entry's query, as described here
- If the *queryType* used to create *queryPool* was VK_QUERY_TYPE_TIMESTAMP, *flags* must not contain VK_QUERY_RESULT_PARTIAL_BIT

Valid bits in *flags* include:

```
typedef enum VkQueryResultFlagBits {
    VK_QUERY_RESULT_64_BIT = 0x00000001,
    VK_QUERY_RESULT_WAIT_BIT = 0x00000002,
    VK_QUERY_RESULT_WITH_AVAILABILITY_BIT = 0x00000004,
    VK_QUERY_RESULT_PARTIAL_BIT = 0x00000008,
} VkQueryResultFlagBits;
```

These bits have the following meanings:

- VK_QUERY_RESULT_64_BIT indicates the results will be written as an array of 64-bit unsigned integer values. If this bit is not set, the results will be written as an array of 32-bit unsigned integer values.

- VK_QUERY_RESULT_WAIT_BIT indicates that Vulkan will wait for each query entry's status to become *available* before retrieving the query results.
- VK_QUERY_RESULT_WITH_AVAILABILITY_BIT indicates that the availability status accompanies the results.
- VK_QUERY_RESULT_PARTIAL_BIT indicates that returning partial results is acceptable.

If no bits are set in *`flags`*, and all requested query entries are in the *available* state, results are written as an array of 32-bit unsigned integer values. The behavior when not all query entries are *available*, is described below.

If VK_QUERY_RESULT_64_BIT is not set and the result overflows a 32-bit value, the value may either wrap or saturate. Similarly, if VK_QUERY_RESULT_64_BIT is set and the result overflows a 64-bit value, the value may either wrap or saturate.

If VK_QUERY_RESULT_WAIT_BIT is set, Vulkan will wait for each entry to be in the *available* state before retrieving the numerical results for that entry. In this case, **`vkGetQueryPoolResults`** is guaranteed to complete and return VK_SUCCESS if the query entries become *available* in a finite time (i.e. if they have been issued and not reset). If queries will never complete (e.g. due to being reset but not issued), then **`vkGetQueryPoolResults`** may not return in finite time.

If VK_QUERY_RESULT_WAIT_BIT and VK_QUERY_RESULT_PARTIAL_BIT are both not set and one or more of the requested query entries are in the *unavailable* state, then nothing is written to *`pData`* for the *unavailable* entries, and **`vkGetQueryPoolResults`** returns VK_NOT_READY.

Note
Applications must take care to ensure that use of the VK_QUERY_RESULT_WAIT_BIT bit has the desired effect.
For example, if a query entry has been used previously and a command buffer records the commands **`vkCmdResetQueryPool`**, **`vkCmdBeginQuery`**, and **`vkCmdEndQuery`** for that entry, then the query pool entry will remain in the *available* state until the **`vkCmdResetQueryPool`** command executes on a queue. Applications can use fences or events to ensure that an entry has already been reset before checking for its results or availability status. Otherwise, a stale value could be returned from a previous use of the query entry.
The above also applies when VK_QUERY_RESULT_WAIT_BIT is used in combination with VK_QUERY_RESULT_WITH_AVAILABILITY_BIT. In this case, the returned availability status may reflect the result of a previous use of the query unless the **`vkCmdResetQueryPool`** command has been executed since the last use of the query.

Note
Applications can double-buffer query pool usage, with a pool per frame, and reset query entries at the end of the frame in which they are read.

If VK_QUERY_RESULT_PARTIAL_BIT is set, VK_QUERY_RESULT_WAIT_BIT is not set, and the query entry's status is *unavailable*, an intermediate value between zero and the final result value may be returned.

VK_QUERY_RESULT_PARTIAL_BIT must not be used if the pool's *`queryType`* is VK_QUERY_TYPE_TIMESTAMP.

If VK_QUERY_RESULT_WITH_AVAILABILITY_BIT is set, the final integer value written for each query is non-zero if the query entry's status was *available* or zero if the status was *unavailable*. When VK_QUERY_

RESULT_WITH_AVAILABILITY_BIT is used, implementations must guarantee that if they return a non-zero availability value then the numerical results must be valid, assuming the results are not reset by a subsequent command.

Note
Satisfying this guarantee may require careful ordering by the application, e.g. to read the availability status before reading the results.

To copy query statuses and numerical results directly to buffer memory, call:

```
void vkCmdCopyQueryPoolResults(
    VkCommandBuffer                             commandBuffer,
    VkQueryPool                                 queryPool,
    uint32_t                                    firstQuery,
    uint32_t                                    queryCount,
    VkBuffer                                    dstBuffer,
    VkDeviceSize                                dstOffset,
    VkDeviceSize                                stride,
    VkQueryResultFlags                          flags);
```

commandBuffer is the VkCommandBuffer handle of the command buffer into which this command will be recorded. *queryPool* is the handle of the VkQueryPool managing the query entries containing the desired results. *firstQuery* and *queryCount* are the initial query entry index and the number of query entries defining a range of queries, respectively. *dstBuffer* is the VkBuffer handle that will receive the results of the copy command. *dstOffset* is the offset into *dstBuffer*, and *stride* is the stride in bytes between results for individual query entries within *dstBuffer*. The size of the backing memory for *dstBuffer* is determined as described above (see **vkGetQueryPoolResults**).

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *queryPool* must be a valid VkQueryPool handle
- *dstBuffer* must be a valid VkBuffer handle
- *flags* must be a valid combination of `VkQueryResultFlagBits` values
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics or compute operations
- This command must only be called outside of a render pass instance
- Each of *commandBuffer*, *queryPool* and *dstBuffer* must have been created, allocated or retrieved from the same VkDevice
- *firstQuery* must be less than the number of query entries in *queryPool*
- The sum of *firstQuery* and *queryCount* must be less than or equal to the number of query entries in *queryPool*

- *dstBuffer* must have enough storage, from *dstOffset*, to contain the result of each entry's query, as described here
- If the *queryType* used to create *queryPool* was VK_QUERY_TYPE_TIMESTAMP, *flags* must not contain VK_QUERY_RESULT_PARTIAL_BIT

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

vkCmdCopyQueryPoolResults is guaranteed to see the effect of previous uses of **vkCmdResetQueryPool** in the same queue, without any additional synchronization. Thus, the results will always reflect the most recent use of the query entry.

flags has the same possible values described above (see VkQueryResultFlagBits), but the different style of execution causes some subtle behavioral differences. Because **vkCmdCopyQueryPoolResults** executes in-order with respect to other query commands, there is less ambiguity about which use of a query entry is being requested.

If no bits are set in *flags*, results for all requested query entries in the *available* state are written as 32-bit unsigned integer values, and nothing is written for entries in the *unavailable* state.

If VK_QUERY_RESULT_64_BIT is set, the results are written as an array of 64-bit unsigned integer values as described for vkGetQueryPoolResults.

If VK_QUERY_RESULT_WAIT_BIT is set, the implementation will wait for each entry's status to be in the *available* state before retrieving the numerical results for that entry. This is guaranteed to reflect the most recent use of the query entry on the same queue, assuming that the entry is not being simultaneously used by other queues. If the query does not become *available* in a finite amount of time (e.g. due to not issuing a query since the last reset), a VK_ERROR_DEVICE_LOST error may occur.

Similarly, if VK_QUERY_RESULT_WITH_AVAILABILITY_BIT is set and VK_QUERY_RESULT_WAIT_BIT is not set, the availability is guaranteed to reflect the most recent use of the query entry on the same queue, assuming that the entry is not being simultaneously used by other queues. As with **vkGetQueryPoolResults**, implementations must guarantee that if they return a non-zero availability value, then the numerical results are valid.

If VK_QUERY_RESULT_PARTIAL_BIT is set, VK_QUERY_RESULT_WAIT_BIT is not set, and the query entry's status is *unavailable*, an intermediate value between zero and the final result value may be returned.

VK_QUERY_RESULT_PARTIAL_BIT must not be used if the pool's *queryType* is VK_QUERY_TYPE_TIMESTAMP.

vkCmdCopyQueryPoolResults is considered to be a transfer operation, and its writes to buffer memory must be synchronized using VK_PIPELINE_STAGE_TRANSFER_BIT and VK_ACCESS_TRANSFER_WRITE_BIT before using the results.

Rendering operations such as clears, MSAA resolves, attachment load/store operations, and blits may or may not count towards the results of queries. This behavior is implementation-dependent and may vary depending on the path used within an implementation. For example, some implementations have several types of clears, some of which may include vertices and some not.

15.3 Occlusion Queries

Occlusion queries track the number of samples that pass the per-fragment tests for a set of drawing commands. As such, occlusion queries are only available on queue families supporting graphics operations. The application can then use these results to inform future rendering decisions. An occlusion query is started and finished by calling **vkCmdBeginQuery** and **vkCmdEndQuery**, respectively. When an occlusion query starts, the count of passing samples always begins at zero. For each drawing command, the count is incremented as described in Sample Counting. If *flags* does not contains VK_QUERY_CONTROL_PRECISE_BIT and at least one sample passed, the implementation will return between one and the number of passing samples.

Note
Not setting VK_QUERY_CONTROL_PRECISE_BIT mode will be more efficient on some implementations, and should be used where it is sufficient to know whether any samples were visible.

When an occlusion query finishes, the query result for that entry is marked as *available*. The application can then either copy the result to a buffer (via **vkCmdCopyQueryPoolResults**) or request it be put into host memory (via **vkGetQueryPoolResults**).

Note
If occluding geometry is not drawn first, samples can pass the depth test, but still not be visible in a final image.

15.4 Pipeline Statistics Queries

Pipeline statistics queries allow the application to sample a specified set of VkPipeline counters. These counters are accumulated by Vulkan for a set of either draw or dispatch commands while a pipeline statistics query is active. As such, pipeline statistics queries are available on queue families supporting either graphics or compute operations. Further, the availability of pipeline statistics queries is indicated by the *pipelineStatisticsQuery* member of the VkPhysicalDeviceFeatures object (see **vkGetPhysicalDeviceFeatures** and **vkCreateDevice** for detecting and requesting this query type on a VkDevice).

A pipeline statistics query is started and finished by calling **vkCmdBeginQuery** and **vkCmdEndQuery**, respectively. When a pipeline statistics query starts, all statistics counters are set to zero. While the query is active, the pipeline type determines which set of statistics are available, but these must be configured on the query pool when it is created. If a statistic counter is issued on a command buffer that does not support the corresponding operation, the value of that counter is undefined after the query has completed. At least one statistic counter relevant to the operations supported on the recording command buffer must be enabled.

pipelineStatisticsQuery is a bitmask indicating different possible pipeline statistics.

Valid bits in *flags* include:

```
typedef enum VkQueryPipelineStatisticFlagBits {
    VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT = 0x00000001,
    VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT = 0x00000002,
    VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT = 0x00000004,
    VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT = 0x00000008,
    VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT = 0x00000010,
    VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT = 0x00000020,
    VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT = 0x00000040,
    VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT = 0x00000080,
    VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT = 0x00000100 ↩
        ,
    VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT = 0 ↩
        x00000200,
    VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT = 0x00000400,
} VkQueryPipelineStatisticFlagBits;
```

These bits have the following meanings:

- If VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_VERTICES_BIT is set, queries managed by the pool will count the number of vertices processed by the input assembly stage. Vertices corresponding to incomplete primitives may or may not contribute to the count.
- If VK_QUERY_PIPELINE_STATISTIC_INPUT_ASSEMBLY_PRIMITIVES_BIT is set, queries managed by the pool will count the number of primitives processed by the input assembly stage. If primitive restart is enabled, restarting the primitive topology has no effect on the count. Incomplete primitives may or may not be counted.
- If VK_QUERY_PIPELINE_STATISTIC_VERTEX_SHADER_INVOCATIONS_BIT is set, queries managed by the pool will count the number of vertex shader invocations. This counter's value is incremented each time a vertex shader is invoked.
- If VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_INVOCATIONS_BIT is set, queries managed by the pool will count the number of geometry shader invocations. This counter's value is incremented each time a geometry shader is invoked. In case of of instanced geometry shaders, the geometry shader invocations count is incremented for each separate instanced invocation.
- If VK_QUERY_PIPELINE_STATISTIC_GEOMETRY_SHADER_PRIMITIVES_BIT is set, queries managed by the pool will count the number of primitives generated by geometry shader invocations. The counter's value is incremented each time the geometry shader emits a primitive. Restarting primitive topology using the SPIR-V instructions **`OpEndPrimitive`** or **`OpEndStreamPrimitive`** has no effect on the geometry shader output primitives count.
- If VK_QUERY_PIPELINE_STATISTIC_CLIPPING_INVOCATIONS_BIT is set, queries managed by the pool will count the number of primitives processed by the Primitive Clipping stage of the pipeline. The counter's value is incremented each time a primitive reaches the primitive clipping stage.
- If VK_QUERY_PIPELINE_STATISTIC_CLIPPING_PRIMITIVES_BIT is set, queries managed by the pool will count the number of primitives output by the Primitive Clipping stage of the pipeline. The counter's value is incremented each time a primitive passes the primitive clipping stage. The actual number of primitives output by the primitive clipping stage for a particular input primitive is implementation-dependent but must satisfy the following conditions:
  - If at least one vertex of the input primitive lies inside the clipping volume, the counter is incremented by one or more.
  - Otherwise, the counter is incremented by zero or more.
- If VK_QUERY_PIPELINE_STATISTIC_FRAGMENT_SHADER_INVOCATIONS_BIT is set, queries managed by the pool will count the number of fragment shader invocations. The counter's value is incremented each time the fragment shader is invoked.
- If VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_CONTROL_SHADER_PATCHES_BIT is set, queries managed by the pool will count the number of patches processed by the tessellation control shader. The counter's value is incremented each time the tessellation control shader is invoked.

- If VK_QUERY_PIPELINE_STATISTIC_TESSELLATION_EVALUATION_SHADER_INVOCATIONS_BIT is set, queries managed by the pool will count the number of invocations of the tessellation evaluation shader. The counter's value is incremented each time the tessellation evaluation shader is invoked.
- If VK_QUERY_PIPELINE_STATISTIC_COMPUTE_SHADER_INVOCATIONS_BIT is set, queries managed by the pool will count the number of compute shader invocations. The counter's value is incremented every time the compute shader is invoked. Implementations may skip the execution of certain compute shader invocations or execute additional compute shader invocations for implementation-dependent reasons as long as the results of rendering otherwise remain unchanged.

These values are intended to measure relative statistics on one implementation. Various device architectures will count these values differently. Any or all counters may be affected by the issues described in Query Operation.

Note
For example, tile-based rendering devices may need to replay the scene multiple times, affecting some of the counts.

If a pipeline has *`rasterizerDiscardEnable`* enabled, implementations may discard primitives after the final vertex processing stage. As a result, if *`rasterizerDiscardEnable`* is enabled, the clipping input and output primitives counters may not be incremented.

When a pipeline statistics query finishes, the query result for that entry is marked as *available*. The application can copy the result to a buffer (via **`vkCmdCopyQueryPoolResults`**), or request it be put into host memory (via **`vkGetQueryPoolResults`**).

15.5 Timestamp Queries

Timestamps provide applications with a mechanism for timing the execution of commands. A timestamp is a 64-bit value generated by the VkPhysicalDevice. Unlike other queries, timestamps do not operate over a range, and so do not use `vkCmdBeginQuery` or `vkCmdEndQuery`. The mechanism is built around a set of commands that allow the application to tell the VkPhysicalDevice to write timestamp values to a query pool and then either read timestamp values on the host (using `vkGetQueryPoolResults`) or copy timestamp values to a VkBuffer (using `vkCmdCopyQueryPoolResults`). The application can then compute differences between timestamps to determine execution time.

The number of valid bits in a timestamp value is determined by the VkQueueFamilyProperties::*`timestampValidBits`* property of the queue on which the timestamp is written. Timestamps are supported on any queue which reports a non-zero value for *`timestampValidBits`* via `vkGetPhysicalDeviceQueueFamilyProperties`. If the timestampComputeAndGraphics limit is VK_ TRUE, timestamps are supported by every queue family that supports either graphics or compute operations (see `VkQueueFamilyProperties`).

The number of nanoseconds it takes for a timestamp value to be incremented by 1 can be obtained from VkPhysicalDeviceLimits::*`timestampPeriod`* after a call to **`vkGetPhysicalDeviceProperties`**.

A timestamp is requested by calling:

```
void vkCmdWriteTimestamp(
    VkCommandBuffer                             commandBuffer,
    VkPipelineStageFlagBits                     pipelineStage,
    VkQueryPool                                 queryPool,
    uint32_t                                    entry);
```

with *`commandBuffer`* set to the handle of a VkCommandBuffer into which the command will be recorded. *`pipelineStage`* is a stage of the pipeline, specified using the `VkPipelineStageFlagBits` type. The command latches the value of the timer when all previous commands have completed executing as far as the indicated stage of the pipeline, and writes the timestamp value to memory. When the timestamp value is written, the availability status of the query pool entry is set to *available*.

Note
If an implementation is unable to detect completion and latch the timer at any specific stage of the pipeline, it may instead do so at any logically later stage.

`queryPool` is the VkQueryPool handle of the query pool that will manage the timestamp, and *`entry`* is the query entry within the query pool that will contain the timestamp. **`vkCmdCopyQueryPoolResults`** can then be called to copy the timestamp value from the query pool into buffer memory, with ordering and synchronization behavior equivalent to how other queries operate. Timestamp values can also be retrieved from the query pool using **`vkGetQueryPoolResults`**. As with other queries, the query pool entry must be reset using **`vkCmdResetQueryPool`** before requesting the timestamp value be written to it.

While **`vkCmdWriteTimestamp`** can be called inside or outside of a render pass instance, **`vkCmdCopyQueryPoolResults`** must only be called outside of a render pass instance.

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`pipelineStage`* must be a valid `VkPipelineStageFlagBits` value
- *`queryPool`* must be a valid VkQueryPool handle
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support graphics or compute operations
- Each of *`commandBuffer`* and *`queryPool`* must have been created, allocated or retrieved from the same VkDevice
- The query identified by *`queryPool`* and *`entry`* must be *unavailable*
- The command pool's queue family must support a non-zero value of *`timestampValidBits`*

Host Synchronization

- Host access to *`commandBuffer`* must be externally synchronized

Chapter 16

Clear Commands

16.1 Clearing Images Outside A Render Pass Instance

Color and depth images can be cleared outside a render pass instance using **`vkCmdClearColorImage`** or **`vkCmdClearDepthStencilImage`**, respectively. These commands are only allowed outside of a render pass instance.

To clear a color image, call:

```
void vkCmdClearColorImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     image,
    VkImageLayout                               imageLayout,
    const VkClearColorValue*                    pColor,
    uint32_t                                    rangeCount,
    const VkImageSubresourceRange*              pRanges);
```

`image` is a handle to the image to be cleared. *`imageLayout`* specifies the current layout of the image subresource ranges to be cleared and must be VK_IMAGE_LAYOUT_GENERAL or VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL. *`pColor`* is a pointer to a VkClearColorValue structure that contains the values the image subresource ranges will be cleared to (see Section 16.3 below). **`vkCmdClearColorImage`** can be used to clear multiple ranges of mipmap levels, array layers, and aspects. *`pRanges`* points to an array of VkImageSubresourceRange structures that describe these ranges. The *`aspectMask`* of all subresource ranges must only include VK_IMAGE_ASPECT_COLOR_BIT. *`rangeCount`* describes the number of elements in *`pRanges`*. The VkImageSubresourceRange structure is defined in Image Views.

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`image`* must be a valid VkImage handle
- *`imageLayout`* must be a valid `VkImageLayout` value

- `pColor` must be a pointer to a valid VkClearColorValue union
- `pRanges` must be a pointer to an array of `rangeCount` valid VkImageSubresourceRange structures
- `commandBuffer` must be in the recording state
- The VkCommandPool that `commandBuffer` was allocated from must support graphics or compute operations
- This command must only be called outside of a render pass instance
- The value of `rangeCount` must be greater than `0`
- Each of `commandBuffer` and `image` must have been created, allocated or retrieved from the same VkDevice
- `image` must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- `imageLayout` must specify the layout of the subresource ranges of `image` specified in `pRanges` at the time this command is executed on a VkDevice
- `imageLayout` must be either of VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL
- The image range of any given element of `pRanges` must be a subresource range that is contained within `image`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

To clear a depth, stencil, or depth-stencil image, call:

```
void vkCmdClearDepthStencilImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     image,
    VkImageLayout                               imageLayout,
    const VkClearDepthStencilValue*             pDepthStencil,
    uint32_t                                    rangeCount,
    const VkImageSubresourceRange*              pRanges);
```

`image`, `imageLayout`, `rangeCount`, and `pRanges` have the same meaning and constraints as for **`vkCmdClearColorImage`**. The `aspectMask` of each subresource range in `pRanges` can include VK_IMAGE_ASPECT_DEPTH_BIT for depth images, VK_IMAGE_ASPECT_STENCIL_BIT for stencil images or both bits for depth-stencil images. `pDepthStencil` is a pointer to a VkClearDepthStencilValue structure that contains the values the image subresource ranges will be cleared to (see Section 16.3 below).

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *image* must be a valid VkImage handle
- *imageLayout* must be a valid `VkImageLayout` value
- *pDepthStencil* must be a pointer to a valid VkClearDepthStencilValue structure
- *pRanges* must be a pointer to an array of *rangeCount* valid VkImageSubresourceRange structures
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- This command must only be called outside of a render pass instance
- The value of *rangeCount* must be greater than `0`
- Each of *commandBuffer* and *image* must have been created, allocated or retrieved from the same VkDevice
- *image* must have been created with *VK_IMAGE_USAGE_TRANSFER_DST_BIT* usage flag
- *imageLayout* must specify the layout of the subresource ranges of *image* specified in *pRanges* at the time this command is executed on a VkDevice
- *imageLayout* must be either of VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL
- The image range of any given element of *pRanges* must be a subresource range that is contained within *image*

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Clears outside render pass instances are treated as transfer operations for the purposes of memory barriers.

16.2 Clearing Images Inside A Render Pass Instance

To clear color and depth/stencil attachments inside a render pass instance, call:

```
void vkCmdClearAttachments(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    attachmentCount,
    const VkClearAttachment*                    pAttachments,
    uint32_t                                    rectCount,
    const VkClearRect*                          pRects);
```

This command must be called only inside a render pass instance, and implicitly selects the images to clear based on the current attachments and the command parameters.

attachmentCount is the number of entries in the *pAttachments* array where each element defines the attachment to clear and the clear value. VkClearAttachment is defined below.

vkCmdClearAttachments can clear multiple regions of each attachment used in the current subpass of a render pass instance. *pRects* points to an array of VkClearRect structures with *rectCount* elements that describe these regions.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *pAttachments* must be a pointer to an array of *attachmentCount* valid VkClearAttachment structures
- *pRects* must be a pointer to an array of *rectCount* VkClearRect structures
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- This command must only be called inside of a render pass instance
- The value of *attachmentCount* must be greater than 0
- The value of *rectCount* must be greater than 0
- If the *aspectMask* member of any given element of *pAttachments* contains VK_IMAGE_ASPECT_COLOR_BIT, the *colorAttachment* member of those elements must refer to a valid color attachment in the current subpass
- The rectangular region specified by a given element of *pRects* must be contained within the render area of the current render pass instance
- The layers specified by a given element of *pRects* must be contained within every attachment that *pAttachments* refers to

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

The VkClearRect struct is defined as follows:

```
typedef struct VkClearRect {
    VkRect2D                                            rect;
    uint32_t                                            baseArrayLayer;
    uint32_t                                            layerCount;
} VkClearRect;
```

- *rect* is the two-dimensional region to be cleared.
- *baseArrayLayer* is the first layer to be cleared.
- *layerCount* is the number of layers to clear.

The layers $[baseArrayLayer, baseArrayLayer + layerCount)$ counting from the base layer of the attachment image view are cleared.

The VkClearAttachment struct is defined as follows:

```
typedef struct VkClearAttachment {
    VkImageAspectFlags                                  aspectMask;
    uint32_t                                            colorAttachment;
    VkClearValue                                        clearValue;
} VkClearAttachment;
```

- *aspectMask* is a mask selecting the color, depth and/or stencil attachments to be cleared. *aspectMask* can include VK_IMAGE_ASPECT_COLOR_BIT for color images, VK_IMAGE_ASPECT_DEPTH_BIT for depth or depth-stencil images, and VK_IMAGE_ASPECT_STENCIL_BIT for stencil or depth-stencil images.
- *colorAttachment* is only meaningful if VK_IMAGE_ASPECT_COLOR_BIT is set in *aspectMask*, in which case it is an index to the *pColorAttachments* array in the `VkSubpassDescription` structure of the current subpass.
- *clearValue* is the color or depth/stencil value to clear the attachment to, as described in Clear Values below.

No memory barriers are needed between **`vkCmdClearAttachments`** and preceding or subsequent draw or attachment clear commands in the same render pass that use the attachment in the same image layout.

The **`vkCmdClearAttachments`** commands is not affected by the bound pipeline state.

Attachments can also be cleared at the beginning of a render pass instance by setting *loadOp* (or *stencilLoadOp*) of `VkAttachmentDescription` to VK_ATTACHMENT_LOAD_OP_CLEAR, as described for `vkCreateRenderPass`.

Valid Usage

- *aspectMask* must be a valid combination of `VkImageAspectFlagBits` values
- *aspectMask* must not be `0`
- If *aspectMask* includes VK_IMAGE_ASPECT_COLOR_BIT, it must not include VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT
- *aspectMask* must not include VK_IMAGE_ASPECT_METADATA_BIT

16.3 Clear Values

The definition of VkClearColorValue is as follows:

```
typedef union VkClearColorValue {
    float                                       float32[4];
    int32_t                                     int32[4];
    uint32_t                                    uint32[4];
} VkClearColorValue;
```

Color clear values are taken from the VkClearColorValue union based on the format of the image or attachment. Floating point, unorm, snorm, uscaled, packed float, and sRGB images use the *float32* member, unsigned integer formats use the *uint32* member, and signed integer formats use the *int32* member. Floating point values are automatically converted to the format of the image, with the clear value being treated as linear if the image is sRGB.

Unsigned integer values are converted to the format of the image by casting to the integer type with fewer bits. Signed integer values are converted to the format of the image by casting to the smaller type (with negative 32-bit values mapping to negative values in the smaller type). If the integer clear value is not representable in the target type (e.g. would overflow in conversion to that type), the clear value is undefined.

The four array elements of the clear color map to R, G, B, and A components of image formats, in order.

If the image has more than one sample, the same value is written to all samples for any pixels being cleared. The **vkClear*Image** commands do not support compressed image formats.

The definition of VkClearDepthStencilValue is as follows:

```
typedef struct VkClearDepthStencilValue {
    float                                       depth;
    uint32_t                                    stencil;
} VkClearDepthStencilValue;
```

- *depth* is the clear value for the depth attachment. It is a floating-point value which is automatically converted to the image format.
- *stencil* is the clear value for the stencil attachment. It is a 32-bit integer value which is converted to the stencil format by taking the appropriate number of LSBs.

Some parts of the API require either color or depth-stencil clears, depending on the attachment. For this the VkClearValue union is defined as follows:

```
typedef union VkClearValue {
    VkClearColorValue                           color;
    VkClearDepthStencilValue                    depthStencil;
} VkClearValue;
```

- *color* specifies the color image clear values to use when referencing a color image.
- *depthStencil* specifies the depth and stencil clear values to use for depth, stencil or depth-stencil images.

This union is used to define the initial clear values in the VkRenderPassBeginInfo structure.

16.4 Filling Buffers

To clear buffer data, call:

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

`dstBuffer` is a handle to the buffer to be filled. `dstOffset` is the byte offset into the buffer to start filling and must be a multiple of 4. `size` is the number of bytes to fill, and must be either a multiple of 4, or VK_WHOLE_SIZE to fill the range from `offset` to the end of the buffer. The `data` value is the 4-byte word written repeatedly to the buffer to fill `size` bytes of data. Each word is written as a whole, preserving the endianness of the system.

This command is treated as "transfer" operation, for the purposes of synchronization barriers. The VK_BUFFER_USAGE_TRANSFER_DST_BIT must be specified in `usage` of VkBufferCreateInfo in order for the buffer to be compatible with **`vkCmdFillBuffer`**.

Valid Usage

- `commandBuffer` must be a valid VkCommandBuffer handle
- `dstBuffer` must be a valid VkBuffer handle
- `commandBuffer` must be in the recording state
- The VkCommandPool that `commandBuffer` was allocated from must support graphics or compute operations
- This command must only be called outside of a render pass instance
- Each of `commandBuffer` and `dstBuffer` must have been created, allocated or retrieved from the same VkDevice
- If `size` is not equal to VK_WHOLE_SIZE, the sum of `dstOffset` and `size` must be less than or equal to the size of `dstBuffer`
- `dstBuffer` must have been created with `VK_BUFFER_USAGE_TRANSFER_DST_BIT` usage flag
- `dstOffset` must be a multiple of `4`
- If `size` is not equal to VK_WHOLE_SIZE, `size` must be a multiple of `4`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

16.5 Updating Buffers

To update buffer data inline in a command buffer, call:

```
void vkCmdUpdateBuffer(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    dstBuffer,
    VkDeviceSize                                dstOffset,
    VkDeviceSize                                dataSize,
    const uint32_t*                             pData);
```

dstBuffer is a handle to the buffer to be updated. *dstOffset* is the byte offset into the buffer to start updating, and must be a multiple of 4. *dataSize* is the number of bytes to update, and must be a multiple of 4. *pData* is the source data for the buffer update and must be at least *dataSize* bytes in size.

dataSize must be less than or equal to 65536 bytes. For larger updates, applications can use buffer to buffer copies.

The source data is copied from the user pointer to the command buffer when the command is called.

The **vkCmdUpdateBuffer** command is only allowed outside of a render pass. This command is treated as "transfer" operation, for the purposes of synchronization barriers. The VK_BUFFER_USAGE_TRANSFER_DST_BIT must be specified in *usage* of VkBufferCreateInfo in order for the buffer to be compatible with **vkCmdUpdateBuffer**.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *dstBuffer* must be a valid VkBuffer handle
- *pData* must be a pointer to an array of *dataSize*/4 uint32_t values
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support transfer, graphics or compute operations
- This command must only be called outside of a render pass instance
- Each of *commandBuffer* and *dstBuffer* must have been created, allocated or retrieved from the same VkDevice
- The sum of *dstOffset* and *dataSize* must be less than or equal to the size of *dstBuffer*
- *dstBuffer* must have been created with *VK_BUFFER_USAGE_TRANSFER_DST_BIT* usage flag
- The value of *dstOffset* must be a multiple of `4`
- The value of *dataSize* must be greater than `0`
- The value of *dataSize* must be less than `65536`
- The value of *dataSize* must be a multiple of `4`

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Chapter 17

Copy Commands

An application can copy buffer and image data using several methods depending on the type of data transfer. Data can be copied between buffer objects with **`vkCmdCopyBuffer`** and a portion of an image can be copied to another image with **`vkCmdCopyImage`**. Image data can also be copied to and from buffer memory using **`vkCmdCopyImageToBuffer`** and **`vkCmdCopyBufferToImage`**. Image data can be blitted (with or without scaling and filtering) with **`vkCmdBlitImage`**. Multisampled images can be resolved to a non-multisampled image with **`vkCmdResolveImage`**.

17.1 Common Operation

Some rules for valid operation are common to all copy commands:

- Copy commands must be recorded outside of a render pass instance.
- For non-sparse resources, the union of the source regions in a given buffer or image must not overlap the union of the destination regions in the same buffer or image.
- For sparse resources, the set of bytes used by all the source regions must not intersect the set of bytes used by all the destination regions.
- Copy regions must be non-empty.
- Regions must not extend outside the bounds of the buffer or image level, except that regions of compressed images can extend as far as the dimension of the image level rounded up to a complete block.
- Source images must be in either the VK_IMAGE_LAYOUT_GENERAL or VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL layout. Destination images must be in either the VK_IMAGE_LAYOUT_GENERAL or VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL layout. As a consequence, if an image is used as both source and destination of a copy, it must be in the VK_IMAGE_LAYOUT_GENERAL layout.
- Source images must have been created with the VK_IMAGE_USAGE_TRANSFER_SRC_BIT usage bit enabled and destination images must have been created with the VK_IMAGE_USAGE_TRANSFER_DST_BIT usage bit enabled
- Source buffers must have been created with the VK_BUFFER_USAGE_TRANSFER_SRC_BIT usage bit enabled and destination buffers must have been created with the VK_BUFFER_USAGE_TRANSFER_DST_BIT usage bit enabled

All copy commands are treated as "transfer" operations for the purposes of synchronization barriers.

17.2 Copying Data Between Buffers

To copy data between buffer objects, call:

```
void vkCmdCopyBuffer(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    srcBuffer,
    VkBuffer                                    dstBuffer,
    uint32_t                                    regionCount,
    const VkBufferCopy*                         pRegions);
```

srcBuffer and *dstBuffer* are handles to the source and destination buffers, respectively. *srcBuffer* and *dstBuffer* can reference the same buffer and/or the same memory, but the result is undefined if the copy regions overlap in memory. *regionCount* is the number of regions to copy, and *pRegions* is a pointer to an array of `VkBufferCopy` structures specifying the regions to copy.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *srcBuffer* must be a valid VkBuffer handle
- *dstBuffer* must be a valid VkBuffer handle
- *pRegions* must be a pointer to an array of *regionCount* VkBufferCopy structures
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support transfer, graphics or compute operations
- This command must only be called outside of a render pass instance
- The value of *regionCount* must be greater than `0`
- Each of *commandBuffer*, *srcBuffer* and *dstBuffer* must have been created, allocated or retrieved from the same VkDevice
- The sum of the *srcOffset* and *copySize* members of a given element of *pRegions* must be less than or equal to the size of *srcBuffer*
- The sum of the *dstOffset* and *copySize* members of a given element of *pRegions* must be less than or equal to the size of *dstBuffer*
- The source and destination regions defined by a given element of *pRegions* must not reference overlapping ranges of VkDeviceMemory
- *srcBuffer* must have been created with *VK_BUFFER_USAGE_TRANSFER_SRC_BIT* usage flag
- *dstBuffer* must have been created with *VK_BUFFER_USAGE_TRANSFER_DST_BIT* usage flag

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Each element of *pRegions* is a structure defined as:

```
typedef struct VkBufferCopy {
    VkDeviceSize                                    srcOffset;
    VkDeviceSize                                    dstOffset;
    VkDeviceSize                                    size;
} VkBufferCopy;
```

- *srcOffset* is the starting offset in bytes from the start of *srcBuffer*.
- *dstOffset* is the starting offset in bytes from the start of *dstBuffer*.
- *size* is the number of bytes to copy.

17.3 Copying Data Between Images

vkCmdCopyImage performs image copies in a similar manner to a host memcpy. It does not perform general-purpose conversions such as scaling, resizing, blending, or color-space or format conversions. Rather, it simply copies raw image data. **vkCmdCopyImage** can copy between images with different formats, provided the formats are compatible as defined below.

To copy data between image objects, call:

```
void vkCmdCopyImage(
    VkCommandBuffer                                 commandBuffer,
    VkImage                                         srcImage,
    VkImageLayout                                   srcImageLayout,
    VkImage                                         dstImage,
    VkImageLayout                                   dstImageLayout,
    uint32_t                                        regionCount,
    const VkImageCopy*                              pRegions);
```

srcImage and *dstImage* are handles to the source and destination images, respectively. *srcImageLayout* and *dstImageLayout* are the current layout of the source and destination images, respectively. *srcImage* and *dstImage* can reference the same image.

regionCount is the number of regions to copy, and *pRegions* is a pointer to an array of VkImageCopy structures specifying the regions to copy.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle

- `srcImage` must be a valid VkImage handle
- `srcImageLayout` must be a valid `VkImageLayout` value
- `dstImage` must be a valid VkImage handle
- `dstImageLayout` must be a valid `VkImageLayout` value
- `pRegions` must be a pointer to an array of `regionCount` valid VkImageCopy structures
- `commandBuffer` must be in the recording state
- The VkCommandPool that `commandBuffer` was allocated from must support transfer, graphics or compute operations
- This command must only be called outside of a render pass instance
- The value of `regionCount` must be greater than `0`
- Each of `commandBuffer`, `srcImage` and `dstImage` must have been created, allocated or retrieved from the same VkDevice
- The source region specified by a given element of `pRegions` must be a region that is contained within `srcImage`
- The destination region specified by a given element of `pRegions` must be a region that is contained within `dstImage`
- The union of all source regions, and the union of all destination regions, specified by the elements of `pRegions`, must not reference overlapping ranges of VkDeviceMemory
- `srcImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_SRC_BIT` usage flag
- `srcImageLayout` must specify the layout of `srcImage` at the time this command is executed on a VkDevice
- `srcImageLayout` must be either of VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL
- `dstImage` must have been created with `VK_IMAGE_USAGE_TRANSFER_DST_BIT` usage flag
- `dstImageLayout` must specify the layout of `dstImage` at the time this command is executed on a VkDevice
- `dstImageLayout` must be either of VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL
- The `VkFormat` of each of `srcImage` and `dstImage` must be compatible, as defined below
- The sample count of `srcImage` and `dstImage` must match

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Each element of *pRegions* is a structure defined as:

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

- *srcSubresource* and *dstSubresource* are the subresources of the images used for the source and destination image data, respectively, and are instances of the VkImageSubresourceLayers structure.
- *srcOffset* and *dstOffset* select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data.
- *extent* is the size in texels of the source image to copy in *width*, *height* and *depth*. 1D images use only *x* and *width*. 2D images use *x*, *y*, *width* and *height*. 3D images use *x*, *y*, *z*, *width*, *height* and *depth*.

Valid Usage

- *srcSubresource* must be a valid VkImageSubresourceLayers structure
- *dstSubresource* must be a valid VkImageSubresourceLayers structure
- The *aspectMask* member of *srcSubresource* and *dstSubresource* must match
- The *layerCount* member of *srcSubresource* and *dstSubresource* must match
- If either of the calling command's *srcImage* or *dstImage* parameters are of `VkImageType` VK_IMAGE_TYPE_3D, the *layerCount* member of *srcSubresource* and *dstSubresource* must be `1`
- The *aspectMask* member of *srcSubresource* must specify aspects present in the calling command's *srcImage*
- The *aspectMask* member of *dstSubresource* must specify aspects present in the calling command's *dstImage*
- *srcOffset.x* and (*extent.width* + *srcOffset.x*) must both be greater than or equal to 0 and less than or equal to the source image subresource width
- *srcOffset.y* and (*extent.height* + *srcOffset.y*) must both be greater than or equal to 0 and less than or equal to the source image subresource height
- *srcOffset.z* and (*extent.depth* + *srcOffset.z*) must both be greater than or equal to 0 and less than or equal to the source image subresource depth
- *dstOffset.x* and (*extent.width* + *dstOffset.x*) must both be greater than or equal to 0 and less than or equal to the destination image subresource width
- *dstOffset.y* and (*extent.height* + *dstOffset.y*) must both be greater than or equal to 0 and less than or equal to the destination image subresource height

- *dstOffset.z* and (*extent.depth* + *dstOffset.z*) must both be greater than or equal to 0 and less than or equal to the destination image subresource depth
- If the calling command's *srcImage* is a compressed format image:
- * all members of *srcOffset* must be a multiple of the block size in the relevant dimensions
- * *extent.width* must be a multiple of the block width or (*extent.width* + *srcOffset.x*) must equal the source image subresource width
- * *extent.height* must be a multiple of the block height or (*extent.height* + *srcOffset.y*) must equal the source image subresource height
- * *extent.depth* must be a multiple of the block depth or (*extent.depth* + *srcOffset.z*) must equal the source image subresource depth
- If the calling command's *dstImage* is a compressed format image:
- * all members of *dstOffset* must be a multiple of the block size in the relevant dimensions
- * *extent.width* must be a multiple of the block width or (*extent.width* + *dstOffset.x*) must equal the destination image subresource width
- * *extent.height* must be a multiple of the block height or (*extent.height* + *dstOffset.y*) must equal the destination image subresource height
- * *extent.depth* must be a multiple of the block depth or (*extent.depth* + *dstOffset.z*) must equal the destination image subresource depth
- *srcOffset*, *dstOffset*, and *extent* must respect the image transfer granularity requirements of the queue family that it will be submitted against, as described in Physical Device Enumeration

The VkImageSubresourceLayers structure is defined as:

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

- *aspectMask* is a combination of `VkImageAspectFlagBits`, selecting the color, depth and/or stencil attachments to be copied.
- *mipLevel* is the mipmap level to copy from.
- *baseArrayLayer* and *layerCount* are the starting layer and number of layers to copy.

Valid Usage

- *aspectMask* must be a valid combination of `VkImageAspectFlagBits` values
- *aspectMask* must not be `0`
- If *aspectMask* contains VK_IMAGE_ASPECT_COLOR_BIT, it must not contain either of VK_IMAGE_ASPECT_DEPTH_BIT or VK_IMAGE_ASPECT_STENCIL_BIT
- *aspectMask* must not contain VK_IMAGE_ASPECT_METADATA_BIT

Copies are done layer by layer starting with *baseArrayLayer* member of *srcSubresource* for the source and *dstSubresource* for the destination. *layerCount* layers are copied to the destination image.

The formats of *srcImage* and *dstImage* must be compatible. Formats are considered compatible if their texel size in bytes is the same between both formats. For example, VK_FORMAT_R8G8B8A8_UNORM is compatible with VK_FORMAT_R32_UINT because because both texels are 4 bytes in size. Formats with both depth and stencil components have to match exactly.

`vkCmdCopyImage` allows copying between size-compatible compressed and uncompressed internal formats. Formats are size-compatible if the texel size of the uncompressed format is equal to the block size in bytes of the compressed format. Such a copy does not perform on-the-fly compression or decompression. When copying from an uncompressed format to a compressed format, each texel of uncompressed data becomes a single block of compressed data. When copying from a compressed format to an uncompressed format, a block of compressed data becomes a single texel of uncompressed data. Thus, for example, it is legal to copy between a 128-bit uncompressed format and a compressed format which uses 8-bit/texel 4x4 blocks, or between a 64-bit uncompressed format and a compressed format which uses 4-bit/texel 4x4 blocks.

When copying between compressed and uncompressed formats the *extent* members represent the texel dimensions of the source image and not the destination. When copying from a compressed image to an uncompressed image the image texel dimensions written to the uncompressed image will be source extent divided by the block size. When copying from an uncompressed image to a compressed image the image texel dimensions written to the compressed image will be the source extent multiplied by the block size. In both cases the number of bytes read and the number of bytes written will be identical.

Copying to or from block-compressed images is typically done in multiples of the block. For this reason the *extent* must be a multiple of the block dimension. There is one exception to this rule which is required to handle compressed images created with dimensions that are not a multiple of the block dimensions. If the *srcImage* is compressed and if *extent.width* is not a multiple of the block width then (*extent.width* + *srcOffset.x*) must equal the subresource width, if *extent.height* is not a multiple of the block height then (*extent.height* + *srcOffset.y*) must equal the subresource height and if *extent.depth* is not a multiple of the block depth then (*extent.depth* + *srcOffset.z*) must equal the subresource depth. Similarily if the *dstImage* is compressed and if *extent.width* is not a multiple of the block width then (*extent.width* + *dstOffset.x*) must equal the subresource width, if *extent.height* is not a multiple of the block height then (*extent.height* + *dstOffset.y*) must equal the subresource height and if *extent.depth* is not a multiple of the block depth then (*extent.depth* + *dstOffset.z*) must equal the subresource depth. This allows the last block of the image in each non-multiple dimension to be included as a source or target of the copy.

`vkCmdCopyImage` can be used to copy image data between multisample images, but both images must have the same number of samples.

17.4 Copying Data Between Buffers and Images

To copy data from a buffer object to an image object, call:

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

srcBuffer and *dstImage* are handles to the source buffer and destination image, respectively. *dstImageLayout* is the layout of the destination image for the copy.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *srcBuffer* must be a valid VkBuffer handle
- *dstImage* must be a valid VkImage handle
- *dstImageLayout* must be a valid `VkImageLayout` value
- *pRegions* must be a pointer to an array of *regionCount* valid VkBufferImageCopy structures
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support transfer, graphics or compute operations
- This command must only be called outside of a render pass instance
- The value of *regionCount* must be greater than `0`
- Each of *commandBuffer*, *srcBuffer* and *dstImage* must have been created, allocated or retrieved from the same VkDevice
- The buffer region specified by a given element of *pRegions* must be a region that is contained within *srcBuffer*
- The image region specified by a given element of *pRegions* must be a region that is contained within *dstImage*
- The source and destination regions defined by a given element of *pRegions* must not reference overlapping ranges of VkDeviceMemory
- *srcBuffer* must have been created with *VK_BUFFER_USAGE_TRANSFER_SRC_BIT* usage flag
- *dstImage* must have been created with *VK_IMAGE_USAGE_TRANSFER_DST_BIT* usage flag
- *dstImage* must have a sample count equal to VK_SAMPLE_COUNT_1_BIT
- *dstImageLayout* must specify the layout of *dstImage* at the time this command is executed on a VkDevice
- *dstImageLayout* must be either of VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

To copy data from an image object to a buffer object, call:

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

srcImage and *dstBuffer* are handles to the source image and destination buffer, respectively. *srcImageLayout* is the layout of the source image for the copy.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *srcImage* must be a valid VkImage handle
- *srcImageLayout* must be a valid `VkImageLayout` value
- *dstBuffer* must be a valid VkBuffer handle
- *pRegions* must be a pointer to an array of *regionCount* valid VkBufferImageCopy structures
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support transfer, graphics or compute operations
- This command must only be called outside of a render pass instance
- The value of *regionCount* must be greater than `0`
- Each of *commandBuffer*, *srcImage* and *dstBuffer* must have been created, allocated or retrieved from the same VkDevice
- The image region specified by a given element of *pRegions* must be a region that is contained within *srcImage*
- The buffer region specified by a given element of *pRegions* must be a region that is contained within *dstBuffer*

- The source and destination regions defined by a given element of *pRegions* must not reference overlapping ranges of VkDeviceMemory
- *srcImage* must have been created with *VK_IMAGE_USAGE_TRANSFER_SRC_BIT* usage flag
- *srcImage* must have a sample count equal to VK_SAMPLE_COUNT_1_BIT
- *srcImageLayout* must specify the layout of *srcImage* at the time this command is executed on a VkDevice
- *srcImageLayout* must be either of VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL
- *dstBuffer* must have been created with *VK_BUFFER_USAGE_TRANSFER_DST_BIT* usage flag

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

For both **vkCmdCopyBufferToImage** and **vkCmdCopyImageToBuffer**, *regionCount* is the number of regions to copy, and *pRegions* is a pointer to an array of `VkBufferImageCopy` structures specifying the regions to copy.

Each element of *pRegions* is a structure defined as:

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

- *bufferOffset* is the offset in bytes from the start of the buffer object where the image data is copied from or to.
- *bufferRowLength* and *bufferImageHeight* specify the data in buffer memory as a subregion of a larger two- or three-dimensional image, and control the addressing calculations of data in buffer memory. If either of these values is zero, that aspect of the buffer memory is considered to be tightly packed according to the *imageExtent*.
- *imageSubresource* is an `VkImageSubresourceLayers` used to specify the specific subresources of the image used for the source or destination image data.
- *imageOffset* selects the initial x, y, z offsets in texels of the sub-region of the source or destination image data.
- *imageExtent* is the size in texels of the image to copy in *width*, *height* and *depth*. 1D images use only *x* and *width*. 2D images use *x*, *y*, *width* and *height*. 3D images use *x*, *y*, *z*, *width*, *height* and *depth*.

When copying to or from a depth or stencil aspect, the data in buffer memory uses a layout that is a (mostly) tightly packed representation of the depth or stencil data. Specifically:

- data copied to or from the stencil aspect of any depth/stencil or VK_FORMAT_S8_UINT format is tightly packed with one VK_FORMAT_S8_UINT value per texel.
- data copied to or from the depth aspect of a VK_FORMAT_D16_UNORM or VK_FORMAT_D16_UNORM_S8_UINT format is tightly packed with one VK_FORMAT_D16_UNORM value per texel.
- data copied to or from the depth aspect of a VK_FORMAT_D32_SFLOAT or VK_FORMAT_D32_SFLOAT_S8_UINT format is tightly packed with one VK_FORMAT_D32_SFLOAT value per texel.
- data copied to or from the depth aspect of a VK_FORMAT_X8_D24_UNORM_PACK32 or VK_FORMAT_D24_UNORM_S8_UINT format is packed with one 32-bit word per texel with the D24 value in the LSBs of the word, and undefined values in the eight MSBs.

Note
To copy both the depth and stencil aspects of a depth/stencil format, two entries in *pRegions* can be used, where one specifies the depth aspect in *imageSubresource*, and the other specifies the stencil aspect.

Because depth/stencil aspect buffer to image copies may require format conversions on some implementations, they are not supported on queues that do not support graphics.

Copies are done layer by layer starting with image layer *baseArrayLayer* member of *imageSubresource*. *layerCount* layers are copied from the source image or to the destination image.

Valid Usage

- *imageSubresource* must be a valid VkImageSubresourceLayers structure
- *bufferOffset* must be a multiple of the calling command's VkImage parameter's texel size
- *bufferOffset* must be a multiple of `4`
- *bufferRowLength* must be `0`, or greater than or equal to the *width* member of *imageExtent*
- *bufferImageHeight* must be `0`, or greater than or equal to the *height* member of *imageExtent*
- *imageOffset.x* and (*imageExtent.width* + *imageOffset.x*) must both be greater than or equal to 0 and less than or equal to the image subresource width
- *imageOffset.y* and (imageExtent.height + *imageOffset.y*) must both be greater than or equal to 0 and less than or equal to the image subresource height
- *imageOffset.z* and (imageExtent.depth + *imageOffset.z*) must both be greater than or equal to 0 and less than or equal to the image subresource depth
- If the calling command's VkImage parameter is a compressed format image:
- * *bufferRowLength*, *bufferImageHeight* and all members of *imageOffset* must be a multiple of the block size in the relevant dimensions

- * `bufferOffset` must be a multiple of the block size in bytes
- * `imageExtent.width` must be a multiple of the block width or (`imageExtent.width` + `imageOffset.x`) must equal the image subresource width
- * `imageExtent.height` must be a multiple of the block height or (`imageExtent.height` + `imageOffset.y`) must equal the image subresource height
- * `imageExtent.depth` must be a multiple of the block depth or (`imageExtent.depth` + `imageOffset.z`) must equal the image subresource depth
- `bufferOffset`, `bufferRowLength`, `bufferImageHeight` and all members of `imageOffset` and `imageExtent` must respect the image transfer granularity requirements of the queue family that it will be submitted against, as described in Physical Device Enumeration
- The `aspectMask` member of `srcSubresource` must specify aspects present in the calling command's VkImage parameter
- The `aspectMask` member of `pSubresource` must only have a single bit set
- If the calling command's VkImage parameter is of `VkImageType` VK_IMAGE_TYPE_3D, the `layerCount` member of `srcSubresource` and `dstSubresource` must be `1`

Pseudocode for image/buffer addressing is:

```
rowLength = region->bufferRowLength;
if (rowLength == 0) {
    rowLength = region->imageExtent.width;
}
imageHeight = region->bufferImageHeight;
if (imageHeight == 0) {
    imageHeight = region->imageExtent.height;
}
texelSize = <texel size taken from the src/dstImage>;

address of (x,y,z) = region->bufferOffset + (((z * imageHeight) + y) * rowLength + x ↩
    ) * texelSize;

where x,y,z range from (0,0,0) to region->imageExtent.{width,height,depth}.
```

Note that `imageOffset` does not affect addressing calculations for buffer memory. Instead, `bufferOffset` can be used to select the starting address in buffer memory.

For block-compression formats, all parameters are still specified in texels rather than blocks, but the addressing math operates on whole blocks. Pseudocode for compressed copy addressing is:

```
rowLength = region->bufferRowLength;
if (rowLength == 0) {
    rowLength = region->imageExtent.width;
}
imageHeight = region->bufferImageHeight;
if (imageHeight == 0) {
    imageHeight = region->imageExtent.height;
}
blockSizeInBytes = <block size taken from the src/dstImage>;
```

```
rowLength /= blockWidth;
imageHeight /= blockHeight;

address of (x,y,z) = region->bufferOffset + (((z * imageHeight) + y) * rowLength + x ↩
    ) * blockSizeInBytes;

where x,y,z range from (0,0,0) to region->imageExtent.{width/blockWidth,height/ ↩
    blockHeight,depth/blockDepth}.
```

Copying to or from block-compressed images is typically done in multiples of the block. For this reason the *imageExtent* must be a multiple of the block dimension. There is one exception to this rule which is required to handle compressed images created with dimensions that are not a multiple of the block dimensions. If *imageExtent.width* is not a multiple of the block width then (*imageExtent.width* + *imageOffset.x*) must equal the subresource width, if *imageExtent.height* is not a multiple of the block height then (*imageExtent.height* + *imageOffset.y*) must equal the subresource height and if *imageExtent.depth* is not a multiple of the block depth then (*imageExtent.depth* + *imageOffset.z*) must equal the subresource depth. This allows the last block of the image in each non-multiple dimension to be included as a source or target of the copy.

17.5 Image Copies with Scaling

To copy regions of a source image into a destination image, potentially performing format conversion, arbitrary scaling, and filtering, call:

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

srcImage and *dstImage* are handles to the source and destination images, respectively.

srcImageLayout and *dstImageLayout* are the layout of the source and destination images, respectively.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *srcImage* must be a valid VkImage handle
- *srcImageLayout* must be a valid `VkImageLayout` value
- *dstImage* must be a valid VkImage handle
- *dstImageLayout* must be a valid `VkImageLayout` value
- *pRegions* must be a pointer to an array of *regionCount* valid VkImageBlit structures

- *filter* must be a valid VkFilter value
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- This command must only be called outside of a render pass instance
- The value of *regionCount* must be greater than 0
- Each of *commandBuffer*, *srcImage* and *dstImage* must have been created, allocated or retrieved from the same VkDevice
- The source region specified by a given element of *pRegions* must be a region that is contained within *srcImage*
- The destination region specified by a given element of *pRegions* must be a region that is contained within *dstImage*
- The source and destination regions defined by a given element of *pRegions* must not reference overlapping ranges of VkDeviceMemory
- *srcImage* must use a format that supports VK_FORMAT_FEATURE_BLIT_SRC_BIT, which is indicated by VkFormatProperties::*linearTilingFeatures* (for linear tiled images) or VkFormatProperties::*optimalTilingFeatures* (for optimally tiled images) - as returned by **vkGetPhysicalDeviceFormatProperties**
- *srcImage* must have been created with *VK_IMAGE_USAGE_TRANSFER_SRC_BIT* usage flag
- *srcImageLayout* must specify the layout of *srcImage* at the time this command is executed on a VkDevice
- *srcImageLayout* must be either of VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL
- *dstImage* must use a format that supports VK_FORMAT_FEATURE_BLIT_DST_BIT, which is indicated by VkFormatProperties::*linearTilingFeatures* (for linear tiled images) or VkFormatProperties::*optimalTilingFeatures* (for optimally tiled images) - as returned by **vkGetPhysicalDeviceFormatProperties**
- *dstImage* must have been created with *VK_IMAGE_USAGE_TRANSFER_DST_BIT* usage flag
- *dstImageLayout* must specify the layout of *dstImage* at the time this command is executed on a VkDevice
- *dstImageLayout* must be either of VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL
- The sample count of *srcImage* and *dstImage* must both be equal to VK_SAMPLE_COUNT_1_BIT
- If either of *srcImage* or *dstImage* was created with a signed integer VkFormat, the other must also have been created with a signed integer VkFormat
- If either of *srcImage* or *dstImage* was created with an unsigned integer VkFormat, the other must also have been created with an unsigned integer VkFormat
- If either of *srcImage* or *dstImage* was created with a VkFormat that has depth or stencil aspects, the other must have exactly the same format
- If *srcImage* was created with a VkFormat that has depth or stencil aspects, *filter* must be VK_FILTER_NEAREST

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

vkCmdBlitImage must not be used for multisampled source or destination images. Use `vkCmdResolveImage` for this purpose.

regionCount is the number of regions to copy, and *pRegions* is a pointer to an array of `VkImageBlit` structures specifying the regions to copy.

Each element of *pRegions* is a structure defined as:

```
typedef struct VkImageBlit {
    VkImageSubresourceLayers                        srcSubresource;
    VkOffset3D                                      srcOffset;
    VkExtent3D                                      srcExtent;
    VkImageSubresourceLayers                        dstSubresource;
    VkOffset3D                                      dstOffset;
    VkExtent3D                                      dstExtent;
} VkImageBlit;
```

For each of element of *pRegions* array, a blit operation is performed between the region of *srcSubresource* of *srcImage* (bounded by *srcOffset* and *srcExtent*) and a region of *dstSubresource* of *dstImage* (bounded by *dstOffset* and *dstExtent*).

Valid Usage

- *srcSubresource* must be a valid VkImageSubresourceLayers structure
- *dstSubresource* must be a valid VkImageSubresourceLayers structure
- The *aspectMask* member of *srcSubresource* and *dstSubresource* must match
- The *layerCount* member of *srcSubresource* and *dstSubresource* must match
- If either of the calling command's *srcImage* or *dstImage* parameters are of `VkImageType` VK_IMAGE_TYPE_3D, the *layerCount* member of *srcSubresource* and *dstSubresource* must be `1`
- The *aspectMask* member of *srcSubresource* must specify aspects present in the calling command's *srcImage*
- The *aspectMask* member of *dstSubresource* must specify aspects present in the calling command's *dstImage*

If sizes of source and destination extents do not match, scaling is performed by applying the filtering mode specified by *filter* parameter. VK_FILTER_LINEAR uses bilinear interpolation, and VK_FILTER_NEAREST uses point sampling. When using VK_FILTER_LINEAR, filtering may reference pixel coordinates outside the source region (for magnifying blits). In case those pixel coordinates are outside bounds of the image level, the coordinates are clamped as in VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE texture clamping mode. However if the coordinates are outside the source region but inside the image level, the implementation may or may not clamp coordinates to the source region.

If source and destination extents are identical, no filtering is performed. In case of negative values passed in *`srcExtent`* or *`dstExtent`*, the copied pixels are reversed in appropriate direction(s). If *`width`* is negative, then the region is bounded in the x dimension by [x+width, x) rather than [x, x+width), and similarly for y/height and z/depth. If corresponding members in both *`srcExtent`* and *`dstExtent`* are negative, no reversal is performed.

Blits are done layer by layer starting with *`baseArrayLayer`* member of *`srcSubresource`* for the source and *`dstSubresource`* for the destination. *`layerCount`* layers are blitted to the destination image.

3D textures are blitted slice by slice, with the number of slices equal to the *`depth`* member of *`dstExtent`*. The source region is bounded by the *`srcOffset.z`* and *`srcExtent.depth`* and the destination region is bounded by the *`dstOffset.z`* and *`dstExtent.depth`*. For each destination slice, a source z coordinate is linearly interpolated between *`srcOffset.z`* and *`srcOffset.z`* + *`srcExtent.depth`*. If the *`filter`* parameter is VK_FILTER_LINEAR then the value sampled from the source image is taken by doing linear filtering using the interpolated z coordinate. If *`filter`* parameter is VK_FILTER_NEAREST then value sampled from the source image is taken from the single nearest slice (with undefined rounding mode).

If **`vkCmdBlitImage`** is used on images of different formats, the following conversion rules apply:

- Integer formats can only be converted to other integer formats with the same signedness.
- No format conversion is available between depth and stencil images - the formats must match.
- Format conversions on unorm, snorm, unscaled and packed float formats of the copied aspect of the image are performed by first converting the pixels to float values.
- In case of sRGB source format, values are converted to linear color space prior to filtering.
- After filtering, the float values are first clamped and then cast to the destination image format. In case of sRGB destination format, values are converted to sRGB color space before writing the pixel to the image.

Signed and unsigned integers are converted by first clamping to the representable range of destination format, then casting the value.

17.6 Resolving Multisample Images

To resolve a multisample image to a non-multisample image, call:

```
void vkCmdResolveImage(
    VkCommandBuffer                             commandBuffer,
    VkImage                                     srcImage,
    VkImageLayout                               srcImageLayout,
    VkImage                                     dstImage,
    VkImageLayout                               dstImageLayout,
    uint32_t                                    regionCount,
    const VkImageResolve*                       pRegions);
```

`srcImage` and *`dstImage`* are a handles to the source and destination images, respectively. *`srcImageLayout`* and *`dstImageLayout`* are the layout of the source and destination images, respectively.

regionCount is the number of regions to resolve, and *pRegions* is a pointer to an array of `VkImageResolve` structures specifying the regions to resolve.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *srcImage* must be a valid VkImage handle
- *srcImageLayout* must be a valid `VkImageLayout` value
- *dstImage* must be a valid VkImage handle
- *dstImageLayout* must be a valid `VkImageLayout` value
- *pRegions* must be a pointer to an array of *regionCount* valid VkImageResolve structures
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- This command must only be called outside of a render pass instance
- The value of *regionCount* must be greater than `0`
- Each of *commandBuffer*, *srcImage* and *dstImage* must have been created, allocated or retrieved from the same VkDevice
- The source region specified by a given element of *pRegions* must be a region that is contained within *srcImage*
- The destination region specified by a given element of *pRegions* must be a region that is contained within *dstImage*
- The source and destination regions defined by a given element of *pRegions* must not reference overlapping ranges of VkDeviceMemory
- *srcImage* must have a sample count equal to any valid sample count value other than VK_SAMPLE_COUNT_1_BIT
- *dstImage* must have a sample count equal to VK_SAMPLE_COUNT_1_BIT
- *srcImageLayout* must specify the layout of *srcImage* at the time this command is executed on a VkDevice
- *srcImageLayout* must be either of VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL
- *dstImageLayout* must specify the layout of *dstImage* at the time this command is executed on a VkDevice
- *dstImageLayout* must be either of VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL or VK_IMAGE_LAYOUT_GENERAL
- If *dstImage* was created with *tiling* equal to VK_IMAGE_TILING_LINEAR, *dstImage* must have been created with a *format* that supports being a color attachment, as specified by the VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT flag in VkFormatProperties::*linearTilingFeatures* returned by **`vkGetPhysicalDeviceFormatProperties`**

- If `dstImage` was created with `tiling` equal to VK_IMAGE_TILING_OPTIMAL, `dstImage` must have been created with a `format` that supports being a color attachment, as specified by the VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT flag in VkFormatProperties::`optimalTilingFeatures` returned by **vkGetPhysicalDeviceFormatProperties**

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

Each element of `pRegions` is a structure defined as:

```
typedef struct VkImageResolve {
    VkImageSubresourceLayers                        srcSubresource;
    VkOffset3D                                      srcOffset;
    VkImageSubresourceLayers                        dstSubresource;
    VkOffset3D                                      dstOffset;
    VkExtent3D                                      extent;
} VkImageResolve;
```

`srcSubresource` and `dstSubresource` are `VkImageSubresourceLayers` structures used to specify the specific subresources of the images used for the source and destination image data, respectively. Resolve of depth or stencil images is not supported.

Valid Usage

- `srcSubresource` must be a valid VkImageSubresourceLayers structure
- `dstSubresource` must be a valid VkImageSubresourceLayers structure
- The `aspectMask` member of `srcSubresource` and `dstSubresource` must only contain VK_IMAGE_ASPECT_COLOR_BIT
- The `layerCount` member of `srcSubresource` and `dstSubresource` must be `1`

During the resolve the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination. If the source formats are floating-point or normalized types, the sample values for each pixel are resolved in an implementation-dependent manner. If the source formats are integer types, a single sample's value is selected for each pixel.

`srcOffset` and `dstOffset` select the initial x, y, and z offsets in texels of the sub-regions of the source and destination image data. `extent` is the size in texels of the source image to resolve in `width`, `height` and `depth`. 1D images use only *`x`* and *`width`*. 2D images use *`x`*, *`y`*, *`width`* and *`height`*. 3D images use *`x`*, *`y`*, *`z`*, *`width`*, *`height`* and *`depth`*. For array images the *`baseArrayLayer`* member of *`srcSubresource`* is the source array layer, and the *`baseArrayLayer`* member of *`dstSubresource`* is the destination array layer.

Chapter 18

Drawing Commands

Drawing commands (command with "Draw" in the name) provoke work in a graphics pipeline. Drawing commands are recorded into a command buffer and when executed by a queue, will produce work which executes according to the currently bound graphics pipeline. A graphics pipeline must be bound to a command buffer before any drawing commands are recorded in that command buffer.

Each draw is made up of zero or more vertices and zero or more instances, which are processed by the device and result in the assembly of primitives. Primitives are assembled according to the *pInputAssemblyState* member of the VkGraphicsPipelineCreateInfo structure, which is of type VkPipelineInputAssemblyStateCreateInfo:

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

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *primitiveRestartEnable* controls whether a special vertex index value is treated as restarting the assembly of primitives. This enable only applies to indexed draws (`vkCmdDrawIndexed` and `vkCmdDrawIndexedIndirect`), and the special index value is either 0xFFFFFFFF when the *indexType* parameter of **`vkCmdBindIndexBuffer`** is equal to VK_INDEX_TYPE_UINT32, or 0xFFFF when *indexType* is equal to VK_INDEX_TYPE_UINT16. Primitive restart is not allowed for "list" topologies.
- *topology* is a `VkPrimitiveTopology` defining the primitive topology, as described below.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`
- *topology* must be a valid `VkPrimitiveTopology` value
- If *topology* is VK_PRIMITIVE_TOPOLOGY_POINT_LIST, VK_PRIMITIVE_TOPOLOGY_LINE_LIST, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY or VK_PRIMITIVE_TOPOLOGY_PATCH_LIST, the value of *primitiveRestartEnable* must be VK_FALSE
- If the geometry shaders feature is not enabled, *topology* must not be any of VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY, VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY or VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY
- If the tessellation shaders feature is not enabled, *topology* must not be VK_PRIMITIVE_TOPOLOGY_PATCH_LIST

Restarting the assembly of primitives discards the most recent index values if those elements formed an incomplete primitive, and restarts the primitive assembly using the subsequent indices, but only assembling the immediately following element through the end of the originally specified elements. The primitive restart index value comparison is performed before adding the *vertexOffset* value to the index value.

18.1 Primitive Topologies

Primitive topology determines how consecutive vertices are organized into primitives, and determines the type of primitive that is used at the beginning of the graphics pipeline. The effective topology for later stages of the pipeline is altered by tessellation or geometry shading (if either is in use) and depends on the execution modes of those shaders. Supported topologies are defined by `VkPrimitiveTopology` and include:

```
typedef enum VkPrimitiveTopology {
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST = 1,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP = 2,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST = 3,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP = 4,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN = 5,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY = 6,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY = 7,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY = 8,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY = 9,
    VK_PRIMITIVE_TOPOLOGY_PATCH_LIST = 10,
} VkPrimitiveTopology;
```

Each primitive topology, and its construction from a list of vertices, is summarized below.

18.1.1 Points

A series of individual points are specified with *topology* VK_PRIMITIVE_TOPOLOGY_POINT_LIST. Each vertex defines a separate point.

18.1.2 Separate Lines

Individual line segments, each defined by a pair of vertices, are specified with *topology* VK_PRIMITIVE_TOPOLOGY_LINE_LIST. The first two vertices define the first segment, with subsequent pairs of vertices each defining one more segment. If the number of vertices is odd, then the last vertex is ignored.

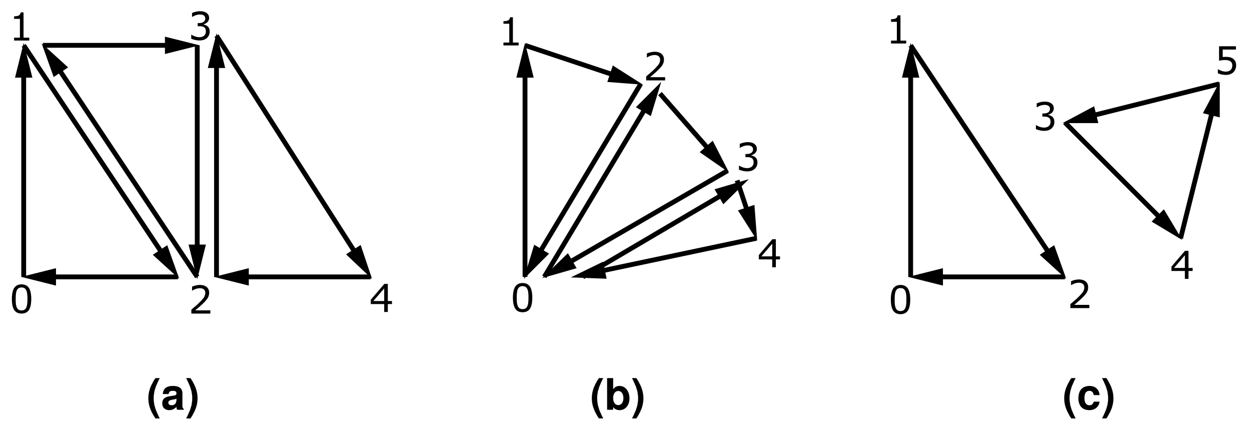
18.1.3 Line Strips

A series of one or more connected line segments are specified with *topology* VK_PRIMITIVE_TOPOLOGY_LINE_STRIP. In this case, the first vertex specifies the first segment's start point while the second vertex specifies the first segment's endpoint and the second segment's start point. In general, the *i*th vertex (for $i > 0$) specifies the beginning of the *i*th segment and the end of the $i-1$st. The last vertex specifies the end of the last segment. If only one vertex is specified, then no primitive is generated.

18.1.4 Triangle Strips

A triangle strip is a series of triangles connected along shared edges, and is specified with *topology* VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP. In this case, the first three vertices define the first triangle, and their order is significant. Each subsequent vertex defines a new triangle using that point along with the last two vertices from the previous triangle, as shown in figure Triangle strips, fans, and lists. If fewer than three vertices are specified, no primitive is produced. The order of vertices in successive triangles changes as shown in the figure, so that all triangle faces have the same orientation.

Triangle strips, fans, and lists



18.1.5 Triangle Fans

A triangle fan is specified with *topology* VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN. It is similar to a triangle strip, but changes the vertex replaced from the previous triangle as shown in figure Triangle strips, fans, and lists, so that all triangles in the fan share a common vertex.

18.1.6 Separate Triangles

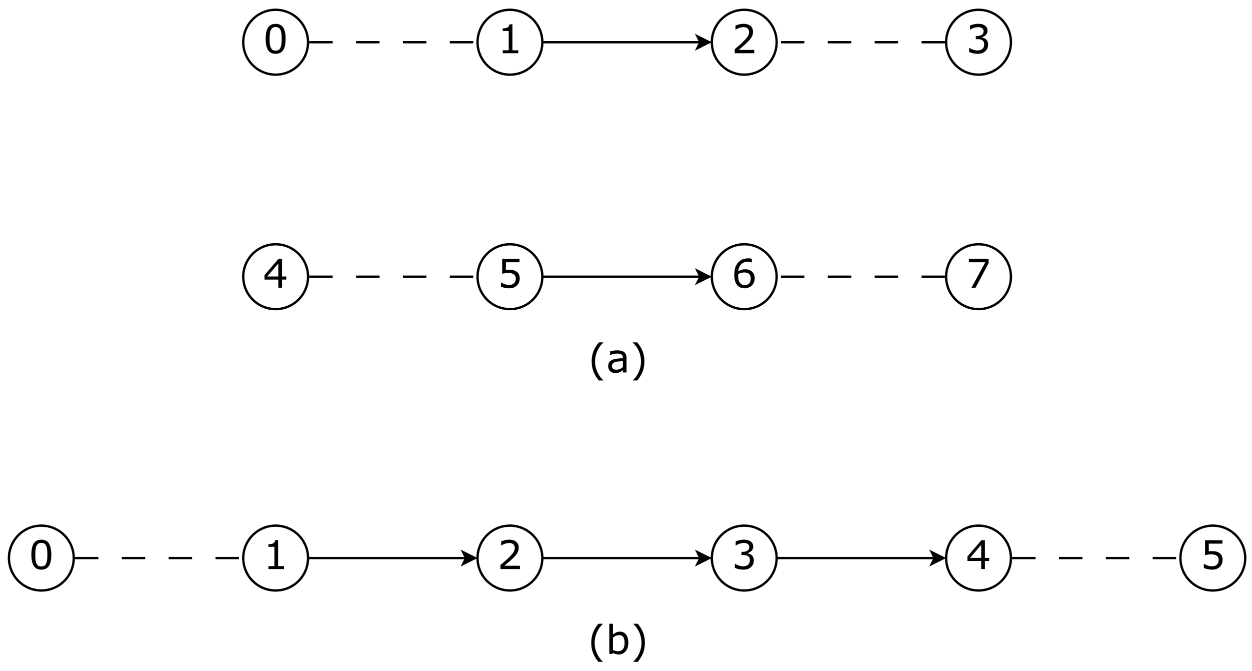
Separate triangles are specified with *`topology`* VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, as shown in figure Triangle strips, fans, and lists. In this case, vertices $3i$, $3i+1$, and $3i+2$ vertices (in that order) determine a triangle for each $i = 0, 1, \ldots, n-1$, where there are $3n+k$ vertices drawn. k is either 0, 1, or 2; if k is not zero, the final k vertices are ignored.

18.1.7 Lines With Adjacency

Lines with adjacency are specified with *`topology`* VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY, and are independent line segments where each endpoint has a corresponding *adjacent* vertex that is accessible in a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

A line segment is drawn from the $4i+1$st vertex to the $4i+2$nd vertex for each $i = 0, 1, \ldots, n-1$, where there are $4n+k$ vertices. k is either 0, 1, 2, or 3; if k is not zero, the final k vertices are ignored. For line segment i, the $4i$th and $4i+3$rd vertices are considered adjacent to the $4i+1$st and $4i+2$nd vertices, respectively, as shown in figure Lines with adjacency.

Lines with adjacency



18.1.8 Line Strips With Adjacency

Line strips with adjacency are specified with *`topology`* VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY and are similar to line strips, except that each line segment has a pair of adjacent vertices that are accessible in a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

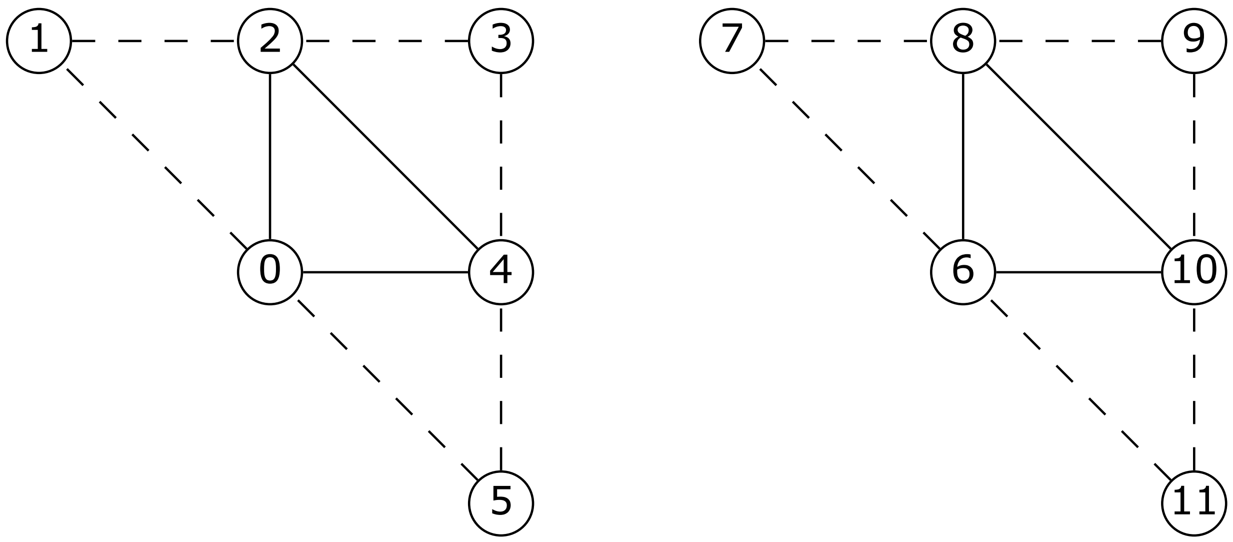
A line segment is drawn from the $i+1$st vertex to the $i+2$nd vertex for each $i = 0, 1, \ldots, n-1$, where there are $n+3$ vertices. If there are fewer than four vertices, all vertices are ignored. For line segment i, the ith and $i+3$rd vertex are considered adjacent to the $i+1$st and $i+2$nd vertices, respectively, as shown in figure Lines with adjacency.

18.1.9 Triangle List With Adjacency

Triangles with adjacency are specified with *topology* VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY, and are similar to separate triangles except that each triangle edge has an adjacent vertex that is accessible in a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

The $6i$th, $6i+2$nd, and $6i+4$th vertices (in that order) determine a triangle for each $i = 0, 1, \ldots, n-1$, where there are $6n+k$ vertices. k is either 0, 1, 2, 3, 4, or 5; if k is non-zero, the final k vertices are ignored. For triangle i, the $6i+1$st, $6i+3$rd, and $6i+5$th vertices are considered adjacent to edges from the $6i$th to the $6i+2$nd, from the $6i+2$nd to the $6i+4$th, and from the $6i+4$th to the $6i$th vertices, respectively, as shown in figure Triangles with adjacency.

Triangles with adjacency



18.1.10 Triangle Strips With Adjacency

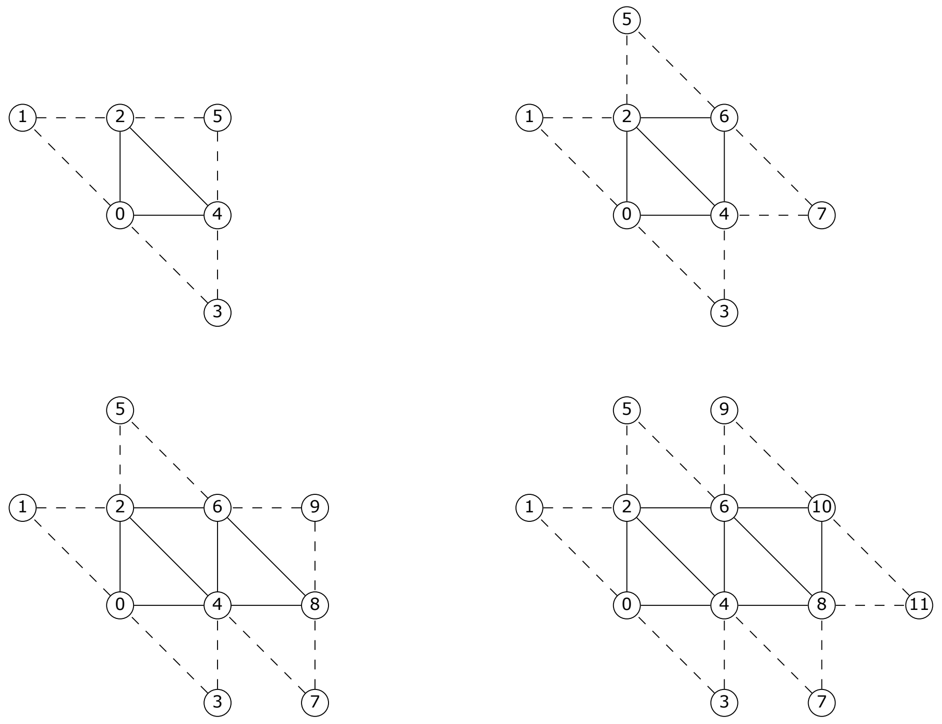
Triangle strips with adjacency are specified with *topology* VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY, and are similar to triangle strips except that each triangle edge has an adjacent vertex that is accessible in a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored.

In triangle strips with adjacency, n triangles are drawn where there are $2(n+2)+k$ vertices. k is either 0 or 1; if k is 1, the final vertex is ignored. If there are fewer than 6 vertices, the entire primitive is ignored. Table Table 18.1 describes the vertices and order used to draw each triangle, and which vertices are considered adjacent to each edge of the triangle, as shown in figure Triangle strips with adjacency.

Table 18.1: Triangles generated by triangle strips with adjacency.

	Primitive Vertices			**Adjacent Vertices**		
Primitive	1st	2nd	3rd	1/2	2/3	3/1
only ($i = 0, n = 1$)	0	2	4	1	5	3
first ($i = 0$)	0	2	4	1	6	3
middle (i odd)	$2i+2$	$2i$	$2i+4$	$2i-2$	$2i+3$	$2i+6$
middle (i even)	$2i$	$2i+2$	$2i+4$	$2i-2$	$2i+6$	$2i+3$
last ($i = n-1$, i odd)	$2i+2$	$2i$	$2i+4$	$2i-2$	$2i+3$	$2i+5$
last ($i = n-1$, i even)	$2i$	$2i+2$	$2i+4$	$2i-2$	$2i+5$	$2i+3$

Triangle strips with adjacency



18.1.11 Separate Patches

Separate patches are specified with *topology* VK_PRIMITIVE_TOPOLOGY_PATCH_LIST. A patch is an ordered collection of vertices used for primitive tessellation. The vertices comprising a patch have no implied geometric ordering, and are used by tessellation shaders and the fixed-function tessellator to generate new point, line, or triangle primitives.

Each patch in the series has a fixed number of vertices, specified by the *patchControlPoints* member of the `VkPipelineTessellationStateCreateInfo` structure passed to `vkCreateGraphicsPipelines`. Once assembled and vertex shaded, these patches are provided as input to the tessellation control shader stage.

If the number of vertices in a patch is given by v, the vith through $vi+v-1$st vertices (in that order) determine a patch for each $i = 0,1,\dots n-1$, where there are $vn+k$ vertices. k is in the range $[0, v-1]$; if k is not zero, the final k vertices are ignored.

18.1.12 General Considerations For Polygon Primitives

Depending on the polygon mode, a *polygon primitive* generated from a drawing command with *topology* VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_

ADJACENCY, or VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY is rendered in one of several ways, such as outlining its border or filling its interior. The order of vertices in such a primitive is significant during polygon rasterization and fragment shading.

18.1.13 Programmable Primitive Shading

Once primitives are assembled, they proceed to the vertex shading stage of the pipeline. If the draw includes multiple instances, then the set of primitives is sent to the vertex shading stage multiple times, once for each instance.

It is undefined whether vertex shading occurs on vertices that are discarded as part of incomplete primitives, but if it does occur then it operates as if they were vertices in complete primitives and such invocations can have side effects.

Vertex shading receives two per-vertex inputs from the primitive assembly stage - the vertexIndex and the instanceIndex. How these values are generated is defined below, with each command.

Drawing commands fall roughly into two categories:

- Non-indexed drawing (**`vkCmdDraw`** and **`vkCmdDrawIndirect`**) commands present a sequential vertexIndex to the vertex shader. The sequential index is generated automatically by the device.
- Indexed drawing commands (**`vkCmdDrawIndexed`** and **`vkCmdDrawIndexed`**) read index values from an *index buffer* and use this to compute the vertexIndex value for the vertex shader.

An index buffer is bound to a command buffer by calling:

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

`buffer` is the handle of the buffer being bound, and *`offset`* is the starting offset in bytes within *`buffer`* used in index buffer address calculations. *`commandbuffer`* is the command buffer that records the command, and *`indexType`* is an enum of type `VkIndexType` that selects whether indices are treated as 16 bits or 32 bits:

```
typedef enum VkIndexType {
    VK_INDEX_TYPE_UINT16 = 0,
    VK_INDEX_TYPE_UINT32 = 1,
} VkIndexType;
```

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`buffer`* must be a valid VkBuffer handle
- *`indexType`* must be a valid `VkIndexType` value
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support graphics operations

- Each of *commandBuffer* and *buffer* must have been created, allocated or retrieved from the same VkDevice
- The value of *offset* must be less than the size of *buffer*
- The sum of *offset*, and the address of the range of VkDeviceMemory object that's backing *buffer*, must be a multiple of the type indicated by *indexType*
- *buffer* must have been created with the VK_BUFFER_USAGE_INDEX_BUFFER_BIT flag

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

The parameters for each drawing command are specified directly in the command or read from buffer memory, depending on the command. Drawing commands that source their parameters from buffer memory are known as *indirect* drawing commands.

All drawing commands interact with the Robust Buffer Access feature.

Primitives assembled by draw commands are considered to have an API order, which defines the order their fragments affect the framebuffer. When a draw command includes multiple instances, the lower numbered instances are earlier in API order. For non-indexed draws, primitives with lower numbered vertexIndex values are earlier in API order. For indexed draws, primitives assembled from lower index buffer addresses are earlier in API order.

To record a non-indexed draw, call:

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

commandbuffer is the command buffer which records the command. When the command is executed, primitives are assembled using *vertexCount* vertices, using consecutive indices with the first vertexIndex value equal to *firstVertex*, and using the current primitive topology. The assembled primitives execute the currently bound graphics pipeline *instanceCount* times, with instanceIndex value starting at *firstInstance* and increasing sequentially for each instance.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *commandBuffer* must be in the recording state

- The VkCommandPool that `commandBuffer` was allocated from must support graphics operations
- This command must only be called inside of a render pass instance
- For each set *n* that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS, a descriptor set must have been bound to *n* at VK_PIPELINE_BIND_POINT_GRAPHICS, with a VkPipelineLayout that is compatible for set *n*, with the VkPipelineLayout used to create the current VkPipeline, as described in Section 13.2.2.1
- For each push constant that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS, a push constant value must have been set for VK_PIPELINE_BIND_POINT_GRAPHICS, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Section 13.2.2.1
- Descriptors in each bound descriptor set, specified via **`vkCmdBindDescriptorSets`**, must be valid if they are used by the currently bound VkPipeline object, specified via **`vkCmdBindPipeline`**
- Any vertex buffer bindings referenced by the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS must have valid buffers bound to them
- For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Section 19.2
- A valid graphics pipeline must be bound to the current command buffer with VK_PIPELINE_BIND_POINT_GRAPHICS
- If the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS requires any dynamic state, that state must have been set on the current command buffer
- Every input attachment referenced by the current subpass must be bound to the pipeline via a descriptor set
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with "ImplicitLod", "Dref" or "Proj" in their name, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a lod bias or any offset values, in any shader stage
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a uniform buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a storage buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

To record an indexed draw, call:

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

commandbuffer is the command buffer which records the command. When the command is executed, primitives are assembled using *indexCount* vertices using vertex indices retrieved from the index buffer, and using the current primitive topology. The index buffer is treated as an array of tightly packed unsigned integers of size defined by the `VkIndexType`. The first index value is at an offset of *firstIndex* * indexSize + *offset* within the currently bound index buffer, where *offset* is the offset specified by **vkCmdBindIndexBuffer** and indexSize is that of the type specified by *indexType*. Subsequent index values are retrieved from consecutive locations in the index buffer. Indices are first compared to the primitive restart value, then zero extended to 32 bits (if VK_INDEX_TYPE_UINT16), then have *vertexOffset* added to them, before being supplied as the vertexIndex value. The assembled primitives execute the currently bound graphics pipeline *instanceCount* times, with instanceIndex value starting at *firstInstance* and increasing sequentially for each instance.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- This command must only be called inside of a render pass instance
- For each set *n* that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS, a descriptor set must have been bound to *n* at VK_PIPELINE_BIND_POINT_GRAPHICS, with a VkPipelineLayout that is compatible for set *n*, with the VkPipelineLayout used to create the current VkPipeline, as described in Section 13.2.2.1
- For each push constant that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS, a push constant value must have been set for VK_PIPELINE_BIND_POINT_GRAPHICS, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Section 13.2.2.1

- Descriptors in each bound descriptor set, specified via **`vkCmdBindDescriptorSets`**, must be valid if they are used by the currently bound VkPipeline object, specified via **`vkCmdBindPipeline`**
- Any vertex buffer bindings referenced by the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS must have valid buffers bound to them
- For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Section 19.2
- A valid graphics pipeline must be bound to the current command buffer with VK_PIPELINE_BIND_POINT_GRAPHICS
- If the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS requires any dynamic state, that state must have been set on the current command buffer
- The total value of (*`indexSize`* * (*`firstIndex`* + *`indexCount`*) + *`offset`*) must be less than or equal to the size of the currently bound index buffer, with indexSize being based on the type specified by *`indexType`*, where the index buffer, *`indexType`*, and *`offset`* are specified via **`vkCmdBindIndexBuffer`**
- Every input attachment referenced by the current subpass must be bound to the pipeline via a descriptor set
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with "ImplicitLod", "Dref" or "Proj" in their name, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a lod bias or any offset values, in any shader stage
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a uniform buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a storage buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set

Host Synchronization

- Host access to *`commandBuffer`* must be externally synchronized

A non-indexed indirect draw is recorded by calling:

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

This command behaves similarly to **`vkCmdDraw`** except that the parameters are read by the device from a buffer during execution. Zero or more draws are executed by the command, where the number of draws is specified by *`drawCount`*. The buffer containing the parameters is specified by *`buffer`* and the byte offset into the buffer where the parameters begin is specified in *`offset`*. At this offset, the parameters of the draw are encoded in an array of VkDrawIndirectCommand structures separated by *`stride`* bytes. If *`stride`* is zero, then the structures are considered to be tightly packed.

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`buffer`* must be a valid VkBuffer handle
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support graphics operations
- This command must only be called inside of a render pass instance
- Each of *`commandBuffer`* and *`buffer`* must have been created, allocated or retrieved from the same VkDevice
- If the multi-draw indirect feature is not enabled, the value of *`drawCount`* must be `0` or `1`
- If the drawIndirectFirstInstance feature is not enabled, all the *`firstInstance`* members of the VkDrawIndirectCommand structures accessed by this command must be **`0`**
- For each set *n* that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS, a descriptor set must have been bound to *n* at VK_PIPELINE_BIND_POINT_GRAPHICS, with a VkPipelineLayout that is compatible for set *n*, with the VkPipelineLayout used to create the current VkPipeline, as described in Section 13.2.2.1
- For each push constant that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS, a push constant value must have been set for VK_PIPELINE_BIND_POINT_GRAPHICS, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Section 13.2.2.1
- Descriptors in each bound descriptor set, specified via **`vkCmdBindDescriptorSets`**, must be valid if they are used by the currently bound VkPipeline object, specified via **`vkCmdBindPipeline`**
- Any vertex buffer bindings referenced by the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS must have valid buffers bound to them
- A valid graphics pipeline must be bound to the current command buffer with VK_PIPELINE_BIND_POINT_GRAPHICS

- If the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS requires any dynamic state, that state must have been set on the current command buffer
- If `drawCount` is greater than `0`, the total value of (`stride` * (`drawCount` - 1) + `offset` + sizeof(VkDrawIndirectCommand)) must be less than or equal to the size of `buffer`
- `drawCount` must be less than or equal to VkPhysicalDeviceLimits::`maxDrawIndirectCount`
- Every input attachment referenced by the current subpass must be bound to the pipeline via a descriptor set
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with "ImplicitLod", "Dref" or "Proj" in their name, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a lod bias or any offset values, in any shader stage
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a uniform buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a storage buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

The definition of VkDrawIndirectCommand is:

```
typedef struct VkDrawIndirectCommand {
    uint32_t                                    vertexCount;
    uint32_t                                    instanceCount;
    uint32_t                                    firstVertex;
    uint32_t                                    firstInstance;
} VkDrawIndirectCommand;
```

The members of VkDrawIndirectCommand have the same meaning as the similarly named parameters of `vkCmdDraw`.

Valid Usage

- For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Section 19.2
- If the drawIndirectFirstInstance feature is not enabled, *`firstInstance`* must be **`0`**

An indexed indirect draw is recorded by calling:

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

As with **`vkCmdDrawIndirect`**, *`buffer`* is the buffer containing the parameters and *`offset`* is the byte offset within the buffer where the parameters begin. The parameters for each of *`drawCount`* draws is contained in an instance of VkDrawIndexedIndirectCommand structure, each separated by *`stride`* bytes.

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`buffer`* must be a valid VkBuffer handle
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support graphics operations
- This command must only be called inside of a render pass instance
- Each of *`commandBuffer`* and *`buffer`* must have been created, allocated or retrieved from the same VkDevice
- If the multi-draw indirect feature is not enabled, the value of *`drawCount`* must be `0` or `1`
- If the drawIndirectFirstInstance feature is not enabled, all the *`firstInstance`* members of the VkDrawIndexedIndirectCommand structures accessed by this command must be **`0`**
- For each set *n* that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS, a descriptor set must have been bound to *n* at VK_PIPELINE_BIND_POINT_GRAPHICS, with a VkPipelineLayout that is compatible for set *n*, with the VkPipelineLayout used to create the current VkPipeline, as described in Section 13.2.2.1

- For each push constant that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS, a push constant value must have been set for VK_PIPELINE_BIND_POINT_GRAPHICS, with a VkPipelineLayout that is compatible for push constants, with the VkPipelineLayout used to create the current VkPipeline, as described in Section 13.2.2.1
- Descriptors in each bound descriptor set, specified via **`vkCmdBindDescriptorSets`**, must be valid if they are used by the currently bound VkPipeline object, specified via **`vkCmdBindPipeline`**
- Any vertex buffer bindings referenced by the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS must have valid buffers bound to them
- A valid graphics pipeline must be bound to the current command buffer with VK_PIPELINE_BIND_POINT_GRAPHICS
- If the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS requires any dynamic state, that state must have been set on the current command buffer
- If *`drawCount`* is greater than `0`, the total value of (*`stride`* * (*`drawCount`* - 1) + *`offset`* + sizeof(VkDrawIndexedIndirectCommand)) must be less than or equal to the size of *`buffer`*
- *`drawCount`* must be less than or equal to VkPhysicalDeviceLimits::*`maxDrawIndirectCount`*
- Every input attachment referenced by the current subpass must be bound to the pipeline via a descriptor set
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with "ImplicitLod", "Dref" or "Proj" in their name, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a lod bias or any offset values, in any shader stage
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a uniform buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_GRAPHICS accesses a storage buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

The definition of VkDrawIndexedIndirectCommand is

```
typedef struct VkDrawIndexedIndirectCommand {
    uint32_t                                          indexCount;
    uint32_t                                          instanceCount;
    uint32_t                                          firstIndex;
    int32_t                                           vertexOffset;
    uint32_t                                          firstInstance;
} VkDrawIndexedIndirectCommand;
```

The members of VkDrawIndexedIndirectCommand have the same meaning as the similarly named parameters of `vkCmdDrawIndexed`.

Valid Usage

- For a given vertex buffer binding, any attribute data fetched must be entirely contained within the corresponding vertex buffer binding, as described in Section 19.2
- The total value of (*indexSize* * (*firstIndex* + *indexCount*) + *offset*) must be less than or equal to the size of the currently bound index buffer, with indexSize being based on the type specified by *indexType*, where the index buffer, *indexType*, and *offset* are specified via **`vkCmdBindIndexBuffer`**
- If the drawIndirectFirstInstance feature is not enabled, *firstInstance* must be **`0`**

Chapter 19

Fixed-Function Vertex Processing

Some implementations have specialized fixed-function hardware for fetching and format-converting vertex input data from buffers, rather than performing the fetch as part of the vertex shader. Vulkan includes a vertex attribute fetch stage in the graphics pipeline in order to take advantage of this.

19.1 Vertex Attributes

Vertex shaders can define input variables, which receive *vertex attribute* data transferred from one or more VkBuffer(s) by drawing commands. Vertex shader input variables are bound to buffers via an indirect binding where the vertex shader associates a *vertex input attribute* number with each variable, vertex input attributes are associated to *vertex input bindings* on a per-pipeline basis, and vertex input bindings are associated with specific buffers on a per-draw basis via the **`vkCmdBindVertexBuffers`** command. Vertex input attribute and vertex input binding descriptions also contain format information controlling how data is extracted from buffer memory and converted to the format expected by the vertex shader.

There are VkPhysicalDeviceLimits::*`maxVertexInputAttributes`* number of vertex input attributes and VkPhysicalDeviceLimits::*`maxVertexInputBindings`* (each referred to by zero-based indices), where there are at least as many vertex input attributes as there are vertex input bindings. Applications can store multiple vertex input attributes interleaved in a single buffer, and use a single vertex input binding to access those attributes.

In GLSL, vertex shaders associate input variables with a vertex input attribute number using the **`location`** layout qualifier. The **`component`** layout qualifier associates components of a vertex shader input variable with components of a vertex input attribute.

GLSL example

```
// Assign location N to variableName
layout (location=N) in typeName variableName;

// Assign locations [N,N+L) to the array elements of variableName
layout (location=N) in typeName variableName[L];
```

In SPIR-V, vertex shaders associate input variables with a vertex input attribute number using the **`Location`** decoration. The **`Component`** decoration associates components of a vertex shader input variable with components of a vertex input attribute. The **`Location`** and **`Component`** decorations are specified via the **`OpDecorate`** instruction.

SPIR-V example

19.1.1 Attribute Location and Component Assignment

Vertex shaders allow **`Location`** and **`Component`** decorations on input variable declarations. The **`Location`** decoration specifies which vertex input attribute is used to read and interpret the data that a variable will consume. The **`Component`** decoration allows the location to be more finely specified for scalars and vectors, down to the individual components within a location that are consumed. The components within a location are 0, 1, 2, and 3. A variable starting at component N will consume components N, N+1, N+2, … up through its size. For single precision types, it is invalid if the sequence of components gets larger than 3.

When a vertex shader input variable declared using a scalar or vector 32-bit data type is assigned a location, its value(s) are taken from the components of the input attribute specified with the corresponding VkVertexInputAttributeDescription::*`location`*. The components used depend on the type of variable and the value of the **`Component`** decoration specified in the variable declaration, as identified in Table 19.1. Any 32-bit scalar or vector input will consume a single location. For 32-bit data types, missing components are filled in with default values as described below.

Table 19.1: Input attribute components accessed by 32-bit input variables

32-bit data type	**`Component` decoration**	**Components consumed**
scalar	0 or unspecified	(x, o, o, o)
scalar	1	(o, y, o, o)
scalar	2	(o, o, z, o)
scalar	3	(o, o, o, w)
two-component vector	0 or unspecified	(x, y, o, o)
two-component vector	1	(o, y, z, o)
two-component vector	2	(o, o, z, w)
three-component vector	0 or unspecified	(x, y, z, o)
three-component vector	1	(o, y, z, w)
four-component vector	0 or unspecified	(x, y, z, w)

Components indicated by ‘o’ are available for use by other input variables which are sourced from the same attribute, and if used, are either filled with the corresponding component from the input format (if present), or the default value.

When a vertex shader input variable declared using a 32-bit floating point matrix type is assigned a location *i*, its values are taken from consecutive input attributes starting with the corresponding VkVertexInputAttributeDescription::*`location`*. Such matrices are treated as an array of column vectors with values taken from the input attributes identified in Table 19.2. The VkVertexInputAttributeDescription::*`format`* must be specified with a `VkFormat` that corresponds to the appropriate type of column vector. The **`Component`** decoration must not be used with matrix types.

Table 19.2: Input attributes accessed by 32-bit input matrix variables

Data type	**Column vector type**	**Locations consumed**	**Components consumed**
mat2	two-component vector	i, i+1	(x, y, o, o), (x, y, o, o)
mat2x3	three-component vector	i, i+1	(x, y, z, o), (x, y, z, o)
mat2x4	four-component vector	i, i+1	(x, y, z, w), (x, y, z, w)
mat3x2	two-component vector	i, i+1, i+2	(x, y, o, o), (x, y, o, o), (x, y, o, o)
mat3	three-component vector	i, i+1, i+2	(x, y, z, o), (x, y, z, o), (x, y, z, o)
mat3x4	four-component vector	i, i+1, i+2	(x, y, z, w), (x, y, z, w), (x, y, z, w)
mat4x2	two-component vector	i, i+1, i+2, i+3	(x, y, o, o), (x, y, o, o), (x, y, o, o), (x, y, o, o)
mat4x3	three-component vector	i, i+1, i+2, i+3	(x, y, z, o), (x, y, z, o), (x, y, z, o), (x, y, z, o)
mat4	four-component vector	i, i+1, i+2, i+3	(x, y, z, w), (x, y, z, w), (x, y, z, w), (x, y, z, w)

Components indicated by ‘o’ are available for use by other input variables which are sourced from the same attribute, and if used, are either filled with the corresponding component from the input (if present), or the default value.

When a vertex shader input variable declared using a scalar or vector 64-bit data type is assigned a location *i*, its values are taken from consecutive input attributes starting with the corresponding VkVertexInputAttributeDescription::`location`. The locations and components used depend on the type of variable and the **`Component`** decoration specified in the variable declaration, as identified in Table 19.3. For 64-bit data types, no default attribute values are provided. Input variables must not use more components than provided by the attribute. Input attributes which have one- or two-component 64-bit formats will consume a single location. Input attributes which have three- or four-component 64-bit formats will consume two consecutive locations. A 64-bit scalar data type will consume two components, and a 64-bit two-component vector data type will consume all four components available within a location. A three- or four-component 64-bit data type must not specify a component. A three-component 64-bit data type will consume all four components of the first location and components 0 and 1 of the second location. This leaves components 2 and 3 available for other component-qualified declarations. A four-component 64-bit data type will consume all four components of the first location and all four components of the second location. It is invalid for a scalar or two-component 64-bit data type to specify a component of 1 or 3.

Table 19.3: Input attribute locations and components accessed by 64-bit input variables

Input format	**Locations consumed**	**64-bit data type**	**`Location` decoration**	**`Component` decoration**	**32-bit components consumed**
R64	i	scalar	i	0 or unspecified	(x, y, -, -)
R64G64	i	scalar	i	0 or unspecified	(x, y, o, o)

Table 19.3: (continued)

Input format	Locations consumed	64-bit data type	`Location` decoration	`Component` decoration	32-bit components consumed
		scalar	i	2	(o, o, z, w)
		two-component vector	i	0 or unspecified	(x, y, z, w)
R64G64B64	i, i+1	scalar	i	0 or unspecified	(x, y, o, o), (o, o, -, -)
		scalar	i	2	(o, o, z, w), (o, o, -, -)
		scalar	i+1	0 or unspecified	(o, o, o, o), (x, y, -, -)
		two-component vector	i	0 or unspecified	(x, y, z, w), (o, o, -, -)
		three-component vector	i	unspecified	(x, y, z, w), (x, y, -, -)
R64G64B64A64	i, i+1	scalar	i	0 or unspecified	(x, y, o, o), (o, o, o, o)
		scalar	i	2	(o, o, z, w), (o, o, o, o)
		scalar	i+1	0 or unspecified	(o, o, o, o), (x, y, o, o)
		scalar	i+1	2	(o, o, o, o), (o, o, z, w)
		two-component vector	i	0 or unspecified	(x, y, z, w), (o, o, o, o)
		two-component vector	i+1	0 or unspecified	(o, o, o, o), (x, y, z, w)
		three-component vector	i	unspecified	(x, y, z, w), (x, y, o, o)
		four-component vector	i	unspecified	(x, y, z, w), (x, y, z, w)

Components indicated by ‘o’ are available for use by other input variables which are sourced from the same attribute. Components indicated by ‘-’ are not available for input variables as there are no default values provided for 64-bit data types, and there is no data provided by the input format.

When a vertex shader input variable declared using a 64-bit floating-point matrix type is assigned a location *i*, its values are taken from consecutive input attribute locations. Such matrices are treated as an array of column vectors with values taken from the input attributes as shown in Table 19.3. Each column vector starts at the location immediately following the last location of the previous column vector. The number of attributes and components assigned to each matrix is determined by the matrix dimensions and ranges from two to eight locations.

When a vertex shader input variable declared using an array type is assigned a location, its values are taken from consecutive input attributes starting with the corresponding VkVertexInputAttributeDescription::*`location`*. The number of attributes and components assigned to each element are determined according to the data type of the array elements and **`Component`** decoration (if any) specified in the declaration of the array, as described above. Each element of the array, in order, is assigned to consecutive locations, but all at the same specified component within each location.

Only input variables declared with the data types and component decorations as specified above are supported. *Location aliasing* is causing two variables to have the same location number. *Component aliasing* is assigning the same (or overlapping) component number for two location aliases. Location aliasing is allowed only if it does not cause compenent aliasing. Further, when location aliasing, the aliases sharing the location must have the same underlying numerical type (floating-point or integer). Failure to meet these requirements will result in an invalid pipeline.

19.2 Vertex Input Description

Applications specify vertex input attribute and vertex input binding descriptions as part of graphics pipeline creation, via the *pVertexInputState* member of VkGraphicsPipelineCreateInfo, which is of type VkPipelineVertexInputStateCreateInfo:

```
typedef struct VkPipelineVertexInputStateCreateInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    VkPipelineVertexInputStateCreateFlags       flags;
    uint32_t                                    vertexBindingDescriptionCount;
    const VkVertexInputBindingDescription*      pVertexBindingDescriptions;
    uint32_t                                    vertexAttributeDescriptionCount;
    const VkVertexInputAttributeDescription*    pVertexAttributeDescriptions;
} VkPipelineVertexInputStateCreateInfo;
```

The members of VkPipelineVertexInputStateCreateInfo have the following meanings:

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *vertexBindingDescriptionCount* is the number of vertex binding descriptions provided in *pVertexBindingDescriptions*.
- *pVertexBindingDescriptions* is a pointer to an array of VkVertexInputBindingDescription structures.
- *vertexAttributeDescriptionCount* is the number of vertex attribute descriptions provided in *pVertexAttributeDescriptions*.
- *pVertexAttributeDescriptions* is a pointer to an array of VkVertexInputAttributeDescription structures.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`

- If `vertexBindingDescriptionCount` is not `0`, `pVertexBindingDescriptions` must be a pointer to an array of `vertexBindingDescriptionCount` valid VkVertexInputBindingDescription structures
- If `vertexAttributeDescriptionCount` is not `0`, `pVertexAttributeDescriptions` must be a pointer to an array of `vertexAttributeDescriptionCount` valid VkVertexInputAttributeDescription structures
- `vertexBindingDescriptionCount` must be less than or equal to VkPhysicalDeviceLimits::`maxVertexInputBindings`
- `vertexAttributeDescriptionCount` must be less than or equal to VkPhysicalDeviceLimits::`maxVertexInputAttributes`
- For every value of `binding` specified by any given element of `pVertexAttributeDescriptions`, a VkVertexInputBindingDescription must exist in `pVertexBindingDescriptions` with the same value of `binding`
- All elements of `pVertexBindingDescriptions` must describe distinct binding numbers
- All elements of `pVertexAttributeDescriptions` must describe distinct attribute locations

Each vertex input binding is specified by an instance of the VkVertexInputBindingDescription structure:

```
typedef struct VkVertexInputBindingDescription {
    uint32_t                                    binding;
    uint32_t                                    stride;
    VkVertexInputRate                           inputRate;
} VkVertexInputBindingDescription;
```

The members of VkVertexInputBindingDescription have the following meanings:

- `binding` is the binding number that this structure describes.
- `stride` is the distance in bytes between two consecutive elements within the buffer.
- `inputRate` is a `VkVertexInputRate` value that specifies whether vertex attribute addressing is a function of the vertex index or of the instance index.

Valid Usage

- `inputRate` must be a valid `VkVertexInputRate` value
- `binding` must be less than or equal to VkPhysicalDeviceLimits::`maxVertexInputBindings`
- `stride` must be less than or equal to VkPhysicalDeviceLimits::`maxVertexInputBindingStride`

The definition of `VkVertexInputRate` is:

```
typedef enum VkVertexInputRate {
    VK_VERTEX_INPUT_RATE_VERTEX = 0,
    VK_VERTEX_INPUT_RATE_INSTANCE = 1,
} VkVertexInputRate;
```

The values of `VkVertexInputRate` have the following meanings:

- VK_VERTEX_INPUT_RATE_VERTEX indicates that vertex attribute addressing is a function of the vertex index.
- VK_VERTEX_INPUT_RATE_INSTANCE indicates that vertex attribute addressing is a function of the instance index.

Each vertex input attribute is specified by an instance of the VkVertexInputAttributeDescription structure:

```
typedef struct VkVertexInputAttributeDescription {
    uint32_t                                    location;
    uint32_t                                    binding;
    VkFormat                                    format;
    uint32_t                                    offset;
} VkVertexInputAttributeDescription;
```

The members of VkVertexInputAttributeDescription have the following meanings:

- *location* is the shader binding location number for this attribute.
- *binding* is the binding number which this attribute takes its data from.
- *format* is the size and type of the vertex attribute data.
- *offset* is a byte offset of this attribute relative to the start of an element in the vertex input binding.

Valid Usage

- *format* must be a valid `VkFormat` value
- *binding* must be less than VkPhysicalDeviceLimits::*maxVertexInputBindings*
- *offset* must be less than or equal to VkPhysicalDeviceLimits::*maxVertexInputAttributeOffset*
- *format* must be allowed as a vertex buffer format, as specified by the VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT flag in VkFormatProperties::*bufferFeatures* returned by **vkGetPhysicalDeviceFormatProperties**

Vertex buffers are bound to a command buffer for use in subsequent draw commands via the command

```
void vkCmdBindVertexBuffers(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    firstBinding,
    uint32_t                                    bindingCount,
    const VkBuffer*                             pBuffers,
    const VkDeviceSize*                         pOffsets);
```

- `commandbuffer` is the command buffer that records the command.
- `firstBinding` is the index of the first vertex input binding whose state is updated by the command.
- `bindingCount` is the number of vertex input bindings whose state is updated by the command.
- `pBuffers` is a pointer to an array of buffer handles.
- `pOffsets` is a pointer to an array of buffer offsets.

The values taken from elements *i* of `pBuffers` and `pOffsets` replace the current state for the vertex input binding $firstBinding + i$, for *i* in $[0, bindingCount)$. The vertex input binding is updated to start at the offset indicated by `pOffsets`[i] from the start of the buffer `pBuffers`[i]. All vertex input attributes that use each of these bindings will use these updated addresses in their address calculations for subsequent draw commands.

Valid Usage

- `commandBuffer` must be a valid VkCommandBuffer handle
- `pBuffers` must be a pointer to an array of `bindingCount` valid VkBuffer handles
- `pOffsets` must be a pointer to an array of `bindingCount` VkDeviceSize values
- `commandBuffer` must be in the recording state
- The VkCommandPool that `commandBuffer` was allocated from must support graphics operations
- The value of `bindingCount` must be greater than `0`
- Each of `commandBuffer` and the elements of `pBuffers` must have been created, allocated or retrieved from the same VkDevice
- `firstBinding` must be less than VkPhysicalDeviceLimits::`maxVertexInputBindings`
- The sum of `firstBinding` and `bindingCount` must be less than or equal to VkPhysicalDeviceLimits::`maxVertexInputBindings`
- All elements of `pOffsets` must be less than or equal to the size of the corresponding element in `pBuffers`
- All elements of `pBuffers` must have been created with the VK_BUFFER_USAGE_VERTEX_BUFFER_BIT flag

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

The address of each attribute for each vertexIndex/instanceIndex is calculated as follows:

- Let attribDesc be the member of VkPipelineVertexInputStateCreateInfo::*pVertexAttributeDescriptions* with VkVertexInputAttributeDescription::*location* equal to the vertex input attribute number.
- Let bindingDesc be the member of VkPipelineVertexInputStateCreateInfo::*pVertexBindingDescriptions* with VkVertexInputAttributeDescription::*binding* equal to attribDesc.binding.
- Let vertexIndex be the index of the vertex within the draw (a value between *firstVertex* and *firstVertex*+*vertexCount* for **vkCmdDraw**, or a value taken from the index buffer for **vkCmdDrawIndexed**), and let instanceIndex be the instance number of the draw (a value between *firstInstance* and *firstInstance*+*instanceCount*).

```
bufferBindingAddress = buffer[binding].baseAddress + offset[binding];

if (bindingDesc.inputRate == VK_VERTEX_INPUT_RATE_VERTEX)
    vertexOffset = vertexIndex * bindingDesc.stride;
else
    vertexOffset = instanceIndex * bindingDesc.stride;

attribAddress = bufferBindingAddress + vertexOffset + attribDesc.offset;
```

For each attribute, raw data is extracted starting at `attribAddress` and is converted from the VkVertexInputAttributeDescription's *format* to either to floating-point, unsigned integer, or signed integer based on the base type of the format; the base type of the format must match the base type of the input variable in the shader. If *format* is a packed format, `attribAddress` must be a multiple of the size in bytes of the whole attribute data type as described in Packed Formats. Otherwise, `attribAddress` must be a multiple of the size in bytes of the component type indicated by *format* (see Formats). If the format does not include G, B, or A components, then those are filled with (0,0,1) as needed (using either 1.0f or integer 1 based on the format) for attributes that are not 64-bit data types. The number of components in the vertex shader input variable need not exactly match the number of components in the format. If the vertex shader has fewer components, the extra components are discarded.

19.3 Example

To create a graphics pipeline that uses the following vertex description:

```
struct Vertex
{
    float   x, y, z, w;
    uint8_t u, v;
};
```

The application could use the following set of structures:

```
const VkVertexInputBindingDescription binding =
{
    0,                                              // binding
    sizeof(Vertex),                                 // stride
    VK_VERTEX_INPUT_RATE_VERTEX                     // inputRate
};

const VkVertexInputAttributeDescription attributes[] =
{
    {
        0,                                          // location
        binding.binding,                            // binding
        VK_FORMAT_R32G32B32A32_SFLOAT,          // format
        0                                       // offset
    },
    {
        1,                                      // location
        binding.binding,                        // binding
        VK_FORMAT_R8G8_UNORM,                   // format
        4 * sizeof(float)                       // offset
    }
};

const VkPipelineVertexInputStateCreateInfo viInfo =
{
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_CREATE_INFO,    // sType
    NULL,                                                   // pNext
    0,                                                      // flags
    1,                                                      //  ↩
        vertexBindingDescriptionCount
    &binding,                                               //  ↩
        pVertexBindingDescriptions
    2,                                                      //  ↩
        vertexAttributeDescriptionCount
    &attributes[0]                                          //  ↩
        pVertexAttributeDescriptions
};
```

Chapter 20

Tessellation

Tessellation involves three pipeline stages. First, a tessellation control shader transforms control points of a patch and can produce per-patch data. Second, a fixed-function tessellator generates multiple primitives corresponding to a tessellation of the patch in (u,v) or (u,v,w) parameter space. Third, a tessellation evaluation shader transforms the vertices of the tessellated patch, for example to compute their positions and attributes as part of the tessellated surface. The tessellator is enabled when the pipeline contains both a tessellation control shader and a tessellation evaluation shader.

20.1 Tessellator

If a pipeline includes both tessellation shaders (control and evaluation), the tessellator consumes each input patch (after vertex shading) and produces a new set of independent primitives (points, lines, or triangles). These primitives are logically produced by subdividing a geometric primitive (rectangle or triangle) according to the per-patch tessellation levels written by the tessellation control shader. This subdivision is performed in an implementation-dependent manner. If no tessellation shaders are present in the pipeline, the tessellator is disabled and incoming primitives are passed through without modification.

The type of subdivision performed by the tessellator is specified by an **`OpExecutionMode`** instruction in the tessellation evaluation or tessellation control shader using one of execution modes **`Triangles`**, **`Quads`**, and **`IsoLines`**. Other tessellation-related execution modes can also be specified in either the tessellation control or tessellation evaluation shaders, and if they are specified in both then the modes must be the same.

Tessellation execution modes include:

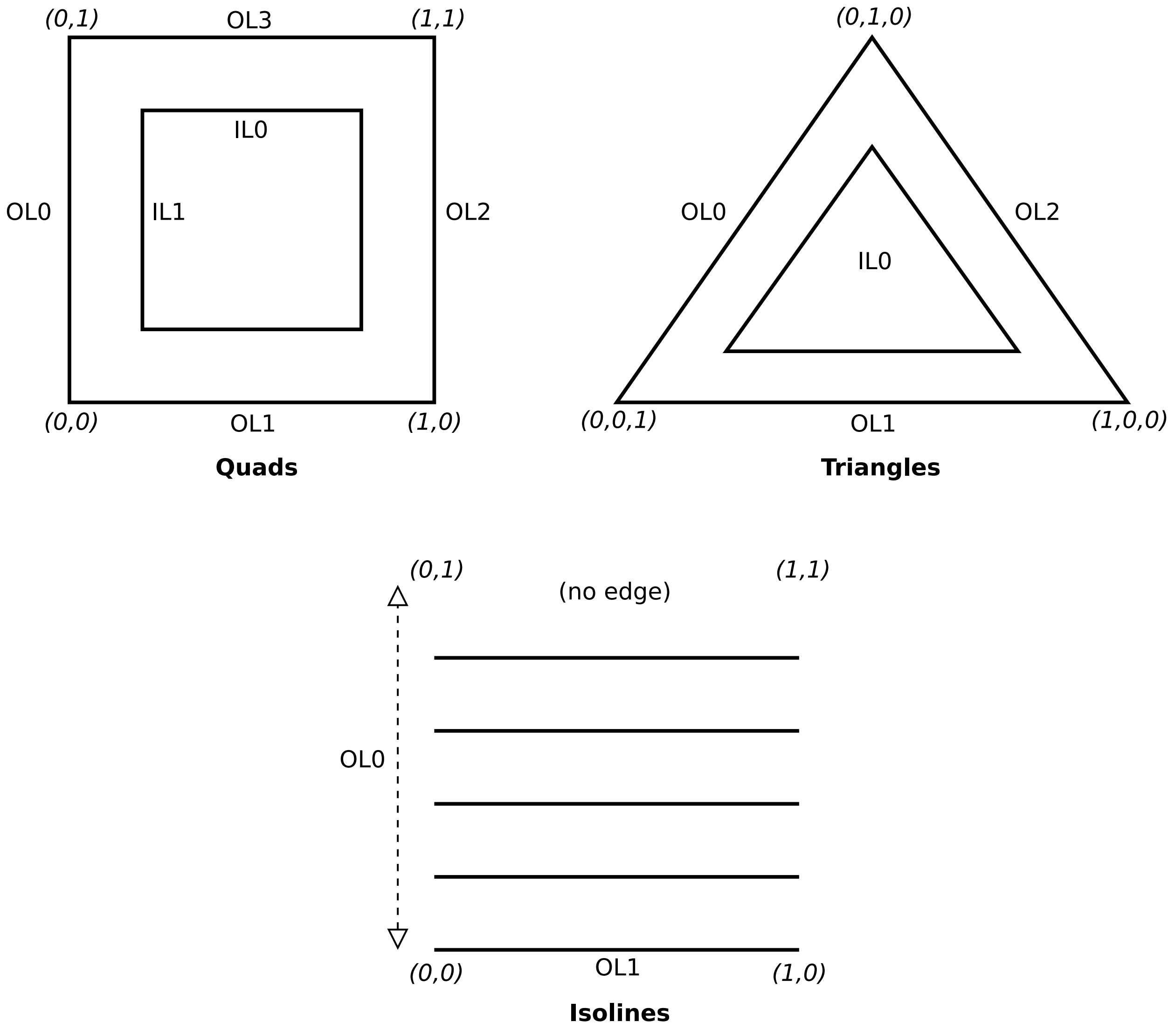
- **`Triangles`**, **`Quads`**, and **`IsoLines`**. These control the type of subdivision and topology of the output primitives. One mode must be set in at least one of the tessellation shader stages.
- **`VertexOrderCw`** and **`VertexOrderCcw`**. These control the orientation of triangles generated by the tessellator. One mode must be set in at least one of the tessellation shader stages.
- **`PointMode`**. Controls generation of points rather than triangles or lines. This functionality defaults to disabled, and is enabled if either shader stage includes the execution mode.
- **`SpacingEqual`**, **`SpacingFractionalEven`**, and **`SpacingFractionalOdd`**. Controls the spacing of segments on the edges of tessellated primitives. One mode must be set in at least one of the tessellation shader stages.

- **`OutputVertices`**. Controls the size of the output patch of the tessellation control shader. One value must be set in at least one of the tessellation shader stages.

For triangles, the tessellator subdivides a triangle primitive into smaller triangles. For quads, the tessellator subdivides a rectangle primitive into smaller triangles. For isolines, the tessellator subdivides a rectangle primitive into a collection of line segments arranged in strips stretching across the rectangle in the *u* dimension (i.e. the coordinates in **`TessCoord`** are of the form (0,x) through (1,x) for all tessellation evaluation shader invocations that share a line).

Each vertex produced by the tessellator has an associated (u,v,w) or (u,v) position in a normalized parameter space, with parameter values in the $[0, 1]$, as illustrated in figure Domain parameterization for tessellation primitive modes.

Domain parameterization for tessellation primitive modes



For triangles, the vertex's position is a barycentric coordinate (u,v,w), where u + v + w = 1.0, and indicates the relative influence of the three vertices of the triangle on the position of the vertex. For quads and isolines, the position is a (u,v) coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle. The subdivision process is explained in more detail in subsequent sections.

20.2 Tessellator Patch Discard

A patch is discarded by the tessellator if any relevant outer tessellation level is less than or equal to zero.

Patches will also be discarded if any relevant outer tessellation level corresponds to a floating-point NaN (not a number) in implementations supporting NaN.

When patches are discarded, no new primitives are generated and the tessellation evaluation shader is not executed. For **`Quads`**, all four outer levels are relevant. For **`Triangles`** and **`IsoLines`**, only the first three or two outer levels, respectively, are relevant. Negative inner levels will not cause a patch to be discarded; they will be clamped as described below.

20.3 Tessellator Spacing

Each of the tessellation levels is used to determine the number and spacing of segments used to subdivide a corresponding edge. The method used to derive the number and spacing of segments is specified by an **`OpExecutionMode`** in the tessellation control or tessellation evaluation shader using one of the identifiers **`SpacingEqual`**, **`SpacingFractionalEven`**, or **`SpacingFractionalOdd`**.

If **`SpacingEqual`** is used, the floating-point tessellation level is first clamped to $[1, maxLevel]$, where $maxLevel$ is the implementation-dependent maximum tessellation level (VkPhysicalDeviceLimits::*`maxTessellationGenerationLevel`*). The result is rounded up to the nearest integer n, and the corresponding edge is divided into n segments of equal length in (u,v) space.

If **`SpacingFractionalEven`** is used, the tessellation level is first clamped to $[2, maxLevel]$ and then rounded up to the nearest even integer n. If **`SpacingFractionalOdd`** is used, the tessellation level is clamped to $[1, maxLevel - 1]$ and then rounded up to the nearest odd integer n. If n is one, the edge will not be subdivided. Otherwise, the corresponding edge will be divided into $n-2$ segments of equal length, and two additional segments of equal length that are typically shorter than the other segments. The length of the two additional segments relative to the others will decrease monotonically with the value of $n-f$, where f is the clamped floating-point tessellation level. When $n-f$ is zero, the additional segments will have equal length to the other segments. As $n-f$ approaches 2.0, the relative length of the additional segments approaches zero. The two additional segments should be placed symmetrically on opposite sides of the subdivided edge. The relative location of these two segments is implementation-dependent, but must be identical for any pair of subdivided edges with identical values of <f>.

When the tessellator produces triangles (in the **`Triangles`** or **`Quads`** modes), the orientation of all triangles is specified with an **`OpExecutionMode`** of **`VertexOrderCw`** or **`VertexOrderCcw`** in the tessellation control or tessellation evaluation shaders. If the order is **`VertexOrderCw`**, the vertices of all generated triangles will have a clockwise ordering in (u,v) or (u,v,w) space. If the order is **`VertexOrderCcw`**, the vertices will be generated with counter-clockwise order.

The vertices of a triangle are defined to be in counter-clockwise ordering if the value:

$$a = u_0v_1 - u_1v_0 + u_1v_2 - u_2v_1 + u_2v_0 - u_0v_2$$

is positive, where u_i and v_i are the u and v coordinates in normalized parameter space of the ith vertex of the triangle. If the value a is negative, the vertices of the triangle are defined to be in clockwise ordering.

Note
The value a is proportional (with a positive factor) to the signed area of the triangle.
In **`Triangles`** mode, even though the vertex coordinates have a w value, it does not participate directly in the computation of a, being an affine combination of u and v.

For all primitive modes, the tessellator is capable of generating points instead of lines or triangles. If the tessellation control or tessellation evaluation shader specifies the **`OpExecutionMode PointMode`**, the primitive generator will generate one point for each distinct vertex produced by tessellation. Otherwise, the tessellator will produce a collection of line segments or triangles according to the primitive mode. When tessellating triangles or quads in point mode with fractional odd spacing, the tessellator may produce "interior" vertices that are positioned on the edge of the patch if an inner tessellation level is less than or equal to one. Such vertices are considered distinct from vertices produced by subdividing the outer edge of the patch, even if there are pairs of vertices with identical coordinates.

The points, lines, or triangles produced by the tessellator are passed to subsequent pipeline stages in an implementation-dependent order.

20.4 Triangle Tessellation

If the tessellation primitive mode is **`Triangles`**, an equilateral triangle is subdivided into a collection of triangles covering the area of the original triangle. First, the original triangle is subdivided into a collection of concentric equilateral triangles. The edges of each of these triangles are subdivided, and the area between each triangle pair is filled by triangles produced by joining the vertices on the subdivided edges. The number of concentric triangles and the number of subdivisions along each triangle except the outermost is derived from the first inner tessellation level. The edges of the outermost triangle are subdivided independently, using the first, second, and third outer tessellation levels to control the number of subdivisions of the u==0 (left), v==0 (bottom), and w==0 (right) edges, respectively. The second inner tessellation level and the fourth outer tessellation level have no effect in this mode.

If the first inner tessellation level and all three outer tessellation levels are exactly one after clamping and rounding, only a single triangle with (u,v,w) coordinates of (0,0,1), (1,0,0), and (0,1,0) is generated. If the inner tessellation level is one and any of the outer tessellation levels is greater than one, the inner tessellation level is treated as though it were originally specified as $1+\varepsilon$ and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce "inner" vertices positioned on the edge of the triangle.

If any tessellation level is greater than one, tessellation begins by producing a set of concentric inner triangles and subdividing their edges. First, the three outer edges are temporarily subdivided using the clamped and rounded first inner tessellation level and the specified tessellation spacing, generating n segments. For the outermost inner triangle, the inner triangle is degenerate — a single point at the center of the triangle — if n is two. Otherwise, for each corner of the outer triangle, an inner triangle corner is produced at the intersection of two lines extended perpendicular to the corner's two adjacent edges running through the vertex of the subdivided outer edge nearest that corner. If n is three, the edges of the inner triangle are not subdivided and is the final triangle in the set of concentric triangles. Otherwise, each edge of the inner triangle is divided into $n-2$ segments, with the $n-1$ vertices of this subdivision produced by intersecting the inner edge with lines perpendicular to the edge running through the $n-1$ innermost vertices of the subdivision of the outer edge. Once the outermost inner triangle is subdivided, the previous subdivision process repeats itself, using the generated triangle as an outer triangle. This subdivision process is illustrated in link.

Once all the concentric triangles are produced and their edges are subdivided, the area between each pair of adjacent inner triangles is filled completely with a set of non-overlapping triangles. In this subdivision, two of the three vertices of each triangle are taken from adjacent vertices on a subdivided edge of one triangle; the third is one of the vertices on the corresponding edge of the other triangle. If the innermost triangle is degenerate (i.e., a point), the triangle containing it is subdivided into six triangles by connecting each of the six vertices on that triangle with the center point. If the innermost triangle is not degenerate, that triangle is added to the set of generated triangles as-is.

After the area corresponding to any inner triangles is filled, the tessellator generates triangles to cover the area between the outermost triangle and the outermost inner triangle. To do this, the temporary subdivision of the outer triangle edge above is discarded. Instead, the u==0, v==0, and w==0 edges are subdivided according to the first, second, and third outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the first inner triangle is retained. The area between the outer and first inner triangles is completely filled by non-overlapping triangles as described above. If the first (and only) inner triangle is degenerate, a set of triangles is produced by connecting each vertex on the outer triangle edges with the center point.

After all triangles are generated, each vertex in the subdivided triangle is assigned a barycentric (u,v,w) coordinate based on its location relative to the three vertices of the outer triangle.

The algorithm used to subdivide the triangular domain in (u,v,w) space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles. The order in which the generated triangles passed to subsequent pipeline stages and the order of the vertices in those triangles are both implementation-dependent. However, when depicted in a manner similar to link, the order of the vertices in the generated triangles will be either all clockwise or all counter-clockwise, according to the vertex order layout declaration.

20.5 Quad Tessellation

If the tessellation primitive mode is **`Quads`**, a rectangle is subdivided into a collection of triangles covering the area of the original rectangle. First, the original rectangle is subdivided into a regular mesh of rectangles, where the number of rectangles along the u==0 and u==1 (vertical) and v==0 and v==1 (horizontal) edges are derived from the first and second inner tessellation levels, respectively. All rectangles, except those adjacent to one of the outer rectangle edges, are decomposed into triangle pairs. The outermost rectangle edges are subdivided independently, using the first, second, third, and fourth outer tessellation levels to control the number of subdivisions of the u==0 (left), v==0 (bottom), u==1 (right), and v==1 (top) edges, respectively. The area between the inner rectangles of the mesh and the outer rectangle edges are filled by triangles produced by joining the vertices on the subdivided outer edges to the vertices on the edge of the inner rectangle mesh.

If both clamped inner tessellation levels and all four clamped outer tessellation levels are exactly one, only a single triangle pair covering the outer rectangle is generated. Otherwise, if either clamped inner tessellation level is one, that tessellation level is treated as though it were originally specified as $1+\epsilon$ and will result in a two- or three-segment subdivision depending on the tessellation spacing. When used with fractional odd spacing, the three-segment subdivision may produce "inner" vertices positioned on the edge of the rectangle.

If any tessellation level is greater than one, tessellation begins by subdividing the u==0 and u==1 edges of the outer rectangle into m segments using the clamped and rounded first inner tessellation level and the tessellation spacing. The v==0 and v==1 edges are subdivided into n segments using the second inner tessellation level. Each vertex on the u==0 and v==0 edges are joined with the corresponding vertex on the u==1 and v==1 edges to produce a set of vertical and horizontal lines that divide the rectangle into a grid of smaller rectangles. The primitive generator emits a pair of non-overlapping triangles covering each such rectangle not adjacent to an edge of the outer rectangle. The boundary of the region covered by these triangles forms an inner rectangle, the edges of which are subdivided by the grid vertices that lie on the edge. If either m or n is two, the inner rectangle is degenerate, and one or both of the rectangle's "edges" consist of a single point. This subdivision is illustrated in Figure link.

After the area corresponding to the inner rectangle is filled, the tessellator must produce triangles to cover the area between the inner and outer rectangles. To do this, the subdivision of the outer rectangle edge above is discarded. Instead, the u==0, v==0, u==1, and v==1 edges are subdivided according to the first, second, third, and fourth outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the inner rectangle is retained. The area between the outer and inner rectangles is completely filled by non-overlapping triangles. Two of the three vertices of each triangle are adjacent vertices on a subdivided edge of one rectangle; the third is one of the vertices on the corresponding edge of the other triangle. If either edge of the innermost rectangle is degenerate, the area near the corresponding outer edges is filled by connecting each vertex on the outer edge with the single vertex making up the inner "edge".

The algorithm used to subdivide the rectangular domain in (u,v) space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles. The order in which the generated triangles passed to subsequent pipeline stages and the order of the vertices in those triangles are both implementation-dependent. However, when depicted in a manner similar to link, the order of the vertices in the generated triangles will be either all clockwise or all counter-clockwise, according to the vertex order layout declaration.

20.6 Isoline Tessellation

If the tessellation primitive mode is **`IsoLines`**, a set of independent horizontal line segments is drawn. The segments are arranged into connected strips called "isolines", where the vertices of each isoline have a constant v coordinate and u coordinates covering the full range [0,1]. The number of isolines generated is derived from the first outer tessellation level; the number of segments in each isoline is derived from the second outer tessellation level. Both inner tessellation levels and the third and fourth outer tessellation levels have no effect in this mode.

As with quad tessellation above, isoline tessellation begins with a rectangle. The u==0 and u==1 edges of the rectangle are subdivided according to the first outer tessellation level. For the purposes of this subdivision, the tessellation spacing mode is ignored and treated as equal_spacing. An isoline is drawn connecting each vertex on the u==0 rectangle edge to the corresponding vertex on the u==1 rectangle edge, except that no line is drawn between (0,1) and (1,1). If the number of isolines on the subdivided u==0 and u==1 edges is n, this process will result in n equally spaced lines with constant v coordinates of 0, $\frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}$.

Each of the n isolines is then subdivided according to the second outer tessellation level and the tessellation spacing, resulting in m line segments. Each segment of each line is emitted by the tessellator.

The order in which the generated line segments are passed to subsequent pipeline stages and the order of the vertices in each generated line segment are both implementation-dependent.

20.7 Tessellation Pipeline State

The *`pTessellationState`* member of `VkGraphicsPipelineCreateInfo` is of type VkPipelineTessellationStateCreateInfo:

```
typedef struct VkPipelineTessellationStateCreateInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    VkPipelineTesselationStateCreateFlags      flags;
    uint32_t                                   patchControlPoints;
} VkPipelineTessellationStateCreateInfo;
```

The members of the VkPipelineTessellationStateCreateInfo structure are as follows:

- *`sType`* is the type of this structure.
- *`pNext`* is `NULL` or a pointer to an extension-specific structure.
- *`flags`* is reserved for future use.
- *`patchControlPoints`* number of control points per patch.

Valid Usage

- `sType` must be VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO
- `pNext` must be `NULL`
- `flags` must be `0`
- `patchControlPoints` must be greater than zero and less than or equal to VkPhysicalDeviceLimits::`maxTessellationPatchSize`

Chapter 21

Geometry Shading

The geometry shader operates on a group of vertices and their associated data assembled from a single input primitive, and emits zero or more output primitives and the group of vertices and their associated data required for each output primitive. Geometry shading is enabled when a geometry shader is included in the pipeline.

21.1 Geometry Shader Input Primitives

Each geometry shader invocation has access to all vertices in the primitive (and their associated data), which are presented to the shader as an array of inputs. The input primitive type expected by the geometry shader is specified with an **`OpExecutionMode`** instruction in the geometry shader, and must be compatible with the primitive topology used by primitive assembly (if tessellation is not in use) or must match the type of primitive generated by the tessellation primitive generator (if tessellation is in use). Compatibility is defined below, with each input primitive type. The input primitive types accepted by a geometry shader are:

Points
: Geometry shaders that operate on points use an **`OpExecutionMode`** instruction specifing the **`InputPoints`** input mode. Such a shader is valid only when the pipeline primitive topology is **`VK_PRIMITIVE_TOPOLOGY_POINT_LIST`** (if tessellation is not in use) or if tessellation is in use and the tessellation evaluation shader uses **`PointMode`**. There is only a single input vertex available for each geometry shader invocation. However, inputs to the geometry shader are still presented as an array, but this array has a length of one.

Lines
: Geometry shaders that operate on line segments are generated by including an **`OpExecutionMode`** instruction with the **`InputLines`** mode. Such a shader is valid only for the **`VK_PRIMITIVE_TOPOLOGY_LINE_LIST`**, and **`VK_PRIMITIVE_TOPOLOGY_LINE_STRIP`** primitive topologies (if tessellation is not in use) or if tessellation is in use and the tessellation mode is **`Isolines`**. There are two input vertices available for each geometry shader invocation. The first vertex refers to the vertex at the beginning of the line segment and the second vertex refers to the vertex at the end of the line segment.

Lines with Adjacency
: Geometry shaders that operate on line segments with adjacent vertices are generated by including an **`OpExecutionMode`** instruction with the **`InputLinesAdjacency`** mode. Such a shader is valid only for the **`VK_PRIMITIVE_TOPOLOGY_LINES_WITH_ADJACENCY`** and **`VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY`** primitive topologies and must not be used when tessellation is in use.

In this mode, there are four vertices available for each geometry shader invocation. The second vertex refers to attributes of the vertex at the beginning of the line segment and the third vertex refers to the vertex at the end of the line segment. The first and fourth vertices refer to the vertices adjacent to the beginning and end of the line segment, respectively.

Triangles

Geometry shaders that operate on triangles are created by including an **`OpExecutionMode`** instruction with the **`Triangles`** mode. Such a shader is valid when the pipeline topology is **`VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST`**, **`VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP`**, or **`VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN`** (if tessellation is not in use) or when tessellation is in use and the tessellation mode is **`Triangles`** or **`Quads`**.

In this mode, there are three vertices available for each program invocation. The first, second and third vertices refer to attributes of the first, second and third vertex of the triangle, respectively.

Triangles with Adjacency

Geometry shaders that operate on triangles with adjacent vertices are created by including an **`OpExecutionMode`** instruction with the **`InputTrianglesAdjacency`** mode. Such a shader is valid when the pipeline topology is **`VK_PRIMITIVE_TOPOLOGY_TRIANGLES_WITH_ADJACENCY`** or **`VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY`**, and must not be used when tessellation is in use.

In this mode, there are six vertices available for each program invocation. The first, third and fifth vertices refer to attributes of the first, second and third vertex of the triangle, respectively. The second, fourth and sixth vertices refer to attributes of the vertices adjacent to the edges from the first to the second vertex, from the second to the third vertex, and from the third to the first vertex, respectively.

21.2 Geometry Shader Output Primitives

A geometry shader generates primitives in one of three output modes: points, line strips, or triangle strips. The primitive mode is specified in the shader using an **`OpExecutionMode`** instruction with the **`OutputPoints`**, **`OutputLineStrip`** or **`OutputTriangleStrip`** modes, respectively. Each geometry shader must include exactly one output primitive mode.

The vertices output by the geometry shader are assembled into points, lines, or triangles based on the output primitive type and the resulting primitives are then further processed as described in Chapter 23. If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, nothing is drawn. The number of vertices output by the geometry shader is limited to a maximum count specified in the shader.

The maximum output vertex count is specified in the shader using an **`OpExecutionMode`** instruction with the mode set to **`OutputVertices`** and the maximum number of vertices that will be produced by the geometry shader specified as a literal. Each geometry shader must specify a maximum output vertex count.

21.3 Multiple Invocations of Geometry Shaders

Geometry shaders can be invoked more than one time for each input primitive. This is known as *geometry shader instancing* and is requested by including an **`OpExecutionMode`** instruction with **`mode`** specified as **`Invocations`** and the number of invocations specified as an integer literal.

In this mode, the geometry shader will execute *n* times for each input primitive, where *n* is the number of invocations specified in the **`OpExecutionMode`** instruction. The instance number is available to each invocation as a built-in input using **`InvocationID`**.

21.4 Geometry Shader Primitive Ordering

Limited guarantees are provided for the relative ordering of primitives produced by a geometry shader.

- For instanced geometry shaders, the output primitives generated from each input primitive are passed to subsequent pipeline stages using the invocation number to order the primitives, from least to greatest.
- All output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.

Chapter 22

Fixed-Function Vertex Post-Processing

After programmable vertex processing, the following fixed-function operations are applied to vertices of the resulting primitives:

- Flatshading (see Flatshading).
- Primitive clipping, including client-defined half-spaces (see Primitive Clipping).
- Shader output attribute clipping (see Clipping Shader Outputs).
- Perspective division on clip coordinates (see Coordinate Transformations).
- Viewport mapping, including depth range scaling (see Controlling the Viewport).
- Front face determination for polygon primitives (see Basic Triangle Rasterization).

Next, rasterization is performed on primitives as described in chapter Rasterization.

22.1 Flatshading

Flatshading a vertex output attribute means to assign all vertices of the primitive the same value for that output.

The output values assigned are those of the *provoking vertex* of the primitive. The provoking vertex depends on the primitive topology, and is generally the "first" vertex of the primitive. For primitives not processed by tessellation or geometry shaders, the provoking vertex is selected from the input vertices according to the following table.

Table 22.1: Provoking vertex selection

Primitive type of primitive i	Provoking vertex number
VK_PRIMITIVE_TOPOLOGY_POINT_LIST	i
VK_PRIMITIVE_TOPOLOGY_LINE_LIST	$2i$
VK_PRIMITIVE_TOPOLOGY_LINE_STRIP	i
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST	$3i$
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP	i
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN	$i+1$
VK_PRIMITIVE_TOPOLOGY_LINE_LIST_ WITH_ADJACENCY	$4i+1$

Table 22.1: (continued)

VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY	$i+1$
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY	$6i$
VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY	$2i$

Flatshading is applied to those vertex attributes that match fragment input attributes which are decorated as **`Flat`**.

If a geometry shader is active, the output primitive topology is either points, line strips, or triangle strips, and the selection of the provoking vertex behaves according to the corresponding row of the table. If a tessellation evaluation shader is active and a geometry shader is not active, the provoking vertex is undefined but must be one of the vertices of the primitive.

22.2 Primitive Clipping

Primitives are culled against the *cull volume* and then clipped to the *clip volume*. In clip coordinates, the *view volume* is defined by:

$$-w_c \le x_c \le w_c$$
$$-w_c \le y_c \le w_c$$
$$0 \le z_c \le w_c$$

This view volume can be further restricted by as many as VkPhysicalDeviceLimits::*`maxClipDistances`* client-defined half-spaces.

The cull volume is the intersection of up to VkPhysicalDeviceLimits::*`maxCullDistances`* client-defined half-spaces (if no client-defined cull half-spaces are enabled, culling against the cull volume is skipped).

A shader must write a single cull distance for each enabled cull half-space to elements of the **`CullDistance`** array. If the cull distance for any enabled cull half-space is negative for all of the vertices of the primitive under consideration, the primitive is discarded. Otherwise the primitive is clipped against the clip volume as defined below.

The clip volume is the intersection of up to the value of VkPhysicalDeviceLimits::*`maxClipDistances`* client-defined half-spaces with the view volume (if no client-defined clip half-spaces are enabled, the clip volume is the view volume).

A shader must write a single clip distance for each enabled clip half-space to elements of the **`ClipDistance`** array. Clip half-space i is then given by the set of points satisfying the inequality

$$c_i(P) \ge 0$$

where $c_i(P)$ is the value of clip distance i at point P. For point primitives, $c_i(P)$ is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections Basic Line Segment Rasterization and Basic Polygon Rasterization, using the perspective interpolation equations.

The number of client-defined clip and cull half-spaces that are enabled is determined by the explicit size of the built-in array **`ClipDistance`** or **`CullDistance`** declared as an output in the interface of the entry point of the final shader stage before clipping.

Depth clamping is enabled or disabled via the *`depthClampEnable`* enable of the VkPipelineRasterizationStateCreateInfo structure. If depth clamping is enabled, the plane equation

$0 \leq z_c \leq w_c$

(see the clip volume definition above) is ignored by view volume clipping (effectively, there is no near or far plane clipping).

If the primitive under consideration is a point, then clipping passes it unchanged if it lies within the clip volume; otherwise, it is discarded.

If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume, and discards it if it lies entirely outside the volume.

If part of the line segment lies in the volume and part lies outside, then the line segment is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, $0 \leq t \leq 1$, for each clipped vertex. If the coordinates of a clipped vertex are $\mathbf{P}$ and the original vertices' coordinates are $\mathbf{P}_1$ and $\mathbf{P}_2$, then t is given by

$\mathbf{P} = t\mathbf{P}_1 + (1-t)\mathbf{P}_2.$

The value of t is used to clip vertex output attributes as described in Clipping Shader Outputs.

If the primitive is a polygon, it passes unchanged if every one of its edges lie entirely inside the clip volume, and it is discarded if every one of its edges lie entirely outside the clip volume. If the edges of the polygon intersect the boundary of the clip volume, the intersecting edges are reconnected by new edges that lie along the boundary of the clip volume - in some cases requiring the introduction of new vertices into a polygon.

If a polygon intersects an edge of the clip volume's boundary, the clipped polygon must include a point on this boundary edge.

Primitives rendered with user-defined half-spaces must satisfy a complementarity criterion. Suppose a series of primitives is drawn where each vertex i has a single specified clip distance d_i (or a number of similarly specified clip distances, if multiple half-spaces are enabled). Next, suppose that the same series of primitives are drawn again with each such clip distance replaced by $-d_i$ (and the graphics pipeline is otherwise the same). In this case, primitives must not be missing any pixels, and pixels must not be drawn twice in regions where those primitives are cut by the clip planes.

22.3 Clipping Shader Outputs

Next, vertex output attributes are clipped. The output values associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the output values assigned to vertices produced by clipping are clipped.

Let the output values assigned to the two vertices $\mathbf{P}_1$ and $\mathbf{P}_2$ of an unclipped edge be $\mathbf{c}_1$ and $\mathbf{c}_2$. The value of t (see Primitive Clipping) for a clipped point $\mathbf{P}$ is used to obtain the output value associated with $\mathbf{P}$ as

$\mathbf{c} = t\mathbf{c}_1 + (1-t)\mathbf{c}_2.$

(Multiplying an output value by a scalar means multiplying each of x, y, z, and w by the scalar.)

Since this computation is performed in clip space before division by w_c, clipped output values are perspective-correct.

Polygon clipping creates a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one half-space at a time. Output value clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

For vertex output attributes whose matching fragment input attributes are decorated with **`NoPerspective`**, the value of t used to obtain the output value associated with $\mathbf{P}$ will be adjusted to produce results that vary linearly in framebuffer space.

Output attributes of integer or unsigned integer type must always be flatshaded. Flatshaded attributes are constant over the primitive being rasterized (see Basic Line Segment Rasterization and Basic Polygon Rasterization), and no interpolation is performed. The output value **c** is taken from either $\mathbf{c}_1$ or $\mathbf{c}_2$, since flatshading has already occured and the two values are identical.

22.4 Coordinate Transformations

Clip coordinates for a vertex result from shader execution, which yields a vertex coordinate **`Position`**.

Perspective division on clip coordinates yields *normalized device coordinates*, followed by a *viewport* transformation (see Controlling the Viewport) to convert these coordinates into *framebuffer coordinates*.

If a vertex in clip coordinates has a position given by

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$$

then the vertex's normalized device coordinates are

$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \end{pmatrix}$$

22.5 Controlling the Viewport

The viewport transformation is determined by the selected viewport's width and height in pixels, p_x and p_y, respectively, and its center (o_x, o_y) (also in pixels), as well as its depth range min and max determining a depth range scale value p_z and a depth range bias value o_z (defined below). The vertex's framebuffer coordinates, $\begin{pmatrix} x_f \\ y_f \\ z_f \end{pmatrix}$, are given by

$$\begin{pmatrix} x_f \\ y_f \\ z_f \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2} x_d + o_x \\ \frac{p_y}{2} y_d + o_y \\ p_z \times z_d + o_z \end{pmatrix}.$$

Multiple viewports are available and are numbered zero up to the value of VkPhysicalDeviceLimits::*`maxViewports`*. The number of viewports used by a pipeline is controlled by the *`viewportCount`* member of the VkPipelineViewportStateCreateInfo structure used in pipeline creation:

```
typedef struct VkPipelineViewportStateCreateInfo {
    VkStructureType                                sType;
    const void*                                    pNext;
    VkPipelineViewportStateCreateFlags             flags;
    uint32_t                                       viewportCount;
    const VkViewport*                              pViewports;
    uint32_t                                       scissorCount;
    const VkRect2D*                                pScissors;
} VkPipelineViewportStateCreateInfo;
```

The members of the VkPipelineViewportStateCreateInfo structure are as follows:

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *viewportCount* is the number of viewports used by the pipeline.
- *pViewports* is a pointer to an array of `VkViewport` structs, defining the viewport transforms. If the viewport state is dynamic, this member is ignored.
- *scissorCount* is the number of scissors and must match the number of viewports.
- *pScissors* is a pointer to an array of VkRect2D structs which define the rectangular bounds of the scissor for the corresponding viewport. If the scissor state is dynamic, this member is ignored.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`
- The value of *viewportCount* must be greater than `0`
- The value of *scissorCount* must be greater than `0`
- If the multiple viewports feature is not enabled, *viewportCount* must be `1`
- If the multiple viewports feature is not enabled, *scissorCount* must be `1`
- *viewportCount* must be between `1` and VkPhysicalDeviceLimits::*maxViewports*, inclusive
- *scissorCount* must be between `1` and VkPhysicalDeviceLimits::*maxViewports*, inclusive
- The values of *scissorCount* and *viewportCount* must be identical

If a geometry shader is active and writes to **`ViewportIndex`**, the viewport transformation uses the viewport corresponding to the value assigned to **`ViewportIndex`** taken from an implementation-dependent vertex of each primitive. If the value of the viewport index is outside the range zero to the value of *viewportCount* minus one, the results of the viewport transformation are undefined. If no geometry shader is active, or if the active geometry shader does not statically write to **`ViewportIndex`**, the viewport numbered zero is used by the viewport transformation.

A single vertex can be used in more than one individual primitive, in primitives such as VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP. In this case, the viewport transformation is applied separately for each primitive.

Viewport transformation parameters are specified using the *pViewports* member of VkPipelineViewportStateCreateInfo in the pipeline state object. If the pipeline state object was created with the VK_DYNAMIC_STATE_VIEWPORT dynamic state enabled, modify the viewport transformation parameters dynamically with the command:

```
void vkCmdSetViewport(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    firstViewport,
    uint32_t                                    viewportCount,
    const VkViewport*                           pViewports);
```

- *commandbuffer* is the command buffer that records the command.
- *firstViewport* is the index of the first viewport whose parameters are updated by the command.
- *viewportCount* is the number of viewports whose parameters are updated by the command.
- *pViewports* is a pointer to an array of viewport parameters.

The viewport parameters taken from element i of *pViewports* replace the current state for the viewport index $\mathit{firstViewport} + i$, for i in $[0, \mathit{viewportCount})$.

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *pViewports* must be a pointer to an array of *viewportCount* valid VkViewport structures
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- The value of *viewportCount* must be greater than 0
- *firstViewport* must be less than VkPhysicalDeviceLimits::*maxViewports*
- The sum of *firstViewport* and *viewportCount* must be between 1 and VkPhysicalDeviceLimits::*maxViewports*, inclusive

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

Either of these methods of setting the viewport transformation parameters use the VkViewport struct:

```
typedef struct VkViewport {
    float                                       x;
    float                                       y;
    float                                                       width;
    float                                                       height;
    float                                                       minDepth;
    float                                                       maxDepth;
} VkViewport;
```

- `x` and `y` are the viewport's upper left corner (x, y).
- `width` and `height` are the viewport's width and height, respectively.
- `minDepth` and `maxDepth` are the depth range for the viewport. It is valid for `minDepth` to be greater than or equal to `maxDepth`.

Valid Usage

- `width` must be less than or equal to VkPhysicalDeviceLimits::`maxViewportDimensions`[0]
- `height` must be less than or equal to VkPhysicalDeviceLimits::`maxViewportDimensions`[1]
- `x` and `y` must each be between viewportBoundsRange[0] and viewportBoundsRange[1], inclusive
- `minDepth` must be between `0.0` and `1.0`, inclusive
- `maxDepth` must be between `0.0` and `1.0`, inclusive

The framebuffer depth coordinate z_f may be represented using either a fixed-point or floating-point representation. However, a floating-point representation must be used if the depth attachment is a floating-point format. If an m-bit fixed-point representation is used, we assume that it represents each value $\frac{k}{2^m-1}$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones).

The viewport parameters shown in the above equations are found from these values as

$$
\begin{aligned}
o_x &= x + \frac{width}{2} \\
o_y &= y + \frac{height}{2} \\
o_z &= minDepth \\
p_x &= width \\
p_y &= height \\
p_z &= maxDepth - minDepth.
\end{aligned}
$$

The viewport's upper left corner (x, y) is clamped to the implementation-dependent viewport bounds range when specified.

The viewport width and height are clamped to the implementation-dependent maximum viewport dimensions when specified. The maximum viewport dimensions must be greater than or equal to the larger of the visible dimensions of the display being rendered to (if a display exists), and the largest image which can be created and attached to a framebuffer.

The floating-point viewport bounds are represented with an implementation-dependent precision.

Chapter 23

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains associated data such as depth, color, or other attributes.

Rasterizing a primitive begins by determining which squares of an integer grid in framebuffer coordinates are occupied by the primitive, and assigning one or more depth values to each such square. This process is described below for points, lines, and polygons.

A grid square, including its (x,y) framebuffer coordinates, z (depth), and associated data added by fragment shaders, is called a fragment. A fragment is located by its upper left corner, which lies on integer grid coordinates.

Rasterization operations also refer to a fragment's sample locations, which are offset by subpixel fractional values from its upper left corner. The rasterization rules for points, lines, and triangles involve testing whether each sample location is inside the primitive. Fragments need not actually be square, and rasterization rules are not affected by the aspect ratio of fragments. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other.

We assume that fragments are square, since it simplifies antialiasing and texturing. After rasterization, fragments are processed by the early per-fragment tests, if enabled.

Several factors affect rasterization, including the members of VkPipelineRasterizationStateCreateInfo and VkPipelineMultisampleStateCreateInfo.

The VkPipelineRasterizationStateCreateInfo structure is defined as:

```
typedef struct VkPipelineRasterizationStateCreateInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    VkPipelineRasterizationStateCreateFlags    flags;
    VkBool32                                   depthClampEnable;
    VkBool32                                   rasterizerDiscardEnable;
    VkPolygonMode                              polygonMode;
    VkCullModeFlags                            cullMode;
    VkFrontFace                                frontFace;
    VkBool32                                   depthBiasEnable;
    float                                      depthBiasConstantFactor;
    float                                      depthBiasClamp;
    float                                      depthBiasSlopeFactor;
    float                                      lineWidth;
} VkPipelineRasterizationStateCreateInfo;
```

- *sType* is the type of this structure.
- *pNext* is `NULL` or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *depthClampEnable* controls whether to clamp the fragment's depth values instead of clipping primitives to the z planes of the frustum, as described in Primitive Clipping.
- *rasterizerDiscardEnable* controls whether primitives are discarded immediately before the rasterization stage.
- *polygonMode* triangle rendering mode. See `VkPolygonMode`.
- *cullMode* triangle facing direction used for primitive culling. See `VkCullModeFlagBits`.
- *frontFace* front-facing triangle orientation to be used for culling. See `VkFrontFace`.
- *depthBiasEnable* controls whether to bias fragment depth values.
- *depthBiasConstantFactor* scalar factor controlling a constant depth value added to each fragment.
- *depthBiasClamp* maximum (or minimum) depth bias of a fragment.
- *depthBiasSlopeFactor* scalar factor applied to a fragment's slope in depth bias calculations.
- *lineWidth* is the width of rasterized line segments.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`
- *polygonMode* must be a valid `VkPolygonMode` value
- *cullMode* must be a valid combination of `VkCullModeFlagBits` values
- *frontFace* must be a valid `VkFrontFace` value
- If the depth clamping feature is not enabled, the value of *depthClampEnable* must be VK_FALSE
- If the non-solid fill modes feature is not enabled, the value of *fillMode* must be VK_POLYGON_MODE_FILL

The VkPipelineMultisampleStateCreateInfo structure is defined as:

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

The members of the VkPipelineMultisampleStateCreateInfo structure are as follows:

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *rasterizationSamples* is an VkSampleCountFlagBits specifying the number of samples per pixel used in rasterization.
- *sampleShadingEnable* specifies that fragment shading executes per-sample if VK_TRUE, or per-fragment if VK_FALSE, as described in Sample Shading.
- *minSampleShading* is the minimum number of unique samples to shade for each fragment.
- *pSampleMask* is a bitmask of static coverage information that is ANDed with the coverage information generated during rasterization, as described in Sample Mask.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO
- *pNext* must be NULL
- *flags* must be 0
- *rasterizationSamples* must be a valid VkSampleCountFlagBits value
- If *pSampleMask* is not NULL, *pSampleMask* must be a pointer to an array of $\lceil \frac{rasterizationSamples}{32} \rceil$ VkSampleMask values
- If the sample rate shading feature is not enabled, *sampleShadingEnable* must be VK_FALSE
- If the alpha to one feature is not enabled, *alphaToOneEnable* must be VK_FALSE

Rasterization only produces fragments corresponding to pixels in the framebuffer. Fragments which would be produced by application of any of the primitive rasterization rules described below but which lie outside the framebuffer are not produced, nor are they processed by any later stage of the pipeline, including any of the early per-fragment tests described in Early Per-Fragment Tests.

Surviving fragments are processed by fragment shaders. Fragment shaders determine associated data for fragments, and can also modify or replace their assigned depth values.

If the subpass for which this pipeline is being created will reference color and/or depth/stencil attachments, then the value of *rasterizationSamples* must be the same as the sample count for those subpass attachments. Otherwise, the value of *rasterizationSamples* must follow the rules for a zero-attachment subpass.

23.1 Discarding Primitives Before Rasterization

Primitives are discarded before rasterization if the *`rasterizerDiscardEnable`* member of `VkPipelineRasterizationStateCreateInfo` is enabled. When enabled, primitives are discarded after the last enabled vertex, tessellation, or geometry shader, and before rasterization.

23.2 Multisampling

Multisampling is a mechanism to antialias all Vulkan primitives: points, lines, and polygons. The technique is to sample all primitives multiple times at each pixel. Each sample in each framebuffer attachment has storage for a color, depth, and/or stencil value, such that per-fragment operations apply to each sample independently. The color sample values are later *resolved* to a single color. Images with only a single sample must not be the target of a resolve.

Vulkan defines rasterization rules for single-sample modes in a way that is equivalent to a multisample mode with a single sample in the center of each pixel.

Each fragment includes a coverage value with *`rasterizationSamples`* bits (see Sample Mask). Each fragment includes *`rasterizationSamples`* depth values and sets of associated data. An implementation may choose to assign the same associated data to more than one sample. The location for evaluating such associated data may be anywhere within the pixel including the pixel center or any of the sample locations. When *`rasterizationSamples`* is VK_SAMPLE_COUNT_1_BIT, the pixel center must be used. The different associated data values need not all be evaluated at the same location. Each pixel fragment thus consists of integer x and y grid coordinates, *`rasterizationSamples`* depth values and sets of associated data, and a coverage value with *`rasterizationSamples`* bits.

It is understood that each pixel has *`rasterizationSamples`* locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a sample point. The sample points associated with a pixel must be located inside or on the boundary of the unit square that is considered to bound the pixel. Furthermore, the relative locations of sample points may be identical for each pixel in the framebuffer, or they may differ. If the current pipeline includes a fragment shader with one or more variables in its interface decorated with **`Sample`** and **`Input`**, the data associated with those variables will be assigned independently for each sample. The values for each sample must be evaluated at the location of the sample. The data associated with any other variables not decorated with **`Sample`** and **`Input`** need not be evaluated independently for each sample.

If the *`standardSampleLocations`* member of `VkPhysicalDeviceFeatures` is VK_TRUE, then the sample counts VK_SAMPLE_COUNT_1_BIT, VK_SAMPLE_COUNT_2_BIT, VK_SAMPLE_COUNT_4_BIT, VK_SAMPLE_COUNT_8_BIT, and VK_SAMPLE_COUNT_16_BIT have sample locations as listed in the following table, with the *i*th entry in the table corresponding to bit *i* in the sample masks. VK_SAMPLE_COUNT_32_BIT and VK_SAMPLE_COUNT_64_BIT do not have standard sample locations. Locations are defined relative to an origin in the upper left corner of the pixel.

Table 23.1: Standard sample locations

VK_SAMPLE_ COUNT_1_BIT	VK_SAMPLE_ COUNT_2_BIT	VK_SAMPLE_ COUNT_4_BIT	VK_SAMPLE_ COUNT_8_BIT	VK_SAMPLE_ COUNT_16_BIT

Table 23.1: (continued)

$(0.5, 0.5)$	$(0.25, 0.25)$	$(0.375, 0.125)$	$(0.5625, 0.3125)$	$(0.5625, 0.5625)$
	$(0.75, 0.75)$	$(0.875, 0.375)$	$(0.4375, 0.6875)$	$(0.4375, 0.3125)$
		$(0.125, 0.625)$	$(0.8125, 0.5625)$	$(0.3125, 0.625)$
		$(0.625, 0.875)$	$(0.3125, 0.1875)$	$(0.75, 0.4375)$
			$(0.1875, 0.8125)$	$(0.1875, 0.375)$
			$(0.0625, 0.4375)$	$(0.625, 0.8125)$
			$(0.6875, 0.9375)$	$(0.8125, 0.6875)$
			$(0.9375, 0.0625)$	$(0.6875, 0.1875)$
				$(0.375, 0.875)$
				$(0.5, 0.0625)$
				$(0.25, 0.125)$
				$(0.125, 0.75)$
				$(0.0, 0.5)$
				$(0.9375, 0.25)$
				$(0.875, 0.9375)$
				$(0.0625, 0.0)$

23.3 Sample Shading

Sample shading can be used to specify a minimum number of unique samples to process for each fragment. Sample shading is controlled by the *`sampleShadingEnable`* member of `VkPipelineMultisampleStateCreateInfo`. If *`sampleShadingEnable`* is disabled, sample shading has no effect. Otherwise, an implementation must provide a minimum of $\max(\lceil minSampleShading \times rasterizationSamples \rceil, 1)$ unique associated data for each fragment, where *`minSampleShading`* is the minimum fraction of sample shading requested in `VkPipelineMultisampleStateCreateInfo` and *`rasterizationSamples`* is the number of samples specified in `VkPipelineMultisampleStateCreateInfo`. These are associated with the samples in an implementation-dependent manner. *`minSampleShading`* must be in the range $[0, 1]$. When the sample shading fraction is 1.0, a separate set of associated data are evaluated for each sample, and each set of values is evaluated at the sample location.

23.4 Points

A point is drawn by generating a set of fragments in the shape of a square centered around the vertex of the point. Each vertex has an associated point size that controls the width/height of that square. The point size is taken from the (potentially clipped) shader built-in **`PointSize`** written by:

- the geometry shader, if active;
- the tessellation evaluation shader, if active and no geometry shader is active;
- the tessellation control shader, if active and no geometry or tessellation evaluation shader is active; or
- the vertex shader, otherwise

and clamped to the implementation-dependent point size range $[\mathit{pointSizeRange}[0], \mathit{pointSizeRange}[1]]$. If the value written to **`PointSize`** is less than or equal to zero, or if no value was written to **`PointSize`**, results are undefined.

Not all point sizes need be supported, but the size 1.0 must be supported. The range of supported sizes and the size of evenly-spaced gradations within that range are implementation-dependent. The range and gradations are obtained from the *`pointSizeRange`* and *`pointSizeGranularity`* members of `VkPhysicalDeviceLimits`. If, for instance, the size range is from 0.1 to 2.0 and the gradation size is 0.1, then the size 0.1, 0.2, . . . , 1.9, 2.0 are supported. Additional point sizes may also be supported. There is no requirement that these sizes be equally spaced. If an unsupported size is requested, the nearest supported size is used instead.

23.4.1 Basic Point Rasterization

Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect a region centered at the point's (x_f, y_f). This region is a square with side equal to the current point size. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0.

All fragments produced in rasterizing a point are assigned the same associated data, which are those of the vertex corresponding to the point. However, the fragment shader built-in **`PointCoord`** contains point sprite texture coordinates. The *s* and *t* point sprite texture coordinates vary from zero to one across the point horizontally left-to-right and top-to-bottom, respectively. The following formulas are used to evaluate *s* and *t*:

$$s = \frac{1}{2} + \frac{(x_p - x_f)}{size}$$

$$t = \frac{1}{2} + \frac{(y_p - y_f)}{size}.$$

where size is the point's size, (x_p, y_p) is the location at which the point sprite coordinates are evaluated - this may be the framebuffer coordinates of the pixel center (i.e. at the half-integer) or the location of an implementation-defined sample, and (x_f, y_f) is the exact, unrounded framebuffer coordinate of the vertex for the point. When *`rasterizationSamples`* is VK_SAMPLE_COUNT_1_BIT, the pixel center must be used.

23.5 Line Segments

23.5.1 Basic Line Segment Rasterization

Rasterized line segments produce fragments which intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment in directions perpendicular to the direction of the line. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage bits that correspond to sample points that intersect the rectangle are 1, other coverage bits are 0.

Next we specify how the data associated with each rasterized fragment are obtained. Let $\mathbf{p}_r = (x_d, y_d)$ be the framebuffer coordinates at which associated data are evaluated. This may be the pixel center of a fragment or an implementation-defined sample location within the fragment. When *`rasterizationSamples`* is VK_SAMPLE_COUNT_1_BIT, the pixel center must be used. Let $\mathbf{p}_a = (x_a, y_a)$ and $\mathbf{p}_b = (x_b, y_b)$ be initial and final endpoints of the line segment, respectively. Set

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}$$

(Note that $t = 0$ at $\mathbf{p}_a$ and $t = 1$ at $\mathbf{p}_b$. Also note that this calculation projects the vector from $\mathbf{p}_a$ to $\mathbf{p}_r$ onto the line, and thus computes the normalized distance of the fragment along the line.)

The value of an associated datum f for the fragment, whether it be a shader output or the clip w coordinate, is found as

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)/w_a + t/w_b}$$

EQUATION 23.1: line_perspective_interpolation

where f_a and f_b are the data associated with the starting and ending endpoints of the segment, respectively; w_a and w_b are the clip w coordinates of the starting and ending endpoints of the segments, respectively. However, depth values for lines must be interpolated by

$$z = (1-t)z_a + tz_b$$

EQUATION 23.2: line_noperspective_interpolation

where z_a and z_b are the depth values of the starting and ending endpoints of the segment, respectively.

The **`NoPerspective`** and **`Flat`** interpolation decorations can be used with fragment shader inputs to declare how they are interpolated. When neither decoration is applied, interpolation is performed as described in Equation line_perspective_interpolation. When the **`NoPerspective`** decoration is used, interpolation is performed in the same fashion as for depth values, as described in Equation line_noperspective_interpolation. When the **`Flat`** decoration is used, no interpolation is performed, and outputs are taken from the corresponding input value of the provoking vertex corresponding to that primitive.

Not all line widths need be supported for line segment rasterization, but width 1.0 antialiased segments must be provided. The range and gradations are obtained from the *`lineWidthRange`* and *`lineWidthGranularity`* members of `VkPhysicalDeviceLimits`. If, for instance, the size range is from 0.1 to 2.0 and the gradation size is 0.1, then the size 0.1, 0.2, . . . , 1.9, 2.0 are supported. Additional line widths may also be supported. There is no requirement that these widths be equally spaced. If an unsupported width is requested, the nearest supported width is used instead.

The line width is set by the *`lineWidth`* property of `VkPipelineRasterizationStateCreateInfo` in the currently active pipeline. If the active pipeline has VK_DYNAMIC_STATE_LINE_WIDTH enabled then the line width is set by calling **`vkCmdSetLineWidth`**:

```
void vkCmdSetLineWidth(
    VkCommandBuffer                                 commandBuffer,
    float                                           lineWidth);
```

- *`commandBuffer`* is the VkCommandBuffer into which the command will be recorded.
- *`lineWidth`* is the width of rasterized line segments.

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle

- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations
- If the wide lines feature is not enabled, the value of *lineWidth* must be `1.0`

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

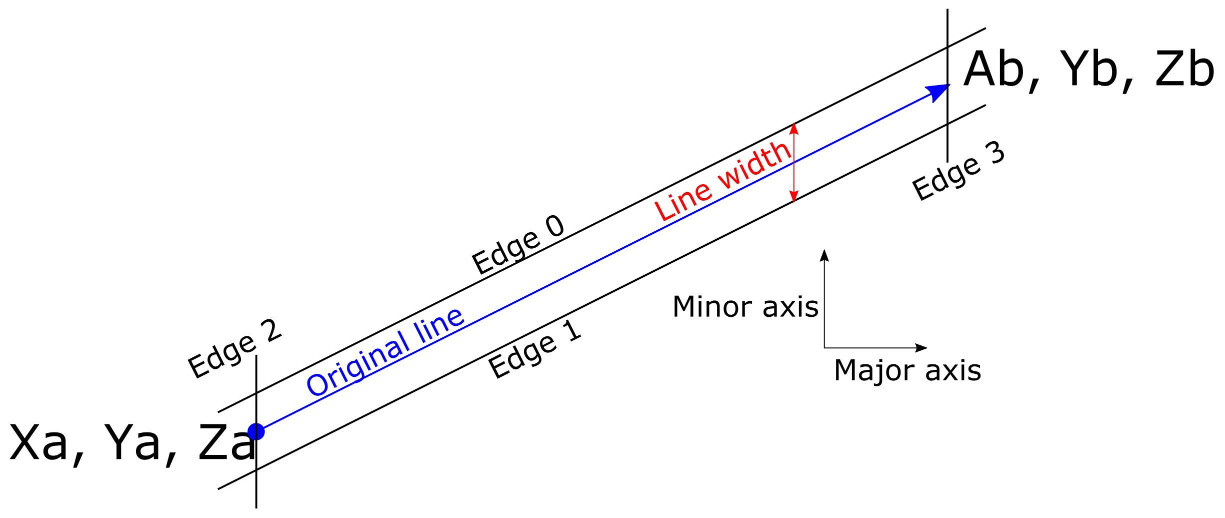
The above description documents the preferred method of line rasterization, and must be used when the implementation advertises the *strictLines* limit in `VkPhysicalDeviceLimits` as VK_TRUE.

When *strictLines* is VK_FALSE, the edges of the lines are generated as a parallelogram surrounding the original line. The major axis is chosen by noting the axis in which there is the greatest distance between the line start and end points. If the difference is equal in both directions then the X axis is chosen as the major axis. Edges 2 and 3 are aligned to the minor axis and are centered on the endpoints of the line as in Non strict lines, and each is *lineWidth* long. Edges 0 and 1 are parallel to the line and connect the endpoints of edges 2 and 3. Coverage bits that correspond to sample points that intersect the parallelogram are 1, other coverage bits are 0.

Samples that fall exactly on the edge of the parallelogram follow the polygon rasterization rules.

Interpolation occurs as if the parallelogram was decomposed into two triangles where each pair of vertices at each end of the line has identical attributes.

Non strict lines



23.6 Polygons

A polygon results from the decomposition of a triangle strip, triangle fan or a series of independent triangles. Like points and line segments, polygon rasterization is controled by several variables in the `VkPipelineRasterizationStateCreateInfo` structure.

23.6.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine whether the triangle is *back-facing* or *front-facing*. This determination is made based on the sign of the (clipped or unclipped) polygon's area computed in framebuffer coordinates. One way to compute this area is:

$$a = -\frac{1}{2}\sum_{i=0}^{n-1} x_f^i y_f^{i\oplus 1} - x_f^{i\oplus 1} y_f^i$$

where x_f^i and y_f^i are the x and y framebuffer coordinates of the ith vertex of the n-vertex polygon (vertices are numbered starting at zero for the purposes of this computation) and $i \oplus 1$ is $(i+1)$ mod n. The interpretation of the sign of this value is determined by the *`frontFace`* property of the `VkPipelineRasterizationStateCreateInfo` in the currently active pipeline, which takes the following values:

```
typedef enum VkFrontFace {
    VK_FRONT_FACE_COUNTER_CLOCKWISE = 0,
    VK_FRONT_FACE_CLOCKWISE = 1,
} VkFrontFace;
```

When this is set to VK_FRONT_FACE_COUNTER_CLOCKWISE, a triangle with positive area is considered front-facing. When it is set to VK_FRONT_FACE_CLOCKWISE, a triangle with negative area is considered front-facing. Any triangle which is not front-facing is back-facing, including zero-area triangles.

Once the orientation of triangles is determined, they are culled according to the setting of *`cullMode`* property in the `VkPipelineRasterizationStateCreateInfo` of the currently active pipeline, which takes the following values:

```
typedef enum VkCullModeFlagBits {
    VK_CULL_MODE_NONE = 0,
    VK_CULL_MODE_FRONT_BIT = 0x00000001,
    VK_CULL_MODE_BACK_BIT = 0x00000002,
    VK_CULL_MODE_FRONT_AND_BACK = 0x3,
} VkCullModeFlagBits;
```

If the *`cullMode`* is set to VK_CULL_MODE_NONE no triangles are discarded, if it is set to VK_CULL_MODE_FRONT_BIT front-facing triangles are discarded, if it is set to VK_CULL_MODE_BACK_BIT then back-facing triangles are discarded and if it is set to VK_CULL_MODE_FRONT_AND_BACK then all triangles are discarded. Following culling, fragments are produced for any triangles which have not been discarded.

The rule for determining which fragments are produced by polygon rasterization is called *point sampling*. The two-dimensional projection obtained by taking the x and y framebuffer coordinates of the polygon's vertices is formed. Fragments are produced for any pixels for which any sample points lie inside of this polygon. Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Special treatment is given to a sample whose sample location lies on a polygon edge. In such a case, if two polygons lie on either side of a common edge (with identical endpoints) on which a sample point lies, then exactly one of the polygons must result in a covered sample for that fragment during rasterization. As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle. Define *barycentric coordinates* for a triangle. Barycentric coordinates are a set of three numbers, a, b, and c, each in the range $[0,1]$, with $a+b+c=1$. These coordinates uniquely specify any point p within the triangle or on the triangle's boundary as

$$p = ap_a + bp_b + cp_c$$

where p_a, p_b, and p_c are the vertices of the triangle. a, b, and c are determined by:

$$a = \frac{\mathrm{A}(pp_bp_c)}{\mathrm{A}(p_ap_bp_c)}, \quad b = \frac{\mathrm{A}(pp_ap_c)}{\mathrm{A}(p_ap_bp_c)}, \quad c = \frac{\mathrm{A}(pp_ap_b)}{\mathrm{A}(p_ap_bp_c)},$$

where $A(lmn)$ denotes the area in framebuffer coordinates of the triangle with vertices l, n, and n.

Denote an associated datum at p_a, p_b, or p_c as f_a, f_b, or f_c, respectively. Then the value f of a datum at a fragment produced by rasterizing a triangle is given by:

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c}$$

EQUATION 23.3: triangle_perspective_interpolation

where w_a, w_b, and w_c are the clip w coordinates of p_a, p_b, and p_c, respectively. a, b, and c are the barycentric coordinates of the location at which the data are produced - this may be a pixel center or an implementation-defined sample location. When *`rasterizationSamples`* is VK_SAMPLE_COUNT_1_BIT, the pixel center must be used. Depth values for triangles must be interpolated by

$$z = az_a + bz_b + cz_c$$

EQUATION 23.4: triangle_noperspective_interpolation

where z_a, z_b, and z_c are the depth values of p_a, p_b, and p_c, respectively.

The **`NoPerspective`** and **`Flat`** interpolation decorations can be used with fragment shader inputs to declare how they are interpolated. When neither decoration is applied, interpolation is performed as described in Equation triangle_perspective_interpolation. When the **`NoPerspective`** decoration is used, interpolation is performed in the same fashion as for depth values, as described in Equation triangle_noperspective_interpolation. When the **`Flat`** decoration is used, no interpolation is performed, and outputs are taken from the corresponding input value of the provoking vertex corresponding to that primitive.

For a polygon with more than three edges, such as are produced by clipping a triangle, a convex combination of the values of the datum at the polygon's vertices must be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

$$f = \sum_{i=1}^{n} a_i f_i$$

where n is the number of vertices in the polygon and f_i is the value of the f at vertex i. For each i, $0 \leq a_i \leq 1$ and $\sum_{i=1}^{n} a_i = 1$. The values of a_i may differ from fragment to fragment, but at vertex i, $a_i = 1$ and $a_j = 0$ for $j \neq i$.

Note
One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case, the numerator and denominator of equation Equation triangle_perspective_interpolation are iterated independently and a division performed for each fragment).

23.6.2 Polygon Mode

The interpretation of polygons for rasterization is controlled using the *polygonMode* member of `VkPipelineRasterizationStateCreateInfo`, which takes the following values:

```
typedef enum VkPolygonMode {
    VK_POLYGON_MODE_FILL = 0,
    VK_POLYGON_MODE_LINE = 1,
    VK_POLYGON_MODE_POINT = 2,
} VkPolygonMode;
```

The *polygonMode* selects which method of rasterization is used for polygons. If *polygonMode* is VK_POLYGON_MODE_POINT, then the vertices of polygons are treated, for rasterization purposes, as if they had been drawn as points. VK_POLYGON_MODE_LINE causes polygon edges to be drawn as line segments. VK_POLYGON_MODE_FILL causes polygons to render using the polygon rasterization rules in this section.

Note that these modes affect only the final rasterization of polygons: in particular, a polygon's vertices are shaded and the polygon is clipped and possibly culled before these modes are applied.

23.6.3 Depth Bias

The depth values of all fragments generated by the rasterization of a polygon can be offset by a single value that is computed for that polygon. This behavior is controlled by the *depthBiasEnable*, *depthBiasConstantFactor*, *depthBiasClamp*, and *depthBiasSlopeFactor* members of `VkPipelineRasterizationStateCreateInfo`, or by the corresponding parameters to the **`vkCmdSetDepthBias`** command if depth bias state is dynamic.

```
void vkCmdSetDepthBias(
    VkCommandBuffer                             commandBuffer,
    float                                       depthBiasConstantFactor,
    float                                       depthBiasClamp,
    float                                       depthBiasSlopeFactor);
```

- *commandBuffer* is the VkCommandBuffer into which the command will be recorded.
- *depthBiasConstantFactor* scalar factor controlling a constant depth value added to each fragment.
- *depthBiasClamp* maximum (or minimum) depth bias of a fragment.
- *depthBiasSlopeFactor* scalar factor applied to a fragment's slope in depth bias calculations.

Valid Usage

- `commandBuffer` must be a valid VkCommandBuffer handle
- `commandBuffer` must be in the recording state
- The VkCommandPool that `commandBuffer` was allocated from must support graphics operations
- If the depth bias clamping feature is not enabled, the value of `depthBiasClamp` must be `0`.0

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

If `depthBiasEnable` is VK_FALSE, no depth bias is applied and the fragment's depth values are unchanged.

`depthBiasSlopeFactor` scales the maximum depth slope of the polygon, and `depthBiasConstantFactor` scales an implementation-dependent constant that relates to the usable resolution of the depth buffer. The resulting values are summed to produce the depth bias value which is then clamped to a minimum or maximum value specified by `depthBiasClamp`. `depthBiasSlopeFactor`, `depthBiasConstantFactor`, and `depthBiasClamp` can each be positive, negative, or zero.

The maximum depth slope m of a triangle is

$$m = \sqrt{\left(\frac{\partial z_f}{\partial x_f}\right)^2 + \left(\frac{\partial z_f}{\partial y_f}\right)^2} \tag{23.1}$$

where (x_f, y_f, z_f) is a point on the triangle. m may be approximated as

$$m = \max\left\{\left|\frac{\partial z_f}{\partial x_f}\right|, \left|\frac{\partial z_f}{\partial y_f}\right|\right\}. \tag{23.2}$$

The minimum resolvable difference r is an implementation-dependent parameter that depends on the depth buffer representation. It is the smallest difference in framebuffer coordinate z values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but z_f values that differ by r, will have distinct depth values.

For fixed-point depth buffer representations, r is constant throughout the range of the entire depth buffer. For floating-point depth buffers, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum exponent, e, in the range of z values spanned by the primitive. If n is the number of bits in the floating-point mantissa, the minimum resolvable difference, r, for the given primitive is defined as

$$r = 2^{e-n} \tag{23.3}$$

If no depth buffer is present, r is undefined.

The bias value o for a polygon is

$$o = \begin{cases} m \times depthBiasSlopeFactor + r \times depthBiasConstantFactor & depthBiasClamp = 0 \ or \ NaN \\ \min(m \times depthBiasSlopeFactor + r \times depthBiasConstantFactor, depthBiasClamp) & depthBiasClamp > 0 \\ \max(m \times depthBiasSlopeFactor + r \times depthBiasConstantFactor, depthBiasClamp) & depthBiasClamp < 0 \end{cases} \tag{23.4}$$

m is computed as described above. If the depth buffer uses a fixed-point representation, m is a function of depth values in the range $[0,1]$, and o is applied to depth values in the same range.

For fixed-point depth buffers, fragment depth values are always limited to the range $[0,1]$ by clamping after depth bias addition is performed. Fragment depth values are clamped even when the depth buffer uses a floating-point representation.

Chapter 24

Fragment Operations

24.1 Early Per-Fragment Tests

Once fragments are produced by rasterization, a number of per-fragment operations are performed prior to fragment shader execution. If a fragment is discarded during any of these operations, it will not be processed by any subsequent stage, including fragment shader execution.

Two fragment operations are performed in the following order:

- the scissor test (see Scissor Test)
- multisample fragment operations (see Sample Mask)

If early per-fragment operations are enabled by the fragment shader, these tests are also performed in the following order:

- the depth bounds tests (see Depth Bounds Tests)
- the stencil test (see Stencil Test)
- the depth test (see Depth Test)
- sample counting (see Sample Counting)

24.2 Scissor Test

The scissor test determines if a fragment's framebuffer coordinates (x_f, y_f) lie within the scissor rectangle corresponding to the viewport index (see Controlling the Viewport) used by the primitive that generated the fragment. The scissor rectangles are set by the `VkPipelineViewportStateCreateInfo` state of the pipeline state object. The scissor rectangles can be modified dynamically if the pipeline state object is created with VK_DYNAMIC_STATE_SCISSOR enabled. In that case the scissor rectangle values are set with the command:

```
void vkCmdSetScissor(
    VkCommandBuffer                             commandBuffer,
    uint32_t                                    firstScissor,
    uint32_t                                    scissorCount,
    const VkRect2D*                             pScissors);
```

- *`commandbuffer`* is the command buffer that records the command.
- *`firstScissor`* is the index of the first scissor whose state is updated by the command.
- *`scissorCount`* is the number of scissors whose rectangles are updated by the command.
- *`pScissors`* is a pointer to an array of scissor rectangles.

The scissor rectangles taken from element i of *`pScissors`* replace the current state for the scissor index $\mathit{firstScissor} + i$, for i in $[0, \mathit{scissorCount})$.

Each scissor rectangle is described by a struct of type `VkRect2D`, with the *`x`* and *`y`* values of *`offset`* determining the upper left corner of the scissor rectangle, and the *`width`* and *`height`* values of *`extent`* determining the size in pixels.

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`pScissors`* must be a pointer to an array of *`scissorCount`* VkRect2D structures
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support graphics operations
- The value of *`scissorCount`* must be greater than `0`
- *`firstScissor`* must be less than VkPhysicalDeviceLimits::*`maxViewports`*
- The sum of *`firstScissor`* and *`scissorCount`* must be between `1` and VkPhysicalDeviceLimits::*`maxViewports`*, inclusive
- The *`x`* and *`y`* members of *`offset`* and the *`width`* and *`height`* members of *`extent`* must be greater than or equal to `0`
- Evaluation of (*`offset.x`* + *`extent.width`*) must not cause a signed integer addition overflow
- Evaluation of (*`offset.y`* + *`extent.height`*) must not cause a signed integer addition overflow

Host Synchronization

- Host access to *`commandBuffer`* must be externally synchronized

If $\mathit{offset.x} \le x_f < \mathit{offset.x} + \mathit{extent.width}$ and $\mathit{offset.y} \le y_f < \mathit{offset.y} + \mathit{extent.height}$ for the selected scissor rectangle, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. For points, lines, and polygons, the scissor rectangle for a primitive is selected in the same manner as the viewport (see Controlling the Viewport). The scissor rectangles only apply to drawing commands, not to other commands like clears or copies.

It is legal for $offset.x + extent.width$ or $offset.y + extent.height$ to exceed the dimensions of the framebuffer - the scissor test still applies as defined above. Rasterization does not produce fragments outside of the framebuffer, so such fragments never have the scissor test performed on them.

The scissor test is always performed. Applications can effectively disable the scissor test by specifying a scissor rectangle that encompasses the entire framebuffer.

24.3 Sample Mask

This step modifies fragment coverage values based on the values in the *`pSampleMask`* array member of `VkPipelineMultisampleStateCreateInfo`, as described previously in section Section 9.2.

`pSampleMask` contains a bitmask of static coverage information that is **`ANDed`** with the coverage information generated during rasterization. Bits that are zero disable coverage for the corresponding sample. Bit B of mask word M corresponds to sample $32 \times M + B$. The array is sized to a length of $\lceil rasterizationSamples/32 \rceil$ words. If *`pSampleMask`* is `NULL`, it is treated as if the mask has all bits enabled, i.e. no coverage is removed from fragments.

24.4 Early Fragment Test Mode

The depth bounds test, stencil test, depth test, and occlusion query sample counting are performed before fragment shading if and only if early fragment tests are enabled by the fragment shader (see Early Fragment Tests). When early per-fragment operations are enabled, these operations are performed prior to fragment shader execution, and the stencil buffer, depth buffer, and occlusion query sample counts will be updated accordingly; these operations will not be performed again after fragment shader execution.

If a pipeline's fragment shader has early fragment tests disabled, these operations are performed only after fragment program execution, in the order described below. If a pipeline does not contain a fragment shader, these operations are performed only once.

If early fragment tests are enabled, any depth value computed by the fragment shader has no effect. Additionally, the depth test (including depth writes), stencil test (including stencil writes) and sample counting operations are performed even for fragments or samples that would be discarded after fragment shader execution due to per-fragment operations such as alpha-to-coverage tests, or due to the fragment being discarded by the shader itself.

24.5 Late Per-Fragment Tests

After programmable fragment processing, per-fragment operations are performed before blending and color output to the framebuffer.

A fragment is produced by rasterization with framebuffer coordinates of (x_f, y_f) and depth z, as described in Rasterization. The fragment is then modified by programmable fragment processing, which adds associated data as described in Shaders. The fragment is then further modified, and possibly discarded by the late per-fragment operations described in this chapter. These operations are diagrammed in figure Fragment Operations, in the order in which they are performed. Finally, if the fragment was not discarded, it is used to update the framebuffer at the fragment's framebuffer coordinates for any samples that remain covered.

The depth bounds test, stencil test, and depth test are performed for each pixel sample, rather than just once for each fragment. Stencil and depth operations are performed for a pixel sample only if that sample's fragment coverage bit is a value of 1 when the fragment executes the corresponding stage of the graphics pipeline. If the corresponding coverage bit is 0, no operations are performed for that sample. Failure of the depth bounds, stencil, or depth test results in termination of the processing of that sample by means of disabling coverage for that sample, rather than discarding of the fragment. If, at any point, a fragment's coverage becomes zero for all samples, then the fragment is discarded. All operations are performed on the depth and stencil values stored in the depth/stencil attachment of the framebuffer. The contents of the color attachments are not modified at this point.

The depth bounds test, stencil test, depth test, and occlusion query operations described in Depth Bounds Test, Stencil Test, Depth Test, Sample Counting are instead performed prior to fragment processing, as described in Early Fragment Test Mode, if requested by the fragment shader.

24.6 Multisample Coverage

If a fragment shader is active and statically assigns to the built-in output variable **`SampleMask`**, the fragment coverage is **`ANDed`** with the bits of the sample mask to generate a new fragment coverage value. If such a fragment shader did not assign a value to **`SampleMask`** due to flow of control, the value **`ANDed`** with the fragment coverage is undefined. If no fragment shader is active, or if the active fragment shader does not statically assign values to **`SampleMask`**, the fragment coverage is not modified.

Next, the fragment alpha and coverage values are modified based on the values of the *`alphaToCoverageEnable`* and *`alphaToOneEnable`* members of the `VkPipelineMultisampleStateCreateInfo` structure.

All alpha values in this section refer only to the alpha component of the fragment shader output linked to color number zero, index zero (see TODO:link). If that shader output is an integer or unsigned integer type, then these operations are skipped.

If *`alphaToCoverageEnable`* is enabled, a temporary coverage value is generated where each bit is determined by the fragment's alpha value. The temporary coverage value is then ANDed with the fragment coverage value to generate a new fragment coverage value.

No specific algorithm is specified for converting the alpha value to a temporary coverage mask. It is intended that the number of 1's in this value be proportional to the alpha value (clamped to $[0,1]$), with all 1's corresponding to a value of 1.0 and all 0's corresponding to 0.0. It is also intended that the algorithm be pseudo-random in nature, to avoid image artifacts due to regular coverage sample locations. The algorithm may be different at different pixel locations.

Note
Using different algorithms at different pixel location may help to avoid artifacts caused by regular coverage sample locations.

Next, if *`alphaToOneEnable`* is enabled, each alpha value is replaced by the maximum representable alpha value for fixed-point color buffers, or by 1.0 for floating-point buffers. Otherwise, the alpha values are not changed.

24.7 Depth Bounds Test

The depth bounds test conditionally disables coverage of a sample based on the outcome of a comparison between the value z_a in the depth attachment at location (x_f, y_f) (for the appropriate sample) and a range of values. The test is enabled or disabled by the *`depthBoundsTestEnable`* member of `VkPipelineDepthStencilStateCreateInfo`. The range of values used in the depth bounds test are also set by members of the `VkPipelineDepthStencilStateCreateInfo` but can be modified dynamically if the pipeline state object is created with the VK_DYNAMIC_STATE_DEPTH_BOUNDS dynamic state enabled. In that case the depth bounds range values are set with the command:

```
void vkCmdSetDepthBounds(
    VkCommandBuffer                             commandBuffer,
    float                                       minDepthBounds,
    float                                       maxDepthBounds);
```

If $minDepthBounds \leq z_a \leq maxDepthBounds$, then the depth bounds test passes. Otherwise, the test fails and the sample's coverage bit is cleared in the fragment. If there is no depth framebuffer attachment or if the depth bounds test is disabled, it is as if the depth bounds tests always passes.

24.8 Stencil Test

The stencil test conditionally disables coverage of a sample based on the outcome of a comparison between the value in the stencil attachment at location (x_f, y_f) (for the appropriate sample) and a reference value. The stencil test also updates the value in the stencil attachment, depending on the test state, the stencil attachment value and the stencil write masks. The test is enabled or disabled by the *stencilTestEnable* member of VkPipelineDepthStencilStateCreateInfo:

```
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;
```

When disabled, the stencil test and associated modifications are not made, and the sample's coverage is not modified.

The stencil test is controlled with the *front* and *back* members of VkPipelineDepthStencilStateCreateInfo which are of type VkStencilOpState.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO
- *pNext* must be `NULL`
- *flags* must be `0`
- *depthCompareOp* must be a valid `VkCompareOp` value
- *front* must be a valid VkStencilOpState structure
- *back* must be a valid VkStencilOpState structure

- If the depth bounds testing feature is not enabled, the value of *depthBoundsTestEnable* must be VK_FALSE

The definition of VkStencilOpState is:

```
typedef struct VkStencilOpState {
    VkStencilOp                                 failOp;
    VkStencilOp                                 passOp;
    VkStencilOp                                 depthFailOp;
    VkCompareOp                                 compareOp;
    uint32_t                                    compareMask;
    uint32_t                                    writeMask;
    uint32_t                                    reference;
} VkStencilOpState;
```

Valid Usage

- *failOp* must be a valid `VkStencilOp` value
- *passOp* must be a valid `VkStencilOp` value
- *depthFailOp* must be a valid `VkStencilOp` value
- *compareOp* must be a valid `VkCompareOp` value

There are two sets of stencil-related state, the front stencil state set and the back stencil state set. Stencil tests and writes use the front set of stencil state when processing fragments rasterized from non-polygon primitives (points and lines) and front-facing polygon primitives while the back set of stencil state is used when processing fragments rasterized from back-facing polygon primitives. For the purposes of stencil testing, a primitive is still considered a polygon even if the polygon is to be rasterized as points or lines due to the current `VkPolygonMode`. Whether a polygon is front- or back-facing is determined in the same manner used for face culling (see Basic Triangle Rasterization).

The operation of the stencil test is also affected by the *compareMask*, *writeMask* and *reference* members of VkStencilOpState set in the pipeline state object. These values can also be changed dynamically if the pipeline state object is created with the VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK, VK_DYNAMIC_STATE_STENCIL_WRITE_MASK or VK_DYNAMIC_STATE_STENCIL_REFERENCE dynamic states enabled.

These values are modified with the commands:

```
void vkCmdSetStencilCompareMask(
    VkCommandBuffer                             commandBuffer,
    VkStencilFaceFlags                          faceMask,
    uint32_t                                    compareMask);
```

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`faceMask`* must be a valid combination of `VkStencilFaceFlagBits` values
- *`faceMask`* must not be `0`
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support graphics operations

Host Synchronization

- Host access to *`commandBuffer`* must be externally synchronized

```
void vkCmdSetStencilWriteMask(
    VkCommandBuffer                             commandBuffer,
    VkStencilFaceFlags                          faceMask,
    uint32_t                                    writeMask);
```

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`faceMask`* must be a valid combination of `VkStencilFaceFlagBits` values
- *`faceMask`* must not be `0`
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support graphics operations

Host Synchronization

- Host access to *`commandBuffer`* must be externally synchronized

```
void vkCmdSetStencilReference(
    VkCommandBuffer                             commandBuffer,
    VkStencilFaceFlags                          faceMask,
    uint32_t                                    reference);
```

Valid Usage

- *commandBuffer* must be a valid VkCommandBuffer handle
- *faceMask* must be a valid combination of `VkStencilFaceFlagBits` values
- *faceMask* must not be `0`
- *commandBuffer* must be in the recording state
- The VkCommandPool that *commandBuffer* was allocated from must support graphics operations

Host Synchronization

- Host access to *commandBuffer* must be externally synchronized

faceMask for all the above commands is a bitmask specifying the fragments for which the stencil test is performed. Bits which can be set in *faceMask* include:

```
typedef enum VkStencilFaceFlagBits {
    VK_STENCIL_FACE_FRONT_BIT = 0x00000001,
    VK_STENCIL_FACE_BACK_BIT = 0x00000002,
    VK_STENCIL_FRONT_AND_BACK = 0x3,
} VkStencilFaceFlagBits;
```

- VK_STENCIL_FACE_FRONT_BIT specifies that the stencil test is performed for fragments generated from non-polygon primitives and front-facing primitives.
- VK_STENCIL_FACE_BACK_BIT specifies that the stencil test is performed for fragments generated from back-facing primitives.
- VK_STENCIL_FRONT_AND_BACK specifies the combination of VK_STENCIL_FACE_FRONT_BIT and VK_STENCIL_FACE_BACK_BIT.

reference is an integer reference value that is used in the unsigned stencil comparison. Stencil comparison clamps the reference value to $[0, 2^s - 1]$, where s is the number of bits in the stencil framebuffer attachment. The s least significant bits of *compareMask* are bitwise **ANDed** with both the reference and the stored stencil value, and the resulting masked values are those that participate in the comparison controlled by *compareOp*. Let R be the masked reference value and S be the masked stored stencil value. *compareOp* is a symbolic constant that determines the stencil comparison function:

```
typedef enum VkCompareOp {
    VK_COMPARE_OP_NEVER = 0,
    VK_COMPARE_OP_LESS = 1,
    VK_COMPARE_OP_EQUAL = 2,
    VK_COMPARE_OP_LESS_OR_EQUAL = 3,
    VK_COMPARE_OP_GREATER = 4,
    VK_COMPARE_OP_NOT_EQUAL = 5,
    VK_COMPARE_OP_GREATER_OR_EQUAL = 6,
    VK_COMPARE_OP_ALWAYS = 7,
} VkCompareOp;
```

- VK_COMPARE_OP_NEVER: the test never passes.
- VK_COMPARE_OP_LESS: the test passes when $R < S$.
- VK_COMPARE_OP_EQUAL: the test passes when $R = S$.
- VK_COMPARE_OP_LESS_OR_EQUAL: the test passes when $R \leq S$.
- VK_COMPARE_OP_GREATER: the test passes when $R > S$.
- VK_COMPARE_OP_NOT_EQUAL: the test passes when $R \neq S$.
- VK_COMPARE_OP_GREATER_OR_EQUAL: the test passes when $R \geq S$.
- VK_COMPARE_OP_ALWAYS: the test always passes.

VkStencilOpState contains three enums that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. Each enum is of type:

```
typedef enum VkStencilOp {
    VK_STENCIL_OP_KEEP = 0,
    VK_STENCIL_OP_ZERO = 1,
    VK_STENCIL_OP_REPLACE = 2,
    VK_STENCIL_OP_INCREMENT_AND_CLAMP = 3,
    VK_STENCIL_OP_DECREMENT_AND_CLAMP = 4,
    VK_STENCIL_OP_INVERT = 5,
    VK_STENCIL_OP_INCREMENT_AND_WRAP = 6,
    VK_STENCIL_OP_DECREMENT_AND_WRAP = 7,
} VkStencilOp;
```

failOp is the action performed on samples that fail the stencil test. *depthFailOp* is the action performed on samples that pass the stencil test and fail the depth test. *passOp* is the action performed on samples that pass both the stencil and depth tests.

The possible values are:

- VK_STENCIL_OP_KEEP keeps the current value.
- VK_STENCIL_OP_ZERO sets the value to 0.
- VK_STENCIL_OP_REPLACE sets the value to *reference*.

- VK_STENCIL_OP_INCREMENT_AND_CLAMP increments the current value and clamps to the maximum representable unsigned value.
- VK_STENCIL_OP_DECREMENT_AND_CLAMP decrements the current value and clamps to 0.
- VK_STENCIL_OP_INVERT bitwise-inverts the current value.
- VK_STENCIL_OP_INCREMENT_AND_WRAP increments the current value and wraps to 0 when the maximum value would have been exceeded.
- VK_STENCIL_OP_DECREMENT_AND_WRAP decrements the current value and wraps to the maximum possible value when the value would go below 0.

For purposes of increment and decrement, the stencil bits are considered as an unsigned integer.

If the stencil test fails, the sample's coverage bit is cleared in the fragment. If there is no stencil framebuffer attachment, stencil modification cannot occur, and it is as if the stencil tests always pass.

If the stencil test passes, the *`writeMask`* member of the `VkStencilOpState` structures controls how the updated stencil value is written to the stencil framebuffer attachment.

The least significant s bits of *`writeMask`*, where s is the number of bits in the stencil framebuffer attachment, specify an integer mask. Where a 1 appears in this mask, the corresponding bit in the stencil attachment is written; where a 0 appears, the bit is not written. The *`writeMask`* value uses either the front-facing or back-facing state based on the facing-ness of the fragment. Fragments generated by front-facing primitives use the front mask and fragments generated by back-facing primitives use the back mask.

24.9 Depth Test

The depth test conditionally disables coverage of a sample based on the outcome of a comparison between the fragment's depth value at the sample location and the sample's depth value in the depth attachment at location (x_f, y_f). The comparison is enabled or disabled with the *`depthTestEnable`* member of the `VkPipelineDepthStencilStateCreateInfo` structure. When disabled, the depth comparison and subsequent possible updates to the depth attachment value are bypassed and the fragment is passed to the next operation. The stencil value, however, can be modified as indicated above if the depth test passed. If enabled, the comparison takes place and the depth attachment value can subsequently be modified.

The comparison is specified with the *`depthCompareOp`* member of `VkPipelineDepthStencilStateCreateInfo`. Let z_f be the incoming fragment's depth value for a sample, and let z_a be the depth attachment value in memory for that sample. The depth test passes under the following conditions:

- VK_COMPARE_OP_NEVER: the test never passes.
- VK_COMPARE_OP_LESS: the test passes when $z_f < z_a$.
- VK_COMPARE_OP_EQUAL: the test passes when $z_f = z_a$.
- VK_COMPARE_OP_LESS_OR_EQUAL: the test passes when $z_f \leq z_a$.
- VK_COMPARE_OP_GREATER: the test passes when $z_f > z_a$.
- VK_COMPARE_OP_NOT_EQUAL: the test passes when $z_f \neq z_a$.
- VK_COMPARE_OP_GREATER_OR_EQUAL: the test passes when $z_f \geq z_a$.

- VK_COMPARE_OP_ALWAYS: the test always passes.

If depth clamping (see Primitive Clipping) is enabled, before the incoming fragment's z_f is compared to z_a, z_f is clamped to $[\min(n, f), \max(n, f)]$, where n and f are the *minDepth* and *maxDepth* depth range values of the viewport used by this fragment, respectively.

If the depth test fails, the sample's coverage bit is cleared in the fragment. The stencil value at the sample's location is updated according to the function currently in effect for depth test failure. Otherwise, the fragment continues to the next operation and the value of the depth framebuffer attachment at the sample's location is conditionally written to the sample's z_f value. In this case the stencil value is updated according to the function currently in effect for depth test success.

Upon passing the depth test, a sample's (possibly clamped) z_f value is conditionally written to the depth framebuffer attachment based on the value of the *depthWriteEnable* member of `VkPipelineDepthStencilStateCreateInfo`. If *depthWriteEnable* is VK_TRUE the value is written, and if it is VK_FALSE the value is not written.

If there is no depth framebuffer attachment, it is as if the depth test always passes.

24.10 Sample Counting

Occlusion queries use query pool entries to track the number of samples that pass all the per-fragment tests. The mechanism of collecting an occlusion query value is described in Occlusion Queries.

The occlusion query sample counter increments by one for each sample with a coverage value of 1 in each fragment that survives all the per-fragment tests, including scissor, sample mask, alpha to coverage, stencil, and depth tests.

Chapter 25

The Framebuffer

25.1 Blending

Blending combines the incoming *source* fragment's R, G, B and A values with the *destination* R, G, B and A values of each sample stored in the framebuffer at the fragment's (x_f, y_f) location. Blending is performed for each pixel sample, rather than just once for each fragment.

Source and destination values are combined according to the *blend operation* (`VkBlendOp`), quadruplets of source and destination weighting factors determined by the *blend factors* (`VkBlendFactor`), and a *blend constant* (TODO:link), to obtain a new set of R, G, B and A values, as described below.

Blending is computed and applied separately to each color attachment used by the subpass, with separate controls for each attachment.

Prior to performing the blend operation, signed and unsigned normalized fixed-point color components undergo an implied conversion to floating-point as specified by Conversion from Normalized Fixed-Point to Floating-Point. Blending computations are treated as if carried out in floating-point, and will be performed with a precision and dynamic range no lower than that used to represent destination components.

Blending applies only to fixed-point and floating-point color attachments. If the color attachment has an integer format, blending is not applied.

The pipeline blend state is included in the VkPipelineColorBlendStateCreateInfo struct during graphics pipeline creation:

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

The members of the VkPipelineColorBlendStateCreateInfo structure are as follows:

- *`sType`* is the type of this structure.

- *pNext* is NULL or a pointer to an extension-specific structure.
- *flags* is reserved for future use.
- *logicOpEnable*: whether to apply Logical Operations.
- *logicOp*: which logical operation to apply.
- *attachmentCount*: the number of VkPipelineColorBlendAttachmentState elements in *pAttachments*. This value must equal the *colorAttachmentCount* for the subpass in which this pipeline is used.
- *pAttachments*: pointer to array of per target attachment states.
- *blendConstants* is an array of four R, G, B and A color values that are used in blending, depending on the blend factor.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO
- *pNext* must be NULL
- *flags* must be 0
- If *attachmentCount* is not 0, *pAttachments* must be a pointer to an array of *attachmentCount* valid VkPipelineColorBlendAttachmentState structures
- If the independent blending feature is not enabled, all elements of *pAttachments* must be identical
- If the logic operations feature is not enabled, *logicOpEnable* must be VK_FALSE
- If *logicOpEnable* is VK_TRUE, *logicOp* must be a valid VkLogicOp value

The elements of the *pAttachments* array specify per-target blending state, and are of type:

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

Blending of each individual color attachment is controlled by the corresponding element of the *pAttachments* array. If the independent blending feature is not enabled on the device, all VkPipelineColorBlendAttachmentState elements in the *pAttachments* array must be identical. The members of the VkPipelineColorBlendAttachmentState struct have the following meanings:

- `blendEnable` controls whether blending is enabled for the corresponding color attachment. If blending is not enabled, the source fragment's color for that attachment is passed through unmodified.
- `srcColorBlendFactor` selects which blend factor is used to determine the source factors S_r, S_g, S_b.
- `dstColorBlendFactor` selects which blend factor is used to determine the destination factors D_r, D_g, D_b.
- `colorBlendOp` selects which blend operation is used to calculate the RGB values to write to the color attachment.
- `srcAlphaBlendFactor` selects which blend factor is used to determine the source factor S_a.
- `dstAlphaBlendFactor` selects which blend factor is used to determine the destination factor D_a.
- `alphaBlendOp` selects which blend operation is use to calculate the alpha values to write to the color attachment.
- `colorWriteMask` is a bitmask selecting which of the R, G, B, and/or A components are enabled for writing, as described later in this chapter.

Valid Usage

- `srcColorBlendFactor` must be a valid `VkBlendFactor` value
- `dstColorBlendFactor` must be a valid `VkBlendFactor` value
- `colorBlendOp` must be a valid `VkBlendOp` value
- `srcAlphaBlendFactor` must be a valid `VkBlendFactor` value
- `dstAlphaBlendFactor` must be a valid `VkBlendFactor` value
- `alphaBlendOp` must be a valid `VkBlendOp` value
- `colorWriteMask` must be a valid combination of `VkColorComponentFlagBits` values
- If the dual source blending feature is not enabled, `srcColorBlendFactor` must not be `VK_BLEND_SRC1_COLOR`, `VK_BLEND_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_SRC1_ALPHA`, or `VK_BLEND_ONE_MINUS_SRC1_ALPHA`
- If the dual source blending feature is not enabled, `dstColorBlendFactor` must not be `VK_BLEND_SRC1_COLOR`, `VK_BLEND_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_SRC1_ALPHA`, or `VK_BLEND_ONE_MINUS_SRC1_ALPHA`
- If the dual source blending feature is not enabled, `srcAlphaBlendFactor` must not be `VK_BLEND_SRC1_COLOR`, `VK_BLEND_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_SRC1_ALPHA`, or `VK_BLEND_ONE_MINUS_SRC1_ALPHA`
- If the dual source blending feature is not enabled, `dstAlphaBlendFactor` must not be `VK_BLEND_SRC1_COLOR`, `VK_BLEND_ONE_MINUS_SRC1_COLOR`, `VK_BLEND_SRC1_ALPHA`, or `VK_BLEND_ONE_MINUS_SRC1_ALPHA`

25.1.1 Blend Factors

The source and destination color and alpha blending factors are selected from the following table.

```
typedef enum VkBlendFactor {
    VK_BLEND_FACTOR_ZERO = 0,
    VK_BLEND_FACTOR_ONE = 1,
    VK_BLEND_FACTOR_SRC_COLOR = 2,
    VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR = 3,
    VK_BLEND_FACTOR_DST_COLOR = 4,
    VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR = 5,
    VK_BLEND_FACTOR_SRC_ALPHA = 6,
    VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA = 7,
    VK_BLEND_FACTOR_DST_ALPHA = 8,
    VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA = 9,
    VK_BLEND_FACTOR_CONSTANT_COLOR = 10,
    VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR = 11,
    VK_BLEND_FACTOR_CONSTANT_ALPHA = 12,
    VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA = 13,
    VK_BLEND_FACTOR_SRC_ALPHA_SATURATE = 14,
    VK_BLEND_FACTOR_SRC1_COLOR = 15,
    VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR = 16,
    VK_BLEND_FACTOR_SRC1_ALPHA = 17,
    VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA = 18,
} VkBlendFactor;
```

Table 25.1: Blend Factors

VkBlendFactor	**RGB Blend Factors (S_r,S_g,S_b) or (D_r,D_g,D_b)**	**Alpha Blend Factor (S_a or D_a)**
VK_BLEND_FACTOR_ZERO	$(0,0,0)$	0
VK_BLEND_FACTOR_ONE	$(1,1,1)$	1
VK_BLEND_FACTOR_SRC_COLOR	(R_{s0},G_{s0},B_{s0})	A_{s0}
VK_BLEND_FACTOR_ONE_MINUS_SRC_COLOR	$(1-R_{s0},1-G_{s0},1-B_{s0})$	$1-A_{s0}$
VK_BLEND_FACTOR_DST_COLOR	(R_d,G_d,B_d)	A_d
VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR	$(1-R_d,1-G_d,1-B_d)$	$1-A_d$
VK_BLEND_FACTOR_SRC_ALPHA	(A_{s0},A_{s0},A_{s0})	A_{s0}
VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA	$(1-A_{s0},1-A_{s0},1-A_{s0})$	$1-A_{s0}$
VK_BLEND_FACTOR_DST_ALPHA	(A_d,A_d,A_d)	A_d

Table 25.1: (continued)

VkBlendFactor	**RGB Blend Factors** (S_r, S_g, S_b) **or** (D_r, D_g, D_b)	**Alpha Blend Factor** $(S_a$ **or** $D_a)$
VK_BLEND_FACTOR_ONE_MINUS_DST_ALPHA	$(1-A_d, 1-A_d, 1-A_d)$	$1-A_d$
VK_BLEND_FACTOR_CONSTANT_COLOR	(R_c, G_c, B_c)	A_c
VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_COLOR	$(1-R_c, 1-G_c, 1-B_c)$	$1-A_c$
VK_BLEND_FACTOR_CONSTANT_ALPHA	(A_c, A_c, A_c)	A_c
VK_BLEND_FACTOR_ONE_MINUS_CONSTANT_ALPHA	$(1-A_c, 1-A_c, 1-A_c)$	$1-A_c$
VK_BLEND_FACTOR_SRC_ALPHA_SATURATE	$(f, f, f); f = \min(A_{s0}, 1-A_d)$	1
VK_BLEND_FACTOR_SRC1_COLOR	(R_{s1}, G_{s1}, B_{s1})	A_{s1}
VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR	$(1-R_{s1}, 1-G_{s1}, 1-B_{s1})$	$1-A_{s1}$
VK_BLEND_FACTOR_SRC1_ALPHA	(A_{s1}, A_{s1}, A_{s1})	A_{s1}
VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA	$(1-A_{s1}, 1-A_{s1}, 1-A_{s1})$	$1-A_{s1}$

In this table, the following conventions are used

- R_{s0}, G_{s0}, B_{s0} and A_{s0} represent the first source color R, G, B and A components, respectively, for the fragment output corresponding to the color attachment being blended.
- R_{s1}, G_{s1}, B_{s1} and A_{s1} represent the second source color R, G, B and A components, respectively, used in dual source blending modes, for the fragment output corresponding to the color attachment being blended.
- R_c, G_c, B_c and A_c represent the blend constant R, G, B and A components, respectively.

The *blend constant* (R_c, G_c, B_c, A_c) is specified via the `blendConstants` member of `VkPipelineColorBlendStateCreateInfo`. The blend constant can be modified dynamically if the pipeline state object is created with the VK_DYNAMIC_STATE_BLEND_CONSTANTS dynamic state enabled. Set the blend constant dynamically by calling the command:

```
void vkCmdSetBlendConstants(
    VkCommandBuffer                             commandBuffer,
    const float                                 blendConstants[4]);
```

- `commandBuffer` is the VkCommandBuffer that this command will be recorded in.
- `blendConstants` is an array of four R, G, B and A color values that are used in blending, depending on the blend factor.

Valid Usage

- `commandBuffer` must be a valid VkCommandBuffer handle
- `commandBuffer` must be in the recording state
- The VkCommandPool that `commandBuffer` was allocated from must support graphics operations

Host Synchronization

- Host access to `commandBuffer` must be externally synchronized

25.1.2 Dual-Source Blending

Blend factors that use the secondary color input $(R_{s1}, G_{s1}, B_{s1}, A_{s1})$ (VK_BLEND_FACTOR_SRC1_COLOR, VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR, VK_BLEND_FACTOR_SRC1_ALPHA, and VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA) may consume hardware resources that could otherwise be used for rendering to multiple color attachments. Therefore, the number of color attachments that can be used in a framebuffer may be lower when using dual-source blending.

Dual-source blending is only supported if the dualSrcBlend feature is enabled.

The maximum number of color attachments that can be used in a subpass when using dual-source blending functions is implementation-dependent and is reported as the `maxFragmentDualSrcAttachments` member of VkPhysicalDeviceLimits.

When using a fragment shader with dual-source blending functions, the color outputs are bound to the first and second inputs of the blender using TODO, as described in TODO. If the second color input to the blender is not written in the shader, or if no output is bound to the second input of a blender, the result of the blending operation is not defined.

25.1.3 Blend Operations

Once the source and destination blend factors have been selected, they along with the source and destination components are passed to the blending operation. The blending operations are selected from the table below, with RGB and alpha components potentially using different blend operations.

```
typedef enum VkBlendOp {
    VK_BLEND_OP_ADD = 0,
    VK_BLEND_OP_SUBTRACT = 1,
    VK_BLEND_OP_REVERSE_SUBTRACT = 2,
    VK_BLEND_OP_MIN = 3,
    VK_BLEND_OP_MAX = 4,
} VkBlendOp;
```

Table 25.2: Blend Operations

VkBlendOp	**RGB Components**	**Alpha Component**
VK_BLEND_OP_ADD	$R = R_{s0} * S_r + R_d * D_r$ $G = G_{s0} * S_g + G_d * D_g$ $B = B_{s0} * S_b + B_d * D_b$	$A = A_{s0} * S_a + A_d * D_a$
VK_BLEND_OP_ SUBTRACT	$R = R_{s0} * S_r - R_d * D_r$ $G = G_{s0} * S_g - G_d * D_g$ $B = B_{s0} * S_b - B_d * D_b$	$A = A_{s0} * S_a - A_d * D_a$
VK_BLEND_OP_ REVERSE_SUBTRACT	$R = R_d * D_r - R_{s0} * S_r$ $G = G_d * D_g - G_{s0} * S_g$ $B = B_d * D_b - B_{s0} * S_b$	$A = A_d * D_a - A_{s0} * S_a$
VK_BLEND_OP_MIN	$R = \min(R_{s0}, R_d)$ $G = \min(G_{s0}, G_d)$ $B = \min(B_{s0}, B_d)$	$A = \min(A_{s0}, A_d)$
VK_BLEND_OP_MAX	$R = \max(R_{s0}, R_d)$ $G = \max(G_{s0}, G_d)$ $B = \max(B_{s0}, B_d)$	$A = \max(A_{s0}, A_d)$

In this table, the following conventions are used:

- R_{s0}, G_{s0}, B_{s0} and A_{s0} represent the first source color R, G, B, and A components, respectively.
- R_d, G_d, B_d and A_d represent the R, G, B, and A components of the destination color. That is, the color currently in the corresponding color attachment for this fragment/sample.
- S_r, S_g, S_b and S_a represent the source blend factor R, G, B, and A components, respectively.
- D_r, D_g, D_b and D_a represent the destination blend factor R, G, B, and A components, respectively.

The blending operation produces a new set of values R, G, B and A, which are written to the framebuffer attachment. If blending is not enabled for this attachment, then R, G, B and A are assigned the values of R_{s0}, G_{s0}, B_{s0} and A_{s0}.

If the color attachment is fixed-point, the components of the source and destination values and blend factors are each clamped to $[0,1]$ or $[-1,1]$ respectively for an unsigned normalized or signed normalized color attachment prior to evaluating the blend operations. If the color attachment is floating-point, no clamping occurs.

The *colorWriteMask* member of `VkPipelineColorBlendAttachmentState` determines whether the final color values R, G, B and A are written to the framebuffer attachment. *colorWriteMask* is any combination of the following bits:

```
typedef enum VkColorComponentFlagBits {
    VK_COLOR_COMPONENT_R_BIT = 0x00000001,
    VK_COLOR_COMPONENT_G_BIT = 0x00000002,
    VK_COLOR_COMPONENT_B_BIT = 0x00000004,
    VK_COLOR_COMPONENT_A_BIT = 0x00000008,
} VkColorComponentFlagBits;
```

If VK_COLOR_COMPONENT_R_BIT is set, then the R value is written to color attachment for the appropriate sample, otherwise the value in memory is unmodified. The VK_COLOR_COMPONENT_G_BIT, VK_COLOR_COMPONENT_B_BIT, and VK_COLOR_COMPONENT_A_BIT bits similarly control writing of the G, B, and A values. The *colorWriteMask* is applied regardless of whether blending is enabled.

If the numeric format of a framebuffer attachment uses sRGB encoding, the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence are linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB texture components in section TODO:link. If the format is not sRGB, no linearization is performed.

25.2 sRGB Conversion

If the numeric format of a framebuffer attachment uses sRGB encoding, then the final R, G and B values are converted into the non-linear sRGB color space before being written to the framebuffer attachment by computing:

$$c_s = \begin{cases} 0.0, & c_l \leq 0 \\ 12.92 c_l, & 0 < c_l < 0.0031308 \\ 1.055 c_l^{0.41666} - 0.055, & 0.0031308 \leq c_l < 1 \\ 1.0, & c_l \geq 1 \end{cases}$$

where c_l is the R, G or B element and c_s is the result (effectively converted into sRGB color space).

If the framebuffer color attachment numeric format is not sRGB encoded then the resulting c_s values for R, G and B are unmodified. The value of A is never sRGB encoded. That is, the alpha component is always stored in memory as linear.

25.3 Logical Operations

The application can enable a *logical operation* between the fragment's color values and the existing value in the framebuffer attachment. This logical operation is applied prior to updating the framebuffer attachment. Logical operations are applied only for signed and unsigned integer and normalized integer framebuffers. Logical operations are not applied to floating-point or sRGB format color attachments.

Logical operations are controlled by the *logicOpEnable* and *logicOp* members of `VkPipelineColorBlendStateCreateInfo`. If *logicOpEnable* is VK_TRUE, then a logical operation selected by *logicOp* is applied between each color attachment and the fragment's corresponding output value, and blending of all attachments is treated as if it were disabled. Any attachments using color formats for which logical operations are not supported simply pass through the color values unmodified. The logical operation is applied independently for each of the red, green, blue, and alpha components. The *logicOp* is selected from the following operations:

```
typedef enum VkLogicOp {
    VK_LOGIC_OP_CLEAR = 0,
    VK_LOGIC_OP_AND = 1,
    VK_LOGIC_OP_AND_REVERSE = 2,
    VK_LOGIC_OP_COPY = 3,
    VK_LOGIC_OP_AND_INVERTED = 4,
    VK_LOGIC_OP_NO_OP = 5,
    VK_LOGIC_OP_XOR = 6,
    VK_LOGIC_OP_OR = 7,
    VK_LOGIC_OP_NOR = 8,
    VK_LOGIC_OP_EQUIVALENT = 9,
    VK_LOGIC_OP_INVERT = 10,
    VK_LOGIC_OP_OR_REVERSE = 11,
    VK_LOGIC_OP_COPY_INVERTED = 12,
    VK_LOGIC_OP_OR_INVERTED = 13,
    VK_LOGIC_OP_NAND = 14,
    VK_LOGIC_OP_SET = 15,
} VkLogicOp;
```

The logical operations supported by Vulkan are summarized in the following table in which

- $\neg$ is bitwise invert,
- $\wedge$ is bitwise and,
- $\vee$ is bitwise or,
- $\oplus$ is bitwise exclusive or,
- s is the fragment's R_{s0}, G_{s0}, B_{s0} or A_{s0} component value for the fragment output corresponding to the color attachment being updated, and
- d is the color attachment's R, G, B or A component value:

Table 25.3: Logical Operations

Mode	**Operation**
VK_LOGIC_OP_CLEAR	0
VK_LOGIC_OP_AND	$s \wedge d$
VK_LOGIC_OP_AND_REVERSE	$s \wedge \neg d$

Table 25.3: (continued)

Mode	**Operation**
VK_LOGIC_OP_COPY	s
VK_LOGIC_OP_AND_INVERTED	$\neg s \wedge d$
VK_LOGIC_OP_NO_OP	d
VK_LOGIC_OP_XOR	$s \oplus d$
VK_LOGIC_OP_OR	$s \vee d$
VK_LOGIC_OP_NOR	$\neg(s \vee d)$
VK_LOGIC_OP_EQUIVALENT	$\neg(s \oplus d)$
VK_LOGIC_OP_INVERT	$\neg d$
VK_LOGIC_OP_OR_REVERSE	$s \vee \neg d$
VK_LOGIC_OP_COPY_INVERTED	$\neg s$
VK_LOGIC_OP_OR_INVERTED	$\neg s \vee d$
VK_LOGIC_OP_NAND	$\neg(s \wedge d)$
VK_LOGIC_OP_SET	all 1s

The result of the logical operation is then written to the color attachment as controlled by the channel write mask, described in Blend Operations.

Chapter 26

Dispatching Commands

Dispatching commands (commands with "Dispatch" in the name) provoke work in a compute pipeline. Dispatching commands are recorded into a command buffer and when executed by a queue, will produce work which executes according to the currently bound compute pipeline. A compute pipeline must be bound to a command buffer before any dispatch commands are recorded in that command buffer.

To record a dispatch, call:

```
void vkCmdDispatch(
    VkCommandBuffer                                 commandBuffer,
    uint32_t                                        x,
    uint32_t                                        y,
    uint32_t                                        z);
```

`commandbuffer` specifies the command buffer which records the command.

When the command is executed, workgroups are assembled using *`x`*, *`y`*, and *`z`*, which specify the number of workgroups that will be dispatched in the X, Y and Z dimensions, respectively.

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support compute operations
- This command must only be called outside of a render pass instance
- *`x`* must be less than or equal to VkPhysicalDeviceLimits::*`maxComputeWorkGroupCount`*[0]
- *`y`* must be less than or equal to VkPhysicalDeviceLimits::*`maxComputeWorkGroupCount`*[1]
- *`z`* must be less than or equal to VkPhysicalDeviceLimits::*`maxComputeWorkGroupCount`*[2]

- For each set *n* that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_COMPUTE, a descriptor set must have been bound to *n* at VK_PIPELINE_BIND_POINT_COMPUTE, with a VkPipelineLayout that is compatible for set *n*, with the VkPipelineLayout used to create the current VkPipeline, as described in Section 13.2.2.1
- Descriptors in each bound descriptor set, specified via **`vkCmdBindDescriptorSets`**, must be valid if they are used by the currently bound VkPipeline object, specified via **`vkCmdBindPipeline`**
- A valid compute pipeline must be bound to the current command buffer with VK_PIPELINE_BIND_POINT_COMPUTE
- For each push constant that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_COMPUTE, a push constant value must have been set for VK_PIPELINE_BIND_POINT_COMPUTE, with a VkPipelineLayout that is compatible for push constants with the one used to create the current VkPipeline, as described in Section 13.2.2.1
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_COMPUTE uses unnormalized coordinates, it must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_TYPE_CUBE_ARRAY, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_COMPUTE uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with "ImplicitLod", "Dref" or "Proj" in their name, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_COMPUTE uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a lod bias or any offset values, in any shader stage
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_COMPUTE accesses a uniform buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_COMPUTE accesses a storage buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set

Host Synchronization

- Host access to *`commandBuffer`* must be externally synchronized

An indirect dispatch is recorded by calling:

```
void vkCmdDispatchIndirect(
    VkCommandBuffer                             commandBuffer,
    VkBuffer                                    buffer,
    VkDeviceSize                                offset);
```

This command behaves similarly to **`vkCmdDispatch`** except that the parameters are read by the device from a buffer during execution. The buffer containing the parameters is specified by *`buffer`* and the byte offset into the buffer where the parameters begin is specified in *`offset`*. At this offset, the parameters of the dispatch are encoded in a VkDispatchIndirectCommand structure.

Valid Usage

- *`commandBuffer`* must be a valid VkCommandBuffer handle
- *`buffer`* must be a valid VkBuffer handle
- *`commandBuffer`* must be in the recording state
- The VkCommandPool that *`commandBuffer`* was allocated from must support compute operations
- This command must only be called outside of a render pass instance
- Each of *`commandBuffer`* and *`buffer`* must have been created, allocated or retrieved from the same VkDevice
- For each set *n* that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_ COMPUTE, a descriptor set must have been bound to *n* at VK_PIPELINE_BIND_POINT_COMPUTE, with a VkPipelineLayout that is compatible for set *n*, with the VkPipelineLayout used to create the current VkPipeline, as described in Section 13.2.2.1
- Descriptors in each bound descriptor set, specified via **`vkCmdBindDescriptorSets`**, must be valid if they are used by the currently bound VkPipeline object, specified via **`vkCmdBindPipeline`**
- A valid compute pipeline must be bound to the current command buffer with VK_PIPELINE_BIND_POINT_ COMPUTE
- *`buffer`* must have been created with the VK_BUFFER_USAGE_INDIRECT_BUFFER_BIT bit set
- The sum of *`offset`* and the size of VkDispatchIndirectCommand must be less than or equal to the size of *`buffer`*
- For each push constant that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_ POINT_COMPUTE, a push constant value must have been set for VK_PIPELINE_BIND_POINT_ COMPUTE, with a VkPipelineLayout that is compatible for push constants with the one used to create the current VkPipeline, as described in Section 13.2.2.1
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_ POINT_COMPUTE uses unnormalized coordinates, it must not be used to sample from any VkImage with a VkImageView of the type VK_IMAGE_VIEW_TYPE_3D, VK_IMAGE_VIEW_TYPE_CUBE, VK_ IMAGE_VIEW_TYPE_1D_ARRAY, VK_IMAGE_VIEW_TYPE_2D_ARRAY or VK_IMAGE_VIEW_ TYPE_CUBE_ARRAY, in any shader stage
- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_ POINT_COMPUTE uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions with "ImplicitLod", "Dref" or "Proj" in their name, in any shader stage

- If any VkSampler object that is referenced by the VkPipeline currently bound to VK_PIPELINE_BIND_POINT_COMPUTE uses unnormalized coordinates, it must not be used with any of the SPIR-V `OpImageSample*` or `OpImageSparseSample*` instructions that includes a lod bias or any offset values, in any shader stage
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_COMPUTE accesses a uniform buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set
- If the robust buffer access feature is not enabled, and any shader stage in the VkPipeline object currently bound to VK_PIPELINE_BIND_POINT_COMPUTE accesses a storage buffer, it must not access values outside of the range of that buffer specified in the currently bound descriptor set

Host Synchronization

- Host access to *`commandBuffer`* must be externally synchronized

The definition of VkDispatchIndirectCommand is:

```
typedef struct VkDispatchIndirectCommand {
    uint32_t                                   x;
    uint32_t                                   y;
    uint32_t                                   z;
} VkDispatchIndirectCommand;
```

The members of VkDispatchIndirectCommand structure have the same meaning as the similarly named parameters of `vkCmdDispatch`.

Valid Usage

- *`x`* must be less than or equal to VkPhysicalDeviceLimits::*`maxComputeWorkGroupCount`*[0]
- *`y`* must be less than or equal to VkPhysicalDeviceLimits::*`maxComputeWorkGroupCount`*[1]
- *`z`* must be less than or equal to VkPhysicalDeviceLimits::*`maxComputeWorkGroupCount`*[2]

Chapter 27

Sparse Resources

As documented in Resource Memory Association, VkBuffer and VkImage resources in Vulkan must be bound completely and contiguously to a single VkDeviceMemory object. This binding must be done before the resource is used, and the binding is immutable for the lifetime of the resource.

Sparse resources relax these restrictions and provide these additional features:

- Sparse resources can be mapped non-contiguously to one or more VkDeviceMemory allocations.
- Sparse resources can be re-bound to different memory allocations over the lifetime of the resource.
- Sparse resources can have descriptors generated and used orthogonally with memory binding commands.

27.1 Sparse Resource Features

Sparse resources have several features that must be enabled explicitly at resource creation time. The features are enabled by including bits in the *`flags`* parameter of `VkImageCreateInfo` or `VkBufferCreateInfo`. Each feature also has one or more corresponding feature enables specified in `VkPhysicalDeviceFeatures`.

- Sparse binding is the base feature, and provides the following capabilities:
  - Resources can be bound at some defined (block) granularity.
  - The entire resource must be bound before use regardless of regions actually accessed.
  - No specific mapping of image region to memory offset is defined, i.e. the location that each texel corresponds to in memory is implementation-dependent.
  - Sparse buffers have a well-defined mapping of buffer range to memory range, where an offset within the buffer corresponds to an identical offset in the memory.
  - Requested via the VK_IMAGE_CREATE_SPARSE_BINDING_BIT and VK_BUFFER_CREATE_SPARSE_BINDING_BIT bits.
  - A sparse image created using VK_IMAGE_CREATE_SPARSE_BINDING_BIT (but not VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT) supports all formats that non-sparse usage supports, and supports both VK_IMAGE_TILING_OPTIMAL and VK_IMAGE_TILING_LINEAR tiling.
- *Sparse Residency* builds on the *`sparseBinding`* feature. It includes the following capabilities:

- Resources do not have to be completely bound before use on the device.
- Images have a prescribed block layout, allowing specific image blocks to be bound to specific offsets in memory allocations.
- Consistency of access to unbound regions of the resource is defined by the absence or presence of VkPhysicalDeviceSparseProperties::`residencyNonResidentStrict`. If this property is present, accesses to unbound regions of the resource are well defined and behave as if the data bound is populated with all zeros; writes are discarded. When this property is absent, accesses are considered safe, but reads will return undefined values.
- Requested via the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT and VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT bits.
- Support is advertised on a finer grain via the following features:
  - sparseResidencyBuffer: Support for creating VkBuffer objects with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT.
  - sparseResidencyImage2D: Support for creating 2D single-sampled VkImage objects with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
  - sparseResidencyImage3D: Support for creating 3D VkImage objects with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
  - sparseResidency2Samples: Support for creating 2D VkImage objects with 2 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
  - sparseResidency4Samples: Support for creating 2D VkImage objects with 4 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
  - sparseResidency8Samples: Support for creating 2D VkImage objects with 8 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
  - sparseResidency16Samples: Support for creating 2D VkImage objects with 16 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

  Implementations supporting *`sparseResidencyImage2D`* are only required to support sparse 2D, single-sampled images. Support is not required for sparse 3D and MSAA images and is enabled via *`sparseResidencyImage3D`*, *`sparseResidency2Samples`*, *`sparseResidency4Samples`*, *`sparseResidency8Samples`*, and *`sparseResidency16Samples`*.
- A sparse image created using VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT supports all non-compressed color formats with power-of-two texel size that non-sparse usage supports. Additional formats may also be supported and can be queried via `vkGetPhysicalDeviceSparseImageFormatProperties`. VK_IMAGE_TILING_LINEAR tiling is not supported.

• Sparse aliasing provides the following capability that can be enabled per resource:

  Allows physical memory blocks to be shared between multiple locations in the same sparse resource or between multiple sparse resources, with each binding of a memory location observing a consistent interpretation of the memory contents.

  See Sparse Memory Aliasing for more information.

27.2 Sparse Buffers and Fully-Resident Images

Both VkBuffer and VkImage objects created with the VK_IMAGE_CREATE_SPARSE_BINDING_BIT or VK_BUFFER_CREATE_SPARSE_BINDING_BIT bits but without the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT or VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT bits can be thought of as a linear region of address space. In the VkImage case, this linear region is opaque, meaning that there is no application-visible mapping between pixel location and memory offset.

Unless VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT or VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT are also used, the entire resource must be bound to one or more VkDeviceMemory objects before use.

27.2.1 Sparse Buffer and Fully-Resident Image Block Size

The block size for sparse buffers and fully-resident images is reported as VkMemoryRequirements::*alignment*. This *alignment* value represents both the memory alignment requirement and the binding granularity (in bytes) for sparse resources.

27.3 Sparse Partially-Resident Buffers

VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT bit allow the buffer to be made only partially resident. Partially resident VkBuffer objects are allocated and bound identically to VkBuffer objects using only the VK_BUFFER_CREATE_SPARSE_BINDING_BIT feature. The only difference is the ability for some regions of the buffer to be unbound during device use.

27.4 Sparse Partially-Resident Images

VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT bit allow blocks of the image to be bound to specific ranges of memory. This allows the application to manage residency at either the subresource or pixel block granularity. Each subresource (outside of the mip tail) starts on a block boundary and has a size that is an integer number of blocks of memory.

Applications can use these types of images to control level-of-detail based on total memory consumption. If memory pressure becomes an issue the application can unbind / delete specific LODs from images without having to recreate resources or pixel data from unaffected LODs.

The application can also use this functionality to access subregions of the image in a "megatexture" fashion. The application can create a VERY large image and only populate the region of the image that is currently being used in the scene.

27.4.1 Accessing Unbound Regions

The following member of VkPhysicalDeviceSparseProperties affect how data in unbound regions of sparse resources are handled by the implementation:

- *residencyNonResidentStrict*

If this property is not present, reads of unbound regions of the image will return undefined values. Both reads and writes are still considered *safe* and will not affect other resources or populated regions of the image.

If this property is present, all reads of unbound regions of the image will behave as if the region was bound to memory populated with all zeros; writes will be discarded.

Formatted accesses to unbound memory may still alter some component values in the natural way for those accesses, e.g. substituting a value of one for alpha in formats that do not have an alpha component.

Example: Reading the alpha component of an unbacked VK_FORMAT_R8_UNORM image will return a value of $1.0f$.

See Physical Device Enumeration for instructions for retrieving physical device properties.

Implementor's Note

For hardware that cannot natively handle access to unbound regions of a resource, the implementation may allocate and map memory to the unbound regions. Reads and writes to unbound regions will access the implementation-managed memory instead of causing a hardware fault.

Given that reads of unbound regions are undefined in this scenario, implementations may use the same physical memory for unbound regions of multiple resources within the same process.

27.4.2 Mip Tail Regions

Sparse images created using VK_IMAGE_CREATE_SPARSE_BINDING_BIT have no specific mapping of image region or subresource to memory offset defined, so the entire image can be thought of as a linear opaque address region. However, images created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT do have a prescribed block layout, and hence each subresource must start on a block boundary. Within each array layer, the set of mip-levels that are too small to fill a block are grouped together into a *mip tail region*.

If the VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT flag is present in the `flags` member of VkSparseImageFormatProperties, for the image's `format`, then any mip-level which is not a multiple of the block size, and all subsequent mip-levels, are also included in the mip tail region.

The following member of VkPhysicalDeviceSparseProperties may affect how the implementation places mip levels in the mip tail region:

- `residencyAlignedMipSize`

Each mip tail region is mapped as an opaque block (i.e. must be mapped using a `VkSparseImageOpaqueMemoryBindInfo` structure) and may be of a size greater than or equal to a normal mappable block. It is guaranteed to be a multiple of the normal image block size (in bytes).

An implementation may choose to allow each array-layer's mip tail region to be mapped independently or require that all array-layer's mip tail regions be treated as one. This is dictated by VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT in VkSparseImageMemoryRequirements::`flags`.

The following diagrams depict how VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT and VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT alter memory usage and requirements.

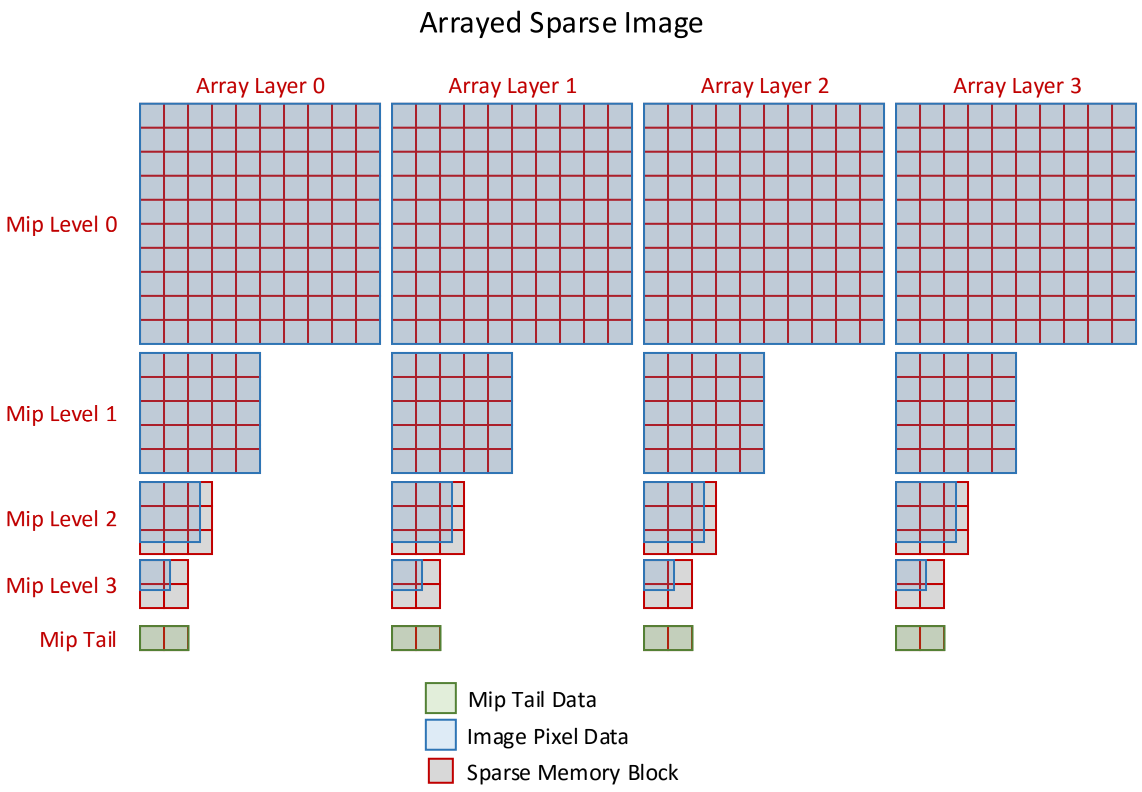


Figure 27.1: Sparse Image

In the absense of VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT and VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT, each array layer contains a mip tail region containing pixel data for all mip levels smaller than the sparse block size in any dimension.

Mip levels that are as large or larger than a block in all dimensions can be bound individually. Right-edges and bottom-edges of each level are allowed to have partially used blocks. Any bound partially-used-blocks must still have their full block size allocated in memory.

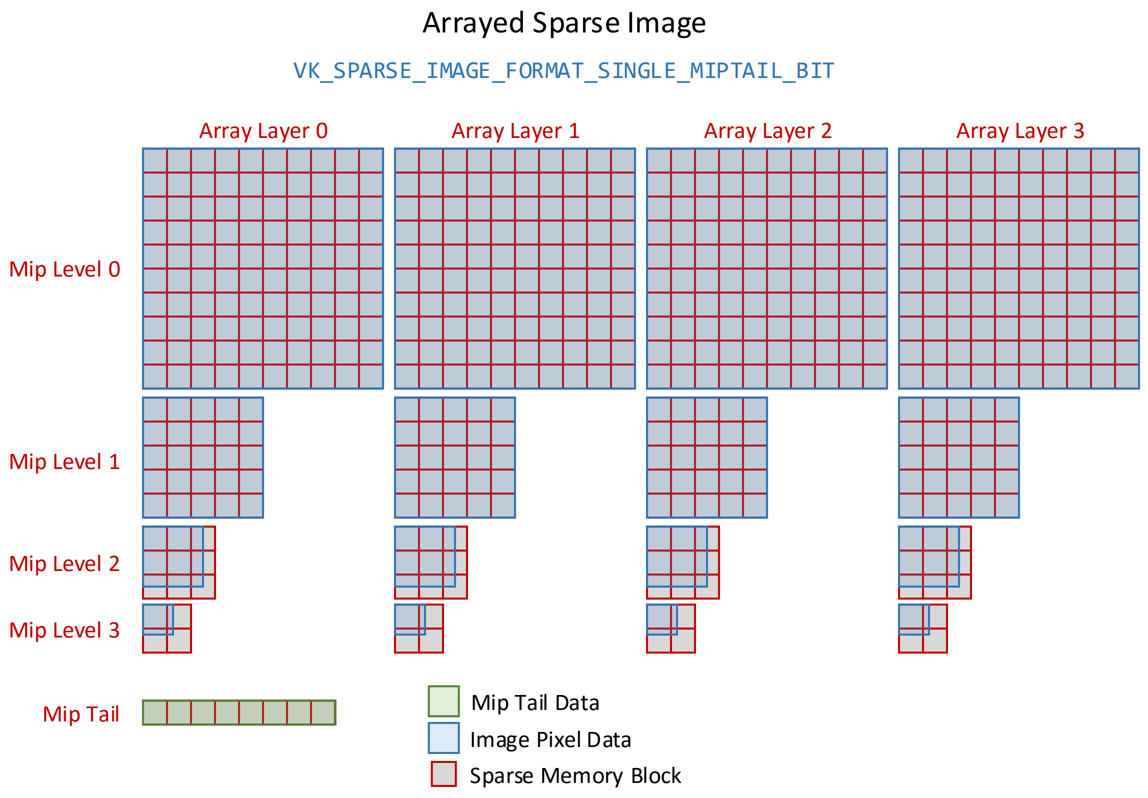


Figure 27.2: Sparse Image with Single Mip Tail

When VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT is present all array layers will share a single mip tail region.

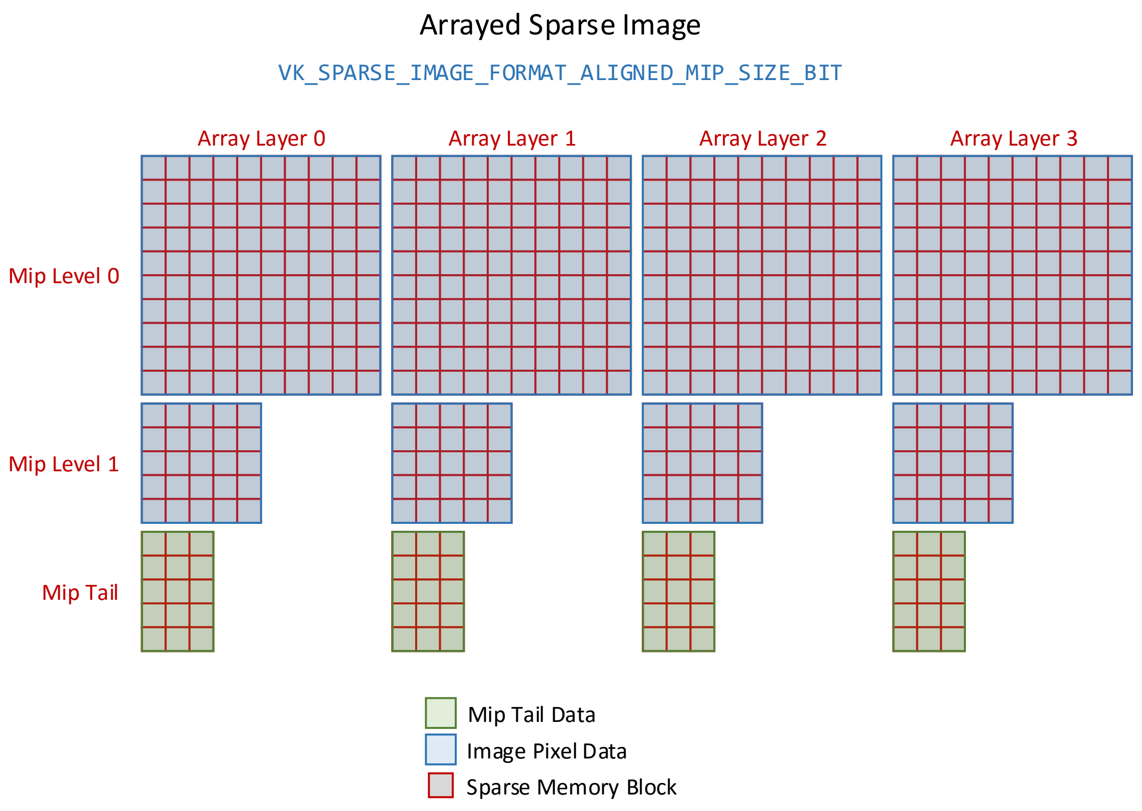


Figure 27.3: Sparse Image with Aligned Mip Size

Note
The mip tail regions are presented here in 2D arrays simply for figure size reasons. Each mip tail is logically a single array of blocks with an implementation-dependent mapping of pixels to blocks.

When VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT is present the first mip level that would contain partially used blocks begins the mip tail region. This level and all subsequent levels are placed in the mip tail. Only the first *N* mip levels that are an exact multiple of the sparse block size can be bound and unbound on a block basis.

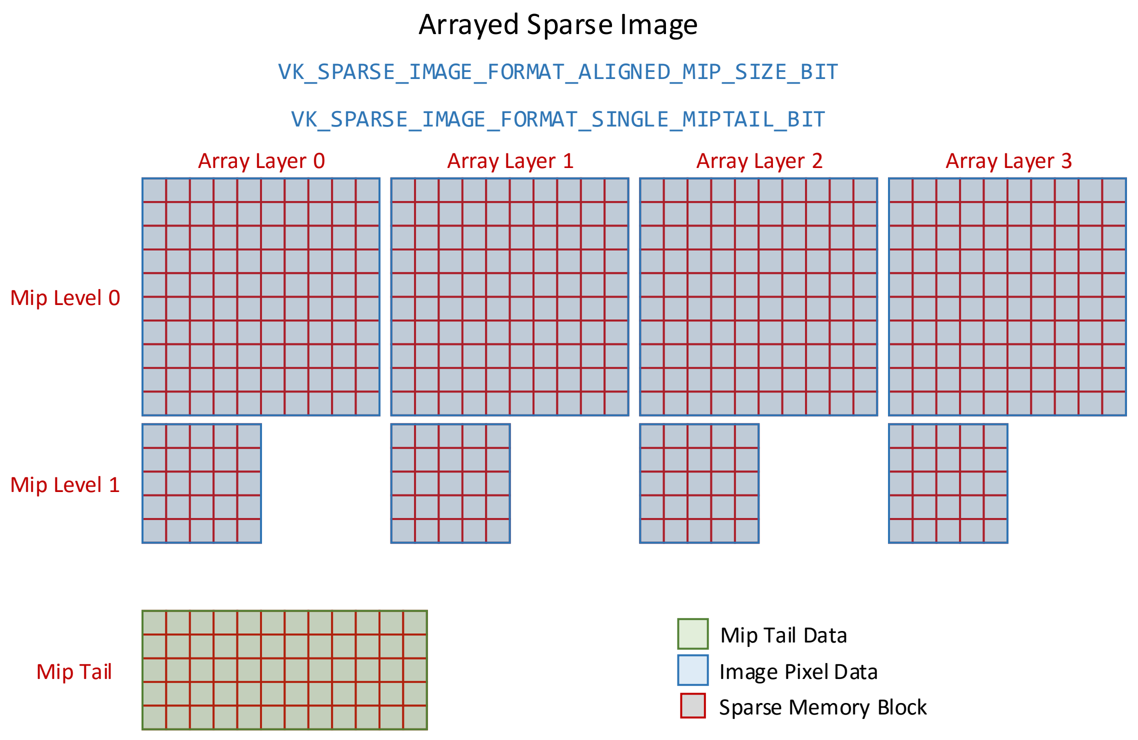


Figure 27.4: Sparse Image with Aligned Mip Size and Single Mip Tail

Note
The mip tail region is presented here in a 2D array simply for figure size reasons. It is logically a single array of blocks with an implementation-dependent mapping of pixels to blocks.

When both VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT and VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT are present the constraints from each of these flags are in effect.

27.4.3 Standard Image Block Sizes

Standard sparse image block sizes are defined by Vulkan and depend on the format of the image. Layout of pixels within a block is implementation dependent. All currently defined standard block shapes are 64 KB in size.

For compressed pixel formats (e.g. VK_FORMAT_BC5_UNORM_BLOCK), the pixel size is the size of the compression block (128-bit for BC5).

Table 27.1: Standard Sparse Image Block Shapes (Single Sample)

PIXEL SIZE (bits)	Block Shape (2D)	Block Shape (3D)
8-Bit	256 x 256 x 1	64 x 32 x 32

Table 27.1: (continued)

PIXEL SIZE (bits)	Block Shape (2D)	Block Shape (3D)
16-Bit	256 x 128 x 1	32 x 32 x 32
32-Bit	128 x 128 x 1	32 x 32 x 16
64-Bit	128 x 64 x 1	32 x 16 x 16
128-Bit	64 x 64 x 1	16 x 16 x 16

Table 27.2: Standard Sparse Image Block Shapes (MSAA)

PIXEL SIZE (bits)	Block Shape (2X)	Block Shape (4X)	Block Shape (8X)	Block Shape (16X)
8-Bit	128 x 256 x 1	128 x 128 x 1	64 x 128 x 1	64 x 64 x 1
16-Bit	128 x 128 x 1	128 x 64 x 1	64 x 64 x 1	64 x 32 x 1
32-Bit	64 x 128 x 1	64 x 64 x 1	32 x 64 x 1	32 x 32 x 1
64-Bit	64 x 64 x 1	64 x 32 x 1	32 x 32 x 1	32 x 16 x 1
128-Bit	32 x 64 x 1	32 x 32 x 1	16 x 32 x 1	16 x 16 x 1

Implementations that support the standard block shape for all applicable formats may advertise the following VkPhysicalDeviceSparseProperties:

- *`residencyStandard2DBlockShape`*
- *`residencyStandard2DMultisampleBlockShape`*
- *`residencyStandard3DBlockShape`*

Reporting each of these features does *not* imply that all possible image types are supported as sparse. Instead, this indicates that no supported sparse image of the corresponding type will use a custom block size.

27.4.4 Custom Image Block Sizes

An implementation that does not support the standard image block sizes may choose to support a custom block size instead. This custom block size will have the pixel region size reported in VkSparseImageFormatProperties::*`imageGranularity`*. As with standard block sizes, the byte-size of the custom block size will be reported in VkMemoryRequirements::*`alignment`*.

Custom block sizes are reported through **`vkGetPhysicalDeviceSparseImageFormatProperties`** and **`vkGetImageSparseMemoryRequirements`**.

An implementation must not support *both* the standard block size and a custom block size for the same image. The standard size must be used if it is supported.

27.4.5 Multiple Aspects

Partially resident images are allowed to report separate sparse properties for different aspects of the image. One example is for depth-stencil images where the implementation separates the depth and stencil data into separate planes. Another reason for multiple aspects is to allow the application to manage memory allocation for implementation-private *metadata* associated with the image. See the figure below:

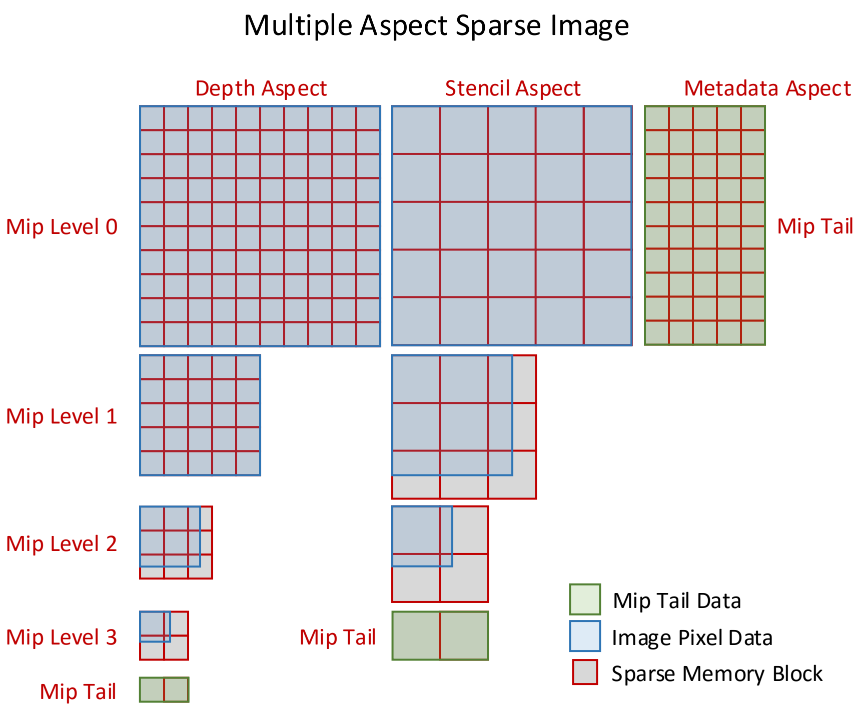


Figure 27.5: Multiple Aspect Sparse Image

Note
The mip tail regions are presented here in 2D arrays simply for figure size reasons. Each mip tail is logically a single array of blocks with an implementation-dependent mapping of pixels to blocks.

In the figure above the depth, stencil, and metadata aspects all have unique sparse properties. The per-pixel stencil data is $^1/_4$ the size of the depth data, hence the stencil sparse blocks include $4x$ the number of pixels. The block byte-size for all of the aspects is identical and defined by VkMemoryRequirements::`alignment`.

27.4.5.1 Metadata

The metadata aspect of an image has the following constraints:

- All metadata is reported in the mip tail region of the metadata aspect.
- All metadata must be bound prior to device use of the sparse image.

27.5 Sparse Memory Aliasing

By default sparse resources have the same aliasing rules as non-sparse resources. See Memory Aliasing for more information.

VkDevice objects that have the sparseResidencyAliased feature enabled are able to use the VK_BUFFER_CREATE_SPARSE_ALIASED_BIT and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT flags for resource creation. These flags allow resources to access physical memory that may be mapped into multiple locations within one or more sparse resources in a *data consistent* fashion. This means that reading physical memory from multiple aliased locations will return the same value.

Care must be taken when performing a write operation to aliased physical memory. Memory dependencies must be used to separate writes to one alias from reads or writes to another alias. Writes to aliased memory that are not properly guarded against accesses to different aliases will have undefined results for all accesses to the aliased memory.

Applications that wish to make use of data consistent memory aliasing must abide by the following guidelines:

- All sparse resources that map aliased physical memory must be created with the VK_BUFFER_CREATE_SPARSE_ALIASED_BIT / VK_IMAGE_CREATE_SPARSE_ALIASED_BIT flag.
- All resources that access aliased physical memory must interpret the memory in the same way. This implies the following:
  - Buffers and images cannot alias the same physical memory in a data consistent fashion. The physical memory blocks must be used exclusively by buffers or used exclusively by images for data consistency to be guaranteed.
  - Memory in sparse image mip tail regions cannot access aliased memory in a data consistent fashion.
  - Sparse images that alias the same physical memory must have compatible formats and be using the same block shape in order to access aliased memory in a data consistent fashion.

Failure to follow all of the above guidelines will require the application to abide by the normal, non-sparse resource aliasing rules. In this case memory cannot be accessed in a data consistent fashion.

Note
Enabling sparse resource memory aliasing can be a way to lower physical memory use. It can also **reduce performance** on some implementations. An application developer must test on their target HW and balance the memory / performance trade-offs measured.

27.6 Sparse Resource Implementation Guidelines

This section is Informative. It is included to aid in implementors' understanding of sparse resources.

Device Virtual Address The basic `sparseBinding` feature allows the resource to reserve its own device virtual address at resource creation time rather relying on a bind operation to set this. Without any other creation flags, no other constraints are relaxed compared to normal resources. All pages must be mapped to physical memory before the device accesses the resource.

The *sparseResidency* feature allows the sparse resource to be used even when not all pages are bound to memory. Hardware that supports access to unbound pages without causing a fault may support *sparseResidencyNonResidentStrict*.

Not faulting on access to unbound pages is not enough to support *sparseResidencyNonResidentStrict*. An implementation must also guarantee that reads after writes to unbound regions of the resource always return data for the read as if the memory contains zeros. Depending on the cache implementation of the hardware this may not always be possible.

Hardware that does not fault, but does not guarantee correct read values will not require dummy pages, but also must not support *sparseResidencyNonResidentStrict*.

Hardware that cannot access unbound pages without causing a fault will require the implementation to map the entire device virtual address range to physical memory. Any pages that the application does not map may be bound to one (or more) *dummy* physical page(s) allocated by the implementation. Given the following properties:

- A process must not access memory from another process
- Reads returned undefined values

It is sufficient for each host process to allocate these dummy pages and use them for all resources in that process. Implementations may allocate more often (per instance, per device, or per resource).

Binding Memory The byte size reported in VkMemoryRequirements::*size* must be greater than or equal to the amount of physical memory required to fully populate the resource. Some hardware requires *holes* in the device virtual address range that are never accessed. These holes may be included in the *size* reported for the resource.

Including or not including the device virtual address holes in the resource size will alter how the implementation provides support for VkSparseImageOpaqueMemoryBindInfo. This operation must be supported for all sparse images, even ones created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

- If the holes are included in the size, this bind function becomes very easy. In most cases the *resourceOffset* is simply a device virtual address offset and the implementation does not require any sophisticated logic to determine what device virtual address to bind. The cost is that the application can allocate more physical memory for the resource than it needs.
- If the holes are not included in the size, the application can allocate less physical memory than otherwise for the resource. However, in this case the implementation must account for the holes when mapping *resourceOffset* to the actual device virtual address intended to be mapped.

Note
If the application always uses VkSparseImageMemoryBindInfo to bind memory for the non-mip-tail levels, any holes that are present in the resource size may never be bound.
Since VkSparseImageMemoryBindInfo uses pixel locations to determine which device virtual addresses to bind, it is impossible to reference device virtual address holes with this operation.

Binding Metadata Memory All metadata for sparse images have their own sparse properties and is embedded in the mip tail region for said properties. See the Multiaspect section for details.

Given that metadata is in a mip tail region, and the mip tail region must be reported as contiguous (either globally or per-array-layer), some implementations will have to resort to complicated offset $\rightarrow$ device virtual address mapping for handling VkSparseImageOpaqueMemoryBindInfo.

To make this easier on the implementation, the VK_SPARSE_MEMORY_BIND_METADATA_BIT explicitly denotes when metadata is bound with VkSparseImageOpaqueMemoryBindInfo. When this flag is not present, the `resourceOffset` may be treated as a strict device virtual address offset.

When VK_SPARSE_MEMORY_BIND_METADATA_BIT is present, the `resourceOffset` must have been derived explicitly from the `imageMipTailOffset` in the sparse resource properties returned for the metadata aspect. By manipulating the value returned for `imageMipTailOffset`, the `resourceOffset` does not have to correlate directly to a device virtual address offset, and may instead be whatever values makes it easiest for the implementation to derive the correct device virtual address.

27.7 Sparse Resource API

The APIs related to sparse resources are grouped into the following categories:

- Physical Device Features
- Physical Device Sparse Properties
- Sparse Image Format Properties
- Sparse Resource Creation
- Sparse Resource Memory Requirements
- Binding Resource Memory

27.7.1 Physical Device Features

Some sparse-resource related features are reported and enabled in VkPhysicalDeviceFeatures. These features must be supported and enabled on the VkDevice object before applications can use them. See Physical Device Features for information on how to get and set enabled device features, and for more detailed explanations of these features.

27.7.1.1 Sparse Physical Device Features

- `sparseBinding`: Support for creating VkBuffer and VkImage objects with the VK_BUFFER_CREATE_SPARSE_BINDING_BIT and VK_IMAGE_CREATE_SPARSE_BINDING_BIT, respectively.
- `sparseResidencyBuffer`: Support for creating VkBuffer objects with the VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT.
- `sparseResidencyImage2D`: Support for creating 2D single-sampled VkImage objects with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- `sparseResidencyImage3D`: Support for creating 3D VkImage objects with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- `sparseResidency2Samples`: Support for creating 2D VkImage objects with 2 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- `sparseResidency4Samples`: Support for creating 2D VkImage objects with 4 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.

- *sparseResidency8Samples*: Support for creating 2D VkImage objects with 8 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- *sparseResidency16Samples*: Support for creating 2D VkImage objects with 16 samples and VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT.
- *sparseResidencyAliased*: Support for creating VkBuffer and VkImage objects with the VK_BUFFER_CREATE_SPARSE_ALIASED_BIT and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT, respectively.

27.7.2 Physical Device Sparse Properties

Some features of the implementation are not possible to disable, and are reported to allow applications to alter their sparse resource usage accordingly. These read-only capabilites are reported in the *sparseProperties* member of `VkPhysicalDeviceProperties`. The definition of *sparseProperties* is

```
typedef struct VkPhysicalDeviceSparseProperties {
    VkBool32                                    residencyStandard2DBlockShape;
    VkBool32                                     ↩
        residencyStandard2DMultisampleBlockShape;
    VkBool32                                    residencyStandard3DBlockShape;
    VkBool32                                    residencyAlignedMipSize;
    VkBool32                                    residencyNonResidentStrict;
} VkPhysicalDeviceSparseProperties;
```

- *residencyStandard2DBlockShape* is VK_TRUE if the physical device will access all single-sample 2D sparse resources using the standard block shapes (based on image format), as described in the Standard Sparse Image Block Shapes (Single Sample) table. If this property is not supported the value returned in the *imageGranularity* member of the VkSparseImageFormatProperties structure for single-sample 2D images is not required to match the standard image block sizes listed in the table.
- *residencyStandard2DMultisampleBlockShape* is VK_TRUE if the physical device will access all multisample 2D sparse resources using the standard block shapes (based on image format), as described in the Standard Sparse Image Block Shapes (MSAA) table. If this property is not supported, the value returned in the *imageGranularity* member of the VkSparseImageFormatProperties structure for multisample 2D images is not required to match the standard image block sizes listed in the table.
- *residencyStandard3DBlockShape* is VK_TRUE if the physical device will access all 3D sparse resources using the standard block shapes (based on image format), as described in the Standard Sparse Image Block Shapes (Single Sample) table. If this property is not supported, the value returned in the *imageGranularity* member of the VkSparseImageFormatProperties structure for 3D images is not required to match the standard image block sizes listed in the table.
- *residencyAlignedMipSize* is VK_TRUE if images with mip level dimensions that are not a multiple of a block size may be placed in the mip tail. If this property is not reported, only mip levels with dimensions smaller than the value of the *imageGranularity* member of the VkSparseImageFormatProperties structure will be placed in the mip tail. If this property is reported the implementation is allowed to return VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT in the *flags* member of VkSparseImageFormatProperties, indicating that mip level dimensions that are not a multiple of a block size will be placed in the mip tail.
- *residencyNonResidentStrict* specifies whether the physical device can consistently access non-resident regions of a resource. If this property is VK_TRUE, access to non-resident regions of resources will be guaranteed to return values as if the resource were populated with 0. Writes to non-resident regions will be discarded.

27.7.3 Sparse Image Format Properties

Given that the sparse image block size is implementation-dependent, **vkGetPhysicalDeviceSparseImageFormatProperties** can be used to query for sparse image format properties prior to resource creation. This command is used to check whether a given set of sparse image parameters is supported and what the sparse block shape will be.

27.7.3.1 Sparse Image Format Properties API

```
typedef struct VkSparseImageFormatProperties {
    VkImageAspectFlags                          aspectMask;
    VkExtent3D                                  imageGranularity;
    VkSparseImageFormatFlags                    flags;
} VkSparseImageFormatProperties;
```

- *aspectMask* is a `VkImageAspectFlags` specifying which components of the image the properties apply to.
- *imageGranularity* is the width, height, and depth of the block in pixels / compression blocks
- *flags* is a bitmask specifying additional information about the sparse resource. Bits which can be set include:

```
typedef enum VkSparseImageFormatFlagBits {
    VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT = 0x00000001,
    VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT = 0x00000002,
    VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT = 0x00000004,
} VkSparseImageFormatFlagBits;
```

  - If VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT is set, the image uses a single mip tail region for all array layers.
  - If VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT is set, the first mip level that is not an exact multiple of the sparse image block size begins the mip tail region.
  - If VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT is set, the image uses a non-standard sparse block size, and the *imageGranularity* values do not match the standard block size for the given pixel format.

vkGetPhysicalDeviceSparseImageFormatProperties returns an array of VkSparseImageFormatProperties. Each element will describe properties for one set of image aspects that are bound simultaneously in the image. This is usually one element for each aspect in the image, but for interleaved depth/stencil images there is only one element describing the combined aspects.

```
void vkGetPhysicalDeviceSparseImageFormatProperties(
    VkPhysicalDevice                            physicalDevice,
    VkFormat                                    format,
    VkImageType                                 type,
    VkSampleCountFlagBits                       samples,
    VkImageUsageFlags                           usage,
    VkImageTiling                               tiling,
    uint32_t*                                   pPropertyCount,
    VkSparseImageFormatProperties*              pProperties);
```

- *physicalDevice* is the owning device.

- *format* is the image format.
- *type* is the dimensionality of image.
- *samples* is the number of samples per pixel as defined in `VkSampleCountFlagBits`.
- *usage* are possible usages for image.
- *tiling* is the image tiling layout.
- *pPropertyCount* points to a variable specifying the length of the *pProperties* array. On return this variable is overwritten with the number of structures actually written to *pProperties*.
- *pProperties* is an array of *pPropertyCount* VkSparseImageFormatProperties format property structures in which values are returned.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *format* must be a valid `VkFormat` value
- *type* must be a valid `VkImageType` value
- *samples* must be a valid `VkSampleCountFlagBits` value
- *usage* must be a valid combination of `VkImageUsageFlagBits` values
- *usage* must not be `0`
- *tiling* must be a valid `VkImageTiling` value
- *pPropertyCount* must be a pointer to a uint32_t value
- If the value referenced by *pPropertyCount* is not `0`, and *pProperties* is not `NULL`, *pProperties* must be a pointer to an array of *pPropertyCount* VkSparseImageFormatProperties structures
- If *format* is an integer format, samples must be one of the bit flag values specified in the value of VkPhysicalDeviceLimits::*sampledImageIntegerSampleCounts*
- If *format* is a non-integer color format, samples must be one of the bit flag values specified in the value of VkPhysicalDeviceLimits::*sampledImageColorSampleCounts*
- If *format* is a depth format, samples must be one of the bit flag values specified in the value of VkPhysicalDeviceLimits::*sampledImageDepthSampleCounts*
- If *format* is a stencil format, samples must be one of the bit flag values specified in the value of VkPhysicalDeviceLimits::*sampledImageStencilSampleCounts*
- If *usage* includes VK_IMAGE_USAGE_STORAGE_BIT, samples must be one of the bit flag values specified in the value of VkPhysicalDeviceLimits::*storageImageSampleCounts*

If the input value of *pPropertyCount* is less than the number of groups of aspects present in the image, no data is written to *pProperties*. Instead, *pPropertyCount* will be updated with the size of the array required.

If VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT is not supported for the given arguments, *pPropertyCount* will be set to zero upon return, and no data will be written to *pProperties*.

Multiple aspects are returned for DepthStencil images that are implemented as separate planes by the implementation. The depth and stencil data planes each have unique VkSparseImageFormatProperties data.

DepthStencil images with depth and stencil data interleaved into a single plane will return a single VkSparseImageFormatProperties structure with the *aspectMask* set to VK_IMAGE_ASPECT_DEPTH_BIT | VK_IMAGE_ASPECT_STENCIL_BIT.

27.7.4 Sparse Resource Creation

Sparse resources require that one or more sparse feature flags be specified (as part of the VkPhysicalDeviceFeatures structure described previously in the Physical Device Features section) at CreateDevice time. When the appropriate device features are enabled, the VK_BUFFER_CREATE_SPARSE_* and VK_IMAGE_CREATE_SPARSE_* flags can be used. See `vkCreateBuffer` and `vkCreateImage` for details of the resource creation APIs.

Note
Specifying VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT or VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT implies specifying VK_BUFFER_CREATE_SPARSE_BINDING_BIT or VK_IMAGE_CREATE_SPARSE_BINDING_BIT, respectively, as well. This means that resources created with either (or both) flag(s) can be used with the sparse binding command (**`vkQueueBindSparse`**).

27.7.5 Sparse Resource Memory Requirements

Sparse resources have specific memory requirements related to binding sparse memory. These memory requirements are reported differently for VkBuffer objects and VkImage objects.

27.7.5.1 Buffer and Fully-Resident Images

Buffers (both fully and partially resident) and fully-resident images can be mapped using only the data from VkMemoryRequirements. For all sparse resources the VkMemoryRequirements::*alignment* member denotes both the bindable block byte-size and required alignment of VkDeviceMemory.

27.7.5.2 Partially Resident Images

Partially resident images have a different method for binding memory. As with buffers and fully resident images, the VkMemoryRequirements::*alignment* field denotes the bindable block byte-size for the image.

Requesting sparse memory requirements for VkImage objects using **`vkGetImageSparseMemoryRequirements`** will return an array of 1 or more VkSparseImageMemoryRequirements structures. Each structure describes the sparse memory requirements for an aspect of the image.

The sparse image must have been created using the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag to retrieve valid sparse image memory requirements.

27.7.5.3 Sparse Image Memory Requirements

```
typedef struct VkSparseImageMemoryRequirements {
    VkSparseImageFormatProperties                  formatProperties;
    uint32_t                                       imageMipTailFirstLod;
    VkDeviceSize                                   imageMipTailSize;
    VkDeviceSize                                   imageMipTailOffset;
    VkDeviceSize                                   imageMipTailStride;
} VkSparseImageMemoryRequirements;
```

- *formatProperties.aspectMask* is the set of aspects of the image that this sparse memory requirement applies to. This will usually have a single aspect specified. However, depth-stencil images may have depth and stencil data interleaved into the same memory blocks, in which case both VK_IMAGE_ASPECT_DEPTH_BIT and VK_IMAGE_ASPECT_STENCIL_BIT would be present.
- *formatProperties.imageGranularity* describes the size of a single bindable block in pixel units. For aspect VK_IMAGE_ASPECT_METADATA_BIT, this region size will be zero pixels. All metadata is located in the mip tail region.
- *formatProperties.flags* contains members of `VkSparseImageFormatFlags`:
  - If VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT is set the image uses a single mip tail region for all array layers.
  - If VK_SPARSE_IMAGE_FORMAT_ALIGNED_MIP_SIZE_BIT is set the image mip levels must be an exact multiple of the sparse image block size for levels not located in the mip tail.
  - If VK_SPARSE_IMAGE_FORMAT_NONSTANDARD_BLOCK_SIZE_BIT is set the image uses a non-standard sparse block size. The *formatProperties.imageGranularity* values do not match the standard block size for image's given pixel format.
- *imageMipTailFirstLod* Is the first mipLevel at which subresources are included in the mip tail block.
- *imageMipTailSize* is the memory size (in bytes) of the mip tail block. If *formatProperties.flags* contains VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT, this is the size of the whole mip tail, otherwise this is the size of the mip tail for each array layer. This value is guaranteed to be a multiple of the block byte-size.
- *imageMipTailOffset* is the opaque memory offset used with `VkSparseImageOpaqueMemoryBindInfo` to bind the mip tail region(s).
- *imageMipTailStride* is the offset stride between each array-layer's mip tail, if *formatProperties.flags* does not contain VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT (otherwise the value is undefined).

Query sparse memory requirements for an image by calling:

```
void vkGetImageSparseMemoryRequirements(
    VkDevice                                       device,
    VkImage                                        image,
    uint32_t*                                      pSparseMemoryRequirementCount,
    VkSparseImageMemoryRequirements*               pSparseMemoryRequirements);
```

- *device* is the VkDevice object that owns the *image*.
- *image* is the VkImage object to get requirements from.

- *`pSparseMemoryRequirementCount`* is the number of requirements in the *`image`*, and is both an input and output parameter:
  - [in] Number of VkSparseImageMemoryRequirements in the *`pSparseMemoryRequirements`* array.
  - [out] Number of VkSparseImageMemoryRequirements required for the operation to complete.
- *`pSparseMemoryRequirements`* is an array of *`pSparseMemoryRequirementCount`* VkSparseImageMemoryRequirements to be written by the API.

Valid Usage

- *`device`* must be a valid VkDevice handle
- *`image`* must be a valid VkImage handle
- *`pSparseMemoryRequirementCount`* must be a pointer to a uint32_t value
- If *`pSparseMemoryRequirements`* is not `NULL`, *`pSparseMemoryRequirements`* must be a pointer to an array of *`pSparseMemoryRequirementCount`* VkSparseImageMemoryRequirements structures
- If *`pSparseMemoryRequirements`* is not `NULL`, the value referenced by *`pSparseMemoryRequirementCount`* must be greater than `0`
- *`image`* must have been created, allocated or retrieved from *`device`*
- Each of *`device`* and *`image`* must have been created, allocated or retrieved from the same VkPhysicalDevice
- *`image`* must have been created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT flag

If the value pointed to by *`pSparseMemoryRequirementCount`* is less than the number of memory requirements, *`pSparseMemoryRequirements`* is not written to.

If the image was not created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT then *`pSparseMemoryRequirementCount`* will be set to zero and *`pSparseMemoryRequirements`* will not be written to.

Note
It is legal for an implementation to report a larger value in VkMemoryRequirements::*`size`* than would be obtained by adding together memory sizes for all VkSparseImageMemoryRequirements returned by **`vkGetImageSparseMemoryRequirements`**. This may occur when the hardware requires unused padding in the address range describing the resource.

27.7.6 Binding Resource Memory

Non-sparse objects are backed by a single physical allocation prior to device use (via **`vkBindImageMemory`** or **`vkBindBufferMemory`**), and their backing must not be changed. Sparse objects are created as virtual-only allocations. Regions of the resource that the application intends to use must be *bound* to physical memory — this binding is effected by updating page tables.

Note
It is important to note that freeing a VkDeviceMemory object with **`vkFreeMemory`** will not cause resources (or resource regions) bound to the memory object to become unbound. Access to resources that are bound to memory objects that have been freed will result in undefined behavior, potentially including application termination.
Implementations must ensure that no access to physical memory owned by the OS or another process will occur in this scenario. In other words, accessing resources bound to freed memory may crash the application, but must not crash the OS or read non-process-accessible memory.

Sparse memory bindings execute on a queue that includes the VK_QUEUE_SPARSE_BINDING_BIT bit. Applications must use synchronization primitives to guarantee that other queues do not access pages of memory concurrently with a binding change. Accessing memory in a page while it is being rebound results in undefined behavior. It is valid to access other pages of the same resource while a bind operation is executing.

Note
Implementations must provide a guarantee that simultaneously binding pages while another queue accesses those same pages via a sparse resource must not access memory owned by another process or otherwise corrupt the system.

While some implementations may include VK_QUEUE_SPARSE_BINDING_BIT support in queue families that also include graphics and compute support, other implementations may only expose a VK_QUEUE_SPARSE_BINDING_BIT-only queue family. In either case, applications must use synchronization primitives to explicitly request any ordering dependencies between sparse memory binding operations and other graphics/compute/transfer operations, as sparse binding operations are not automatically ordered against command buffer execution, even within a single queue.

When binding memory explicitly for the VK_IMAGE_ASPECT_METADATA_BIT the application must use the VK_SPARSE_MEMORY_BIND_METADATA_BIT in the VkSparseMemoryBind::*`flags`* field when binding memory. Binding memory for metadata is done the same as binding memory for the mip tail, with the addition of the VK_SPARSE_MEMORY_BIND_METADATA_BIT flag.

Binding the mip tail for any aspect must only be performed using `VkSparseImageOpaqueMemoryBindInfo`. If *`formatProperties.flags`* contains VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT, then it can be bound with a single `VkSparseMemoryBind` structure, with resourceOffset = imageMipTailOffset and size = imageMipTailSize.

If *`formatProperties.flags`* does *not* contain VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT then the offset for the mip tail in each array layer is given as:

```
arrayMipTailOffset = imageMipTailOffset + arrayLayer * imageMipTailStride;
```

and the mip tail can be bound with **`layerCount`** `VkSparseMemoryBind` structures, each using *`size`* = *`imageMipTailSize`* and *`resourceOffset`* = *`arrayMipTailOffset`* as defined above.

Sparse memory binding is handled by the following APIs and related data structures.

27.7.6.1 Sparse Memory Binding Functions

```
typedef enum VkSparseMemoryBindFlagBits {
    VK_SPARSE_MEMORY_BIND_METADATA_BIT = 0x00000001,
} VkSparseMemoryBindFlagBits;
```

```
typedef VkFlags VkSparseMemoryBindFlags;
```

- VK_SPARSE_MEMORY_BIND_METADATA_BIT is used to indicate that the memory being bound is *only* for the metadata aspect.

```
typedef struct VkSparseMemoryBind {
    VkDeviceSize                                  resourceOffset;
    VkDeviceSize                                  size;
    VkDeviceMemory                                memory;
    VkDeviceSize                                  memoryOffset;
    VkSparseMemoryBindFlags                       flags;
} VkSparseMemoryBind;
```

- `resourceOffset` is the offset into the resource.
- `size` is the size of the memory region to be bound.
- `memory` is the VkDeviceMemory object that the pages of the resource are mapped to. If `memory` is VK_NULL_HANDLE, the pages are unmapped.
- `memoryOffset` is the offset into VkDeviceMemory object. If `memory` is VK_NULL_HANDLE, this value is ignored.
- `flags` are sparse memory binding flags.

Valid Usage

- If `memory` is not VK_NULL_HANDLE, `memory` must be a valid VkDeviceMemory handle
- `flags` must be a valid combination of `VkSparseMemoryBindFlagBits` values
- If `memory` is not VK_NULL_HANDLE, `memory` and `memoryOffset` must match the memory requirements of the resource, as described in section Section 11.6
- If `memory` is not VK_NULL_HANDLE, `memory` must not have been created with a memory type that reports VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT bit set
- `resourceOffset` must be less than the size of the resource
- The sum of `resourceOffset` and `size` must be less than or equal to the size of the resource
- `memoryOffset` must be less than the size of `memory`
- The sum of `memoryOffset` and `size` must be less than or equal to the size of `memory`

The *binding range* $[resourceOffset, resourceOffset + size)$ has different constraints based on the value of `flags`. If `flags` contains VK_SPARSE_MEMORY_BIND_METADATA_BIT, the *binding range* must be within the mip tail region of the metadata aspect. This metadata region is defined by:

$$\begin{aligned}metadataRegion = [&imageMipTailOffset + imageMipTailStride \times n,\\ &imageMipTailOffset + imageMipTailStride \times n + imageMipTailSize)\end{aligned}$$

Where *imageMipTailOffset*, *imageMipTailSize*, and *imageMipTailStride* values are from the `VkSparseImageMemoryRequirements` that correspond to the metadata aspect of the image. The term *n* is a valid array layer index for the image.

imageMipTailStride will be zero for aspects where VkSparseImageMemoryRequirements::*formatProperties.flags* contains VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT.

If *flags* does not contain VK_SPARSE_MEMORY_BIND_METADATA_BIT, the *binding range* must be within the range $[0, VkMemoryRequirements :: size)$.

```
typedef struct VkSparseBufferMemoryBindInfo {
    VkBuffer                                    buffer;
    uint32_t                                    bindCount;
    const VkSparseMemoryBind*                   pBinds;
} VkSparseBufferMemoryBindInfo;
```

This describes a binding operation that can be used with all VkBuffer objects created with the VK_BUFFER_CREATE_SPARSE_BINDING_BIT to map memory regions.

- *buffer* is the VkBuffer object to be bound.
- *bindCount* is the number of VkSparseMemoryBind structures in the *pBinds* array.
- *pBinds* is a pointer to array of VkSparseMemoryBind structures.

Valid Usage

- *buffer* must be a valid VkBuffer handle
- *pBinds* must be a pointer to an array of *bindCount* valid VkSparseMemoryBind structures
- The value of *bindCount* must be greater than `0`

```
typedef struct VkSparseImageOpaqueMemoryBindInfo {
    VkImage                                     image;
    uint32_t                                    bindCount;
    const VkSparseMemoryBind*                   pBinds;
} VkSparseImageOpaqueMemoryBindInfo;
```

This describes a binding operation that can be used with all VkImage objects created with the VK_IMAGE_CREATE_SPARSE_BINDING_BIT to map memory regions.

- *image* is the VkImage object to be bound.

- `bindCount` is the number of VkSparseMemoryBind structures in the `pBinds` array.
- `pBinds` is a pointer to array of VkSparseMemoryBind structures.

Valid Usage

- `image` must be a valid VkImage handle
- `pBinds` must be a pointer to an array of `bindCount` valid VkSparseMemoryBind structures
- The value of `bindCount` must be greater than `0`
- For any given element of `pBinds`, if the `flags` member of that element contains VK_SPARSE_MEMORY_BIND_METADATA_BIT, the *binding range* defined must be within the mip tail region of the metadata aspect of `image`

Note

This operation is normally used to bind memory to fully-resident sparse images or for mip tail regions of partially resident images. However, it can also be used to bind *all* memory for partially resident images.

In the normal case (`flags` does not contain VK_SPARSE_MEMORY_BIND_METADATA_BIT), the `resourceOffset` is in the range [0, VkMemoryRequirements::`size`]. This range includes data from all aspects of the image, including metadata. For most implementations this will probably mean that the `resourceOffset` is a simple device address offset within the resource. It is possible for an application to bind a range of memory that includes both resource data and metadata. However, the application would not know what part of the image the memory is used for, or even that it is being used for metadata.

When `flags` contains VK_SPARSE_MEMORY_BIND_METADATA_BIT, the binding range specified must be within the mip tail region of the metadata aspect. In this case the `resourceOffset` is not required to be a simple device address offset within the resource. However, it *is* defined to be within [imageMipTailOffset, imageMipTailOffset + imageMipTailSize) for the metadata aspect. See `VkSparseMemoryBind` for the full constraints on binding region with this flag present.

```
typedef struct VkSparseImageMemoryBind {
    VkImageSubresource                            subresource;
    VkOffset3D                                    offset;
    VkExtent3D                                    extent;
    VkDeviceMemory                                memory;
    VkDeviceSize                                  memoryOffset;
    VkSparseMemoryBindFlags                       flags;
} VkSparseImageMemoryBind;
```

- `subresource` is the aspectMask and region of interest in the image.
- `offset` are the coordinates of the first texel within the subresource to bind.
- `extent` is the size in texels of the region within the subresource to bind. The extent must be a multiple of the block size, except when binding blocks along the edge of a subresource it can instead be such that any coordinate of $offset + extent$ equals the subresource size in that dimension.

- *memory* is the VkDeviceMemory object that the pages of the image are mapped to. If *memory* is VK_NULL_HANDLE, the pages are unmapped.
- *memoryOffset* is an offset into VkDeviceMemory object. If *memory* is VK_NULL_HANDLE, this value is ignored.
- *flags* are sparse memory binding flags.

Valid Usage

- *subresource* must be a valid VkImageSubresource structure
- If *memory* is not VK_NULL_HANDLE, *memory* must be a valid VkDeviceMemory handle
- *flags* must be a valid combination of `VkSparseMemoryBindFlagBits` values
- If the sparse aliased residency feature is not enabled, and if any other resources are bound to ranges of *memory*, the range of *memory* being bound must not overlap with those bound ranges
- *memory* and *memoryOffset* must match the memory requirements of the calling command's *image*, as described in section Section 11.6
- *subresource* must be a valid subresource for *image* (see Section 11.5)
- *offset.x* must be a multiple of the block width (VkSparseImageFormatProperties::*imageGranularity.width*) of the image
- *extent.width* must either be a multiple of the block width of the image, or else *extent.width* + *offset.x* must equal the width of the image subresource
- *offset.y* must be a multiple of the block height (VkSparseImageFormatProperties::*imageGranularity.height*) of the image
- *extent.height* must either be a multiple of the block height of the image, or else *extent.height* + *offset.y* must equal the height of the image subresource
- *offset.z* must be a multiple of the block depth (VkSparseImageFormatProperties::*imageGranularity.depth*) of the image
- *extent.depth* must either be a multiple of the block depth of the image, or else *extent.depth* + *offset.z* must equal the depth of the image subresource

```
typedef struct VkSparseImageMemoryBindInfo {
    VkImage                                      image;
    uint32_t                                     bindCount;
    const VkSparseImageMemoryBind*               pBinds;
} VkSparseImageMemoryBindInfo;
```

This describes a binding operation that can only be used with VkImage objects created with the VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT to map sparse image blocks:

- *image* is the VkImage object to be bound
- *bindCount* is the number of VkSparseImageMemoryBind structures in pBinds array
- *pBinds* is a pointer to array of VkSparseImageMemoryBind structures

Valid Usage

- *image* must be a valid VkImage handle
- *pBinds* must be a pointer to an array of *bindCount* valid VkSparseImageMemoryBind structures
- The value of *bindCount* must be greater than 0

```
typedef struct VkBindSparseInfo {
    VkStructureType                             sType;
    const void*                                 pNext;
    uint32_t                                    waitSemaphoreCount;
    const VkSemaphore*                          pWaitSemaphores;
    uint32_t                                    bufferBindCount;
    const VkSparseBufferMemoryBindInfo*         pBufferBinds;
    uint32_t                                    imageOpaqueBindCount;
    const VkSparseImageOpaqueMemoryBindInfo*    pImageOpaqueBinds;
    uint32_t                                    imageBindCount;
    const VkSparseImageMemoryBindInfo*          pImageBinds;
    uint32_t                                    signalSemaphoreCount;
    const VkSemaphore*                          pSignalSemaphores;
} VkBindSparseInfo;
```

- *sType* is the type of this structure.
- *pNext* is NULL or a pointer to an extension-specific structure.
- *waitSemaphoreCount* is the number of semaphores upon which to wait before executing the sparse binding operations for the batch.
- *pWaitSemaphores* is a pointer to an array of semaphores upon which to wait before executing the sparse binding operations in the batch.
- *bufferBindCount* is the number of sparse buffer bindings to perform.
- *pBufferBinds* is an array of VkSparseBufferMemoryBindInfo structures, indicating sparse buffer bindings to perform as described above.
- *imageOpaqueBindCount* is the number of opaque sparse image bindings to perform.
- *pImageOpaqueBinds* is an array of VkSparseImageOpaqueMemoryBindInfo structures, indicating opaque sparse image bindings to perform as described above.
- *imageBindCount* is the number of sparse image bindings to perform.

- *pImageBinds* is an array of `VkSparseImageMemoryBindInfo` structures, indicating sparse image bindings to perform as described above.
- *signalSemaphoreCount* is the number of semaphores to be signaled once the sparse binding operations specified by the structure have completed execution.
- *pSignalSemaphores* is a pointer to an array of semaphores which will be signaled when the sparse binding operations for this batch have completed execution.

Valid Usage

- *sType* must be VK_STRUCTURE_TYPE_BIND_SPARSE_INFO
- *pNext* must be `NULL`
- If *waitSemaphoreCount* is not `0`, *pWaitSemaphores* must be a pointer to an array of *waitSemaphoreCount* valid VkSemaphore handles
- If *bufferBindCount* is not `0`, *pBufferBinds* must be a pointer to an array of *bufferBindCount* valid VkSparseBufferMemoryBindInfo structures
- If *imageOpaqueBindCount* is not `0`, *pImageOpaqueBinds* must be a pointer to an array of *imageOpaqueBindCount* valid VkSparseImageOpaqueMemoryBindInfo structures
- If *imageBindCount* is not `0`, *pImageBinds* must be a pointer to an array of *imageBindCount* valid VkSparseImageMemoryBindInfo structures
- If *signalSemaphoreCount* is not `0`, *pSignalSemaphores* must be a pointer to an array of *signalSemaphoreCount* valid VkSemaphore handles
- Each of the elements of *pWaitSemaphores* and the elements of *pSignalSemaphores* that are valid handles must have been created, allocated or retrieved from the same VkDevice

Sparse binding operations are submitted to a queue for execution via the command:

```
VkResult vkQueueBindSparse(
    VkQueue                                     queue,
    uint32_t                                    bindInfoCount,
    const VkBindSparseInfo*                     pBindInfo,
    VkFence                                     fence);
```

Each batch of sparse binding operations is represented by a list of `VkSparseBufferMemoryBindInfo`, `VkSparseImageOpaqueMemoryBindInfo`, and `VkSparseImageMemoryBindInfo` structures (encapsulated in a `VkBindSparseInfo` structure), each preceded by a list of semaphores upon which to wait before beginning execution of the operations, and followed by a second list of semaphores to signal upon completion of the operations.

queue is the handle of the queue that the sparse binding operations will be executed on.

Each call to **`vkQueueBindSparse`** submits zero or more batches of sparse binding operations to the queue for execution. *bindInfoCount* is used to specify the number of batches to submit. Each batch is represented as an instance of the `VkBindSparseInfo` structure stored in an array, the address of which is passed in *pBindInfo*.

If *fence* is not VK_NULL_HANDLE, then it is the handle of a VkFence object. When all sparse binding operations in *pBindInfo* have completed execution, the status of *fence* is set to signaled, providing certain implicit ordering guarantees.

Valid Usage

- *queue* must be a valid VkQueue handle
- If *bindInfoCount* is not `0`, *pBindInfo* must be a pointer to an array of *bindInfoCount* valid VkBindSparseInfo structures
- If *fence* is not VK_NULL_HANDLE, *fence* must be a valid VkFence handle
- The *queue* must support sparse binding operations
- Each of *queue* and *fence* that are valid handles must have been created, allocated or retrieved from the same VkDevice
- *fence* must be unsignalled
- *fence* must not be associated with any other queue command that has not yet completed execution on that queue

Host Synchronization

- Host access to *queue* must be externally synchronized
- Host access to *pBindInfo*[].pWaitSemaphores[] must be externally synchronized
- Host access to *pBindInfo*[].pSignalSemaphores[] must be externally synchronized
- Host access to *pBindInfo*[].pBufferBinds[].buffer must be externally synchronized
- Host access to *pBindInfo*[].pImageOpaqueBinds[].image must be externally synchronized
- Host access to *pBindInfo*[].pImageBinds[].image must be externally synchronized
- Host access to *fence* must be externally synchronized

Within a batch, a given page of a resource must not be bound more than once. Across batches, if a page is to be bound to one allocation and offset and then to another allocation and offset, then the application must guarantee (usually using semaphores) that the binding operations are executed in the correct order, as well as to order binding operations against the execution of command buffer submissions.

27.8 Examples

The following examples illustrate basic creation of sparse images and binding them to physical memory.

27.8.1 Basic Sparse Resources

This basic example creates a normal VkImage object but uses fine-grained memory allocation to back the resource with multiple memory blocks.

```
VkDevice                device;
VkQueue                 queue;
VkImage                 sparseImage;
VkMemoryRequirements    memoryRequirements = {};
VkDeviceSize            offset = 0;
VkSparseMemoryBind      binds[MAX_CHUNKS] = {}; // MAX_CHUNKS is NOT part of Vulkan
uint32_t                bindCount = 0;

// ...

// Allocate image object
const VkImageCreateInfo sparseImageInfo =
{
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO,         // sType
    NULL,                                        // pNext
    VK_IMAGE_CREATE_SPARSE_BINDING_BIT | ...,    // flags
    ...
};
vkCreateImage(device, &sparseImageInfo, &sparseImage);

// Get memory requirements
vkGetImageMemoryRequirements(
    device,
    sparseImage,
    &memoryRequirements);

// Bind memory in fine-grained fashion, find available memory blocks
// from potentially multiple VkDeviceMemory pools.
// (Illustration purposes only, can be optimized for perf)
while (memoryRequirements.size && bindCount < MAX_CHUNKS)
{
    VkSparseMemoryBind* pBind = &binds[bindCount];
    pBind->resourceOffset = offset;

    AllocateOrGetMemoryBlock(
        device,
        &memoryRequirements,
        &pBind->memory,
        &pBind->memoryOffset,
        &pBind->size);

    // memory blocks must be sized as multiples of the alignment
    assert(IsMultiple(pBind->size, memoryRequirements.alignment));
    assert(IsMultiple(pBind->memoryOffset, memoryRequirements.alignment));

    memoryRequirements.size -= pBind->size;
    offset                  += pBind->size;
    bindCount++;
}

// Ensure all image has backing
if (memoryRequirements.size)
{
    // Error condition - too many chunks
}

const VkSparseImageOpaqueMemoryBindInfo opaqueBindInfo =
{
    sparseImage,                                // image
    bindCount,                                  // bindCount
    binds                                       // pBinds
};

const VkBindSparseInfo bindSparseInfo =
{
    VK_STRUCTURE_TYPE_BIND_SPARSE_INFO,         // sType
    NULL,                                       // pNext
    ...
    1,                                          // imageOpaqueBindCount
    &opaqueBindInfo,                            // pImageOpaqueBinds
    ...
};

// vkQueueBindSparse is application synchronized per queue object.
AcquireQueueOwnership(queue);

// Actually bind memory
vkQueueBindSparse(queue, 1, &bindSparseInfo, VK_NULL_HANDLE);

ReleaseQueueOwnership(queue);
```

27.8.2 Advanced Sparse Resources

This more advanced example creates an arrayed color attachment / texture image and binds only LOD zero and the required metadata to physical memory.

```
VkDevice                            device;
VkQueue                             queue;
VkImage                             sparseImage;
VkMemoryRequirements                memoryRequirements = {};
uint32_t                            sparseRequirementsCount = 0;
VkSparseImageMemoryRequirements*    pSparseReqs = NULL;
VkSparseMemoryBind                  binds[MY_IMAGE_ARRAY_SIZE] = {};
VkSparseImageMemoryBind             imageBinds[MY_IMAGE_ARRAY_SIZE] = {};
uint32_t                            bindCount = 0;

// Allocate image object (both renderable and sampleable)
const VkImageCreateInfo sparseImageInfo =
{
    VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO,        // sType
    NULL,                                       // pNext
    VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT | ..., // flags
    ...
    VK_FORMAT_R8G8B8A8_UNORM,                   // format
    ...
    MY_IMAGE_ARRAY_SIZE,                        // arrayLayers
    ...
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT |
    VK_IMAGE_USAGE_SAMPLED_BIT,                 // usage
    ...
};
vkCreateImage(device, &sparseImageInfo, &sparseImage);

// Get memory requirements
vkGetImageMemoryRequirements(
    device,
    sparseImage,
    &memoryRequirements);

// Get sparse image aspect properties
vkGetImageSparseMemoryRequirements(
    device,
    sparseImage,
    &sparseRequirementsCount,
    NULL);

pSparseReqs = (VkSparseImageMemoryRequirements*)
    malloc(reqCount * sizeof(VkSparseImageMemoryRequirements));

vkGetImageSparseMemoryRequirements(
    device,
    sparseImage,
    &sparseRequirementsCount,
    pSparseReqs);

// Bind LOD level 0 and any required metadata to memory
for (uint32_t i = 0; i < sparseRequirementsCount; ++i)
{
    if (pSparseReqs[i].formatProperties.aspectMask &
        VK_IMAGE_ASPECT_METADATA_BIT)
    {
        // Metadata must not be combined with other aspects
        assert(pSparseReqs[i].formatProperties.aspectMask ==
               VK_IMAGE_ASPECT_METADATA_BIT);

        if (pSparseReqs[i].formatProperties.flags &
            VK_SPARSE_IMAGE_FORMAT_SINGLE_MIPTAIL_BIT)
        {
            VkSparseMemoryBind* pBind = &binds[bindCount];
            pBind->memorySize = pSparseReqs[i].imageMipTailSize;
            bindCount++;

            // ... Allocate memory block

            pBind->resourceOffset = pSparseReqs[i].imageMipTailOffset;
            pBind->memoryOffset = /* allocated memoryOffset */;
            pBind->memory = /* allocated memory */;
            pBind->flags = VK_SPARSE_MEMORY_BIND_METADATA_BIT;
        }
        else
        {
            // Need a mip tail block per array layer.
            for (uint32_t a = 0; a < sparseImageInfo.arrayLayers; ++a)
            {
                VkSparseMemoryBind* pBind = &binds[bindCount];
                pBind->memorySize = pSparseReqs[i].imageMipTailSize;
                bindCount++;

                // ... Allocate memory block

                pBind->resourceOffset = pSparseReqs[i].imageMipTailOffset +
                                        (a * pSparseReqs[i].imageMipTailStride);

                pBind->memoryOffset = /* allocated memoryOffset */;
                pBind->memory = /* allocated memory */
                pBind->flags = VK_SPARSE_MEMORY_BIND_METADATA_BIT;
            }
        }
    }
    else
    {
        // resource data
        VkExtent3D lod0BlockSize =
        {
            AlignedDivide(
                sparseImageInfo.extent.width,
                pSparseReqs[i].formatProperties.imageGranularity.width);
            AlignedDivide(
                sparseImageInfo.extent.height,
                pSparseReqs[i].formatProperties.imageGranularity.height);
            AlignedDivide(
                sparseImageInfo.extent.depth,
                pSparseReqs[i].formatProperties.imageGranularity.depth);
        }
        size_t totalBlocks =
            lod0BlockSize.width *
            lod0BlockSize.height *
            lod0BlockSize.depth;

        VkDeviceSize lod0MemSize = totalBlocks * memoryRequirements.alignment;

        // Allocate memory for each array layer
        for (uint32_t a = 0; a < sparseImageInfo.arrayLayers; ++a)
        {
            // ... Allocate memory block

            VkSparseImageMemoryBind* pBind = &imageBinds[a];
            pBind->subresource.aspectMask = pSparseReqs[i].formatProperties. ↩
               aspectMask;
            pBind->subresource.mipLevel = 0;
            pBind->subresource.arrayLayer = a;

            pBind->offset = (VkOffset3D){0, 0, 0};
            pBind->extent = sparseImageInfo.extent;
            pBind->memoryOffset = /* allocated memoryOffset */;
            pBind->memory = /* allocated memory */;
            pBind->flags = 0;
        }
    }
}

const VkSparseImageOpaqueMemoryBindInfo opaqueBindInfo =
{
    sparseImage,                                // image
    bindCount,                                  // bindCount
    binds                                       // pBinds
};

const VkSparseImageMemoryBindInfo imageBindInfo =
{
    sparseImage,                                // image
    sparseImageInfo.arrayLayers,                // bindCount
    imageBinds                                  // pBinds
};

const VkBindSparseInfo bindSparseInfo =
{
    VK_STRUCTURE_TYPE_BIND_SPARSE_INFO,         // sType
    NULL,                                       // pNext
    ...
    1,                                          // imageOpaqueBindCount
    &opaqueBindInfo,                            // pImageOpaqueBinds
    1,                                          // imageBindCount
    &imageBindInfo,                             // pImageBinds
    ...
};

// vkQueueBindSparse is application synchronized per queue object.
AcquireQueueOwnership(queue);

// Actually bind memory
vkQueueBindSparse(queue, 1, &bindSparseInfo, VK_NULL_HANDLE);

ReleaseQueueOwnership(queue);
```

Chapter 28

Extended Functionality

Additional functionality may be provided by layers or extensions. A layer cannot add or modify Vulkan commands, while an extension may do so.

There are two kinds of layers and extensions, global and device. Global layers and extensions are general purpose and do not depend on a specific device. Device layers and extensions operate on specific devices, and require a valid VkDevice to be used. Global extensions usually affect the operation of the API as a whole, whereas device layers and extensions tend to be hardware-specific. Examples of these might be:

- Whole API validation is an example of a good global layer.
- Debug capabilities might make a good global extension.
- A layer that provides hardware-specific performance telemetry and analysis could be a device layer.
- Functions to allow an application to use additional hardware features beyond the core would be a good candidate for a device extension.

28.1 Layers

When a layer is enabled, it inserts itself into the call chain for Vulkan commands the layer is interested in. A common use of layers is to validate application behavior during development. For example, the implementation will not check that Vulkan enums used by the application fall within allowed ranges. Instead, a validation layer would do those checks and flag issues. This avoids a performance penalty during production use of the application because those layers would not be enabled in production.

To query the available global layers, call:

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

- *pPropertyCount* is the number of layer properties that can be returned in *pProperties*.
- *pProperties* is an array of properties of global layers available on this implementation.

Valid Usage

- *pPropertyCount* must be a pointer to a uint32_t value
- If the value referenced by *pPropertyCount* is not `0`, and *pProperties* is not `NULL`, *pProperties* must be a pointer to an array of *pPropertyCount* VkLayerProperties structures

To enable a global layer, the name of the layer should be added to the *ppEnabledLayerNames* member of `VkInstanceCreateInfo` when creating a `VkInstance`.

To query the layers available to a given physical device, call:

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

- *physicalDevice* is the physical device that will be queried.
- *pPropertyCount* is the number of layer properties that can be returned in *pProperties*.
- *pProperties* is an array of properties of layers available on *physicalDevice*

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *pPropertyCount* must be a pointer to a uint32_t value
- If *pProperties* is not `NULL`, *pProperties* must be a pointer to an array of *pPropertyCount* VkLayerProperties structures
- If *pProperties* is not `NULL`, the value referenced by *pPropertyCount* must be greater than `0`

To enable a device layer, the name of the layer should be added to the *ppEnabledLayerNames* member of `VkDeviceCreateInfo` when creating a `VkDevice`.

Both commands will return an array of VkLayerProperties of the respective global or device layers present. Calling **`vkEnumerateInstanceLayerProperties`** or **`vkEnumerateDeviceLayerProperties`** with *pProperties* set to `NULL` will return the number of supported layers in the uint32_t variable pointed to by *pPropertyCount*. If *pProperties* is not set to `NULL`, the query will fill the array and update *pPropertyCount* to indicate the number of VkLayerProperties filled in. If *pPropertyCount* is smaller than the number of layers available, VK_INCOMPLETE will be returned instead of VK_SUCCESS to indicate that not all the available properties were returned.

The definition of VkLayerProperties is:

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

- *layerName* is a null-terminated string specifying the name of the layer. Use this name in the *ppEnabledLayerNames* array in the VkInstanceCreateInfo given to **vkCreateInstance** to enable this layer in the instance.
- *apiVersion* is the Vulkan version the layer was written to, encoded as described in the API Version Numbers and Semantics section.
- *implementationVersion* is the version of this layer. It is an integer, increasing with backward compatible changes.
- *description* is a null-terminated string providing additional details that can be used by the application to identify the layer.

28.2 Extensions

Extensions may define new Vulkan commands, structures, and enumerants. For compilation purposes, the interfaces defined by registered extensions, including new structures and enumerants as well as function pointer types for new commands, are defined in the Khronos-supplied **vulkan**.h together with the core API. However, commands defined by extensions may not be available for static linking - in which case function pointers to these commands should be queried at runtime as described in Section 3.1. Extensions may be provided by layers as well as by a Vulkan implementation.

To query the available global extensions, call:

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

- *pLayerName* is either NULL, or the name of a global layer to retrieve extensions from.
- *pPropertyCount* is the number of extension properties that can be returned in *pProperties*.
- *pProperties* is an array of properties of global extensions available on this implementation.

Valid Usage

- If *pLayerName* is not NULL, *pLayerName* must be a null-terminated string

- *pPropertyCount* must be a pointer to a uint32_t value
- If the value referenced by *pPropertyCount* is not `0`, and *pProperties* is not `NULL`, *pProperties* must be a pointer to an array of *pPropertyCount* VkExtensionProperties structures
- If *pLayerName* is not `NULL`, it must be the name of a layer available on the system

To enable a global extension, the name of the extension should be added to the *ppEnabledExtensionNames* member of `VkInstanceCreateInfo` when creating a `VkInstance`.

To query the extensions available to a given physical device, call:

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

- *physicalDevice* is the physical device that will be queried.
- *pLayerName* is either NULL, or the name of a device or global layer to retrieve extensions from.
- *pPropertyCount* is the number of extension properties that can be returned in *pProperties*.
- *pProperties* is an array of properties of extensions available on *physicalDevice*.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- If *pLayerName* is not `NULL`, *pLayerName* must be a null-terminated string
- *pPropertyCount* must be a pointer to a uint32_t value
- If the value referenced by *pPropertyCount* is not `0`, and *pProperties* is not `NULL`, *pProperties* must be a pointer to an array of *pPropertyCount* VkExtensionProperties structures
- If *pLayerName* is not `NULL`, it must be the name of a layer available on the system

To enable a device layer, the name of the layer should be added to the *ppEnabledExtensionNames* member of `VkDeviceCreateInfo` when creating a `VkDevice`.

Both commands return an array of VkExtensionProperties for any extensions implemented by the given *pLayerName*. Set *pLayerName* to `NULL` to query for extensions not part of any layer. Calling **`vkEnumerateInstanceExtensionProperties`** or **`vkEnumerateDeviceExtensionProperties`** with *pProperties* set to `NULL` will return the count of extensions for the given layer in the uint32_t variable pointed to by *pPropertyCount*. With *pProperties* pointing to an array of VkExtensionProperties,

`vkEnumerateInstanceExtensionProperties` and **`vkEnumerateDeviceExtensionProperties`** will fill the array and update the count to indicate the number of VkExtensionProperties filled in. If the provided count is smaller than the number of extensions available, VK_INCOMPLETE will be returned instead of VK_SUCCESS to indicate that not all the available properties were returned.

The definition of VkExtensionProperties is:

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

- *`extensionName`* is a null-terminated string specifying the name of the extension.
- *`specVersion`* is the version of this extension. It is an integer, incremented with backward compatible changes.

Chapter 29

Features, Limits, and Formats

Vulkan is designed to support a wide range of hardware and as such there are a number of features, limits, and formats which are not supported on all hardware. Features describe functionality that is not required and which must be explicitly enabled. Limits describe implementation-dependent minimums, maximums, and other device characteristics that an application may need to be aware of. Supported buffer and image formats may vary across implementations. A minimum set of format features are guaranteed, but others must be explicitly queried before use to ensure they are supported by the implementation.

29.1 Features

The Specification defines a set of fine-grained features that are not required, but may be supported by a Vulkan implementation. Support for features is reported and enabled on a per-feature basis. Features are properties of the physical device.

To query supported features, call:

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

physicalDevice is the physical device from which to query the supported features. *pFeatures* is a pointer to a VkPhysicalDeviceFeatures structure to be filled in. For each feature, a value of VK_TRUE indicates that a feature is supported on this physical device, and VK_FALSE indicates that the feature is not supported on this physical device.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *pFeatures* must be a pointer to a VkPhysicalDeviceFeatures structure

Fine-grained features used by a logical device must be enabled at VkDevice creation time. If a feature is enabled that the physical device does not support, VkDevice creation will fail. If an application uses a feature without enabling it at VkDevice creation time, the device behaviour is undefined. The validation layer will warn if features are used without being enabled.

The fine-grained features are enabled by passing a pointer to the VkPhysicalDeviceFeatures structure via the *pEnabledFeatures* member of the VkDeviceCreateInfo structure that is passed into the **vkCreateDevice** call. If a member of *pEnabledFeatures* is set to VK_TRUE or VK_FALSE, then the device will be created with the indicated feature enabled or disabled, respectively.

If an application wishes to enable all features supported by a device, it can simply pass in the VkPhysicalDeviceFeatures structure that was previously returned by **vkGetPhysicalDeviceFeatures**. To disable an individual feature, the application can set the desired member to VK_FALSE in the same structure. To disable all features which are not required, set *pEnabledFeatures* to `NULL`.

Note
Some features, such as *robustBufferAccess*, may incur a run-time performance cost. Application writers should carefully consider the implications of enabling all supported features.

The definition of VkPhysicalDeviceFeatures is:

```
typedef struct VkPhysicalDeviceFeatures {
    VkBool32                                    robustBufferAccess;
    VkBool32                                    fullDrawIndexUint32;
    VkBool32                                    imageCubeArray;
    VkBool32                                    independentBlend;
    VkBool32                                    geometryShader;
    VkBool32                                    tessellationShader;
    VkBool32                                    sampleRateShading;
    VkBool32                                    dualSrcBlend;
    VkBool32                                    logicOp;
    VkBool32                                    multiDrawIndirect;
    VkBool32                                    drawIndirectFirstInstance;
    VkBool32                                    depthClamp;
    VkBool32                                    depthBiasClamp;
    VkBool32                                    fillModeNonSolid;
    VkBool32                                    depthBounds;
    VkBool32                                    wideLines;
    VkBool32                                    largePoints;
    VkBool32                                    alphaToOne;
    VkBool32                                    multiViewport;
    VkBool32                                    samplerAnisotropy;
    VkBool32                                    textureCompressionETC2;
    VkBool32                                    textureCompressionASTC_LDR;
    VkBool32                                    textureCompressionBC;
    VkBool32                                    occlusionQueryPrecise;
    VkBool32                                    pipelineStatisticsQuery;
    VkBool32                                    vertexPipelineStoresAndAtomics;
    VkBool32                                    fragmentStoresAndAtomics;
    VkBool32                                     ↩
        shaderTessellationAndGeometryPointSize;
    VkBool32                                    shaderImageGatherExtended;
    VkBool32                                    shaderStorageImageExtendedFormats;
    VkBool32                                    shaderStorageImageMultisample;
    VkBool32                                    shaderStorageImageReadWithoutFormat;
    VkBool32                                    shaderStorageImageWriteWithoutFormat ↩
        ;
    VkBool32                                    ←
       shaderUniformBufferArrayDynamicIndexing;
    VkBool32                                    ←
       shaderSampledImageArrayDynamicIndexing;
    VkBool32                                    ←
       shaderStorageBufferArrayDynamicIndexing;
    VkBool32                                    ←
       shaderStorageImageArrayDynamicIndexing;
    VkBool32                                    shaderClipDistance;
    VkBool32                                    shaderCullDistance;
    VkBool32                                    shaderFloat64;
    VkBool32                                    shaderInt64;
    VkBool32                                    shaderInt16;
    VkBool32                                    shaderResourceResidency;
    VkBool32                                    shaderResourceMinLod;
    VkBool32                                    sparseBinding;
    VkBool32                                    sparseResidencyBuffer;
    VkBool32                                    sparseResidencyImage2D;
    VkBool32                                    sparseResidencyImage3D;
    VkBool32                                    sparseResidency2Samples;
    VkBool32                                    sparseResidency4Samples;
    VkBool32                                    sparseResidency8Samples;
    VkBool32                                    sparseResidency16Samples;
    VkBool32                                    sparseResidencyAliased;
    VkBool32                                    variableMultisampleRate;
    VkBool32                                    inheritedQueries;
} VkPhysicalDeviceFeatures;
```

The members of the VkPhysicalDeviceFeatures structure describe the following features:

- *robustBufferAccess* indicates that out of bounds accesses to buffers via shader operations are well-defined.
  - When enabled, out-of-bounds buffer reads will return any of the following values:
    - Values from anywhere within the buffer object.
    - Zero values, or (0,0,0,x) vectors for vector reads where x is a valid value represented in the type of the vector components and may be any of:
      - 0, 1, or the maximum representable positive integer value, for signed or unsigned integer components
      - 0.0 or 1.0, for floating-point components
  - When enabled, out-of-bounds writes may modify values within the buffer object or be ignored.
  - If not enabled, out of bounds accesses may cause undefined behaviour up-to and including process termination.
- *fullDrawIndexUint32* indicates the full 32-bit range of indices is supported for indexed draw calls when using a `VkIndexType` of VK_INDEX_TYPE_UINT32. *maxDrawIndexedIndexValue* is the maximum index value that may be used (aside from the primitive restart index, which is always 2^{32}-1 when the `VkIndexType` is VK_INDEX_TYPE_UINT32). If this feature is supported, *maxDrawIndexedIndexValue* must be 2^{32}-1; otherwise it must be no smaller than 2^{24}-1. See maxDrawIndexedIndexValue.
- *imageCubeArray* indicates whether image views with a `VkImageViewType` of VK_IMAGE_VIEW_TYPE_CUBE_ARRAY can be created, and that the corresponding **SampledCubeArray** and **ImageCubeArray** SPIR-V capabilities can be used in shader code.
- *independentBlend* indicates whether the VkPipelineColorBlendAttachmentState settings are controlled independently per-attachment. If this feature is not enabled, the VkPipelineColorBlendAttachmentState settings for all color attachments must be identical. Otherwise, a different VkPipelineColorBlendAttachmentState can be provided for each bound color attachment.

- *geometryShader* indicates whether geometry shaders are supported. If this feature is not enabled, the VK_SHADER_STAGE_GEOMETRY_BIT and VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT enum values must not be used. This also indicates whether the **Geometry** SPIR-V OpCapability can be used in shader code.
- *tessellationShader* indicates whether tessellation control and evaluation shaders are supported. If this feature is not enabled, the VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT, VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT, VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT, VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT, and VK_STRUCTURE_TYPE_PIPELINE_TESSELLATION_STATE_CREATE_INFO enum values must not be used. This also indicates whether the **Tessellation** SPIR-V OpCapability can be used in shader code.
- *sampleRateShading* indicates whether per-sample shading and multisample interpolation are supported. If this feature is not enabled, the *sampleShadingEnable* member of the VkPipelineMultisampleStateCreateInfo structure must be set to VK_FALSE and the *minSampleShading* member is ignored. This also indicates whether the **SampleRateShading** SPIR-V OpCapability can be used in shader code.
- *dualSrcBlend* indicates whether blend operations which take two sources are supported. If this feature is not enabled, the VK_BLEND_FACTOR_SRC1_COLOR, VK_BLEND_FACTOR_ONE_MINUS_SRC1_COLOR, VK_BLEND_FACTOR_SRC1_ALPHA, and VK_BLEND_FACTOR_ONE_MINUS_SRC1_ALPHA enum values must not be used as source or destination blending factors. See Section 25.1.2.
- *logicOp* indicates whether logic operations are supported. If this feature is not enabled, the *logicOpEnable* member of the VkPipelineColorBlendStateCreateInfo structure must be set to VK_FALSE, and the *logicOp* member is ignored.
- *multiDrawIndirect* indicates whether multi-draw indirect is supported. If this feature is not enabled, the *drawCount* parameter to the **vkCmdDrawIndirect** and **vkCmdDrawIndexedIndirect** commands must be 1. The *maxDrawIndirectCount* member of the VkPhysicalDeviceLimits structure must also be 1 if this feature is not supported. See maxDrawIndirectCount.
- *drawIndirectFirstInstance* indicates whether indirect draw calls support the *firstInstance* parameter. If this feature is not enabled, the *firstInstance* member of all VkDrawIndirectCommand and VkDrawIndexedIndirectCommand structures that are provided to the **vkCmdDrawIndirect** and **vkCmdDrawIndexedIndirect** commands must be 0.
- *depthClamp* indicates whether depth clamping is supported. If this feature is not enabled, the *depthClampEnable* member of the VkPipelineRasterizationStateCreateInfo structure must be set to VK_FALSE. Otherwise, setting *depthClampEnable* to VK_TRUE will enable depth clamping.
- *depthBiasClamp* indicates whether depth bias clamping is supported. If this feature is not enabled, the *depthBiasClamp* member of the VkPipelineRasterizationStateCreateInfo structure must be set to 0.0.
- *fillModeNonSolid* indicates whether point and wireframe fill modes are supported. If this feature is not enabled, the VK_POLYGON_MODE_POINT and VK_POLYGON_MODE_LINE enum values must not be used.
- *depthBounds* indicates whether depth bounds tests are supported. If this feature is not enabled, the *depthBoundsTestEnable* member of the VkPipelineDepthStencilStateCreateInfo structure must be set to VK_FALSE. When *depthBoundsTestEnable* is set to VK_FALSE, the values of the *minDepthBounds* and *maxDepthBounds* members of the VkPipelineDepthStencilStateCreateInfo structure are ignored.
- *wideLines* indicates whether lines with width other than 1.0 are supported. If this feature is not enabled, the *lineWidth* member of the VkPipelineRasterizationStateCreateInfo structure must be set to 1.0. When this feature is supported, the range and granularity of supported line widths are indicated by the *lineWidthRange* and *lineWidthGranularity* members of the VkPhysicalDeviceLimits structure, respectively.

- *largePoints* indicates whether points with size greater than 1.0 are supported. If this feature is not enabled, only a point size of 1.0 written by a shader is supported. The range and granularity of supported point sizes are indicated by the *pointSizeRange* and *pointSizeGranularity* members of the VkPhysicalDeviceLimits structure, respectively.
- *alphaToOne* indicates whether the implementation is able to replace the alpha value of the color fragment output from the fragment shader with the maximum representable alpha value for fixed-point colors or 1.0 for floating-point colors. If this feature is not enabled, then the *alphaToOneEnable* member of the VkPipelineMultisampleStateCreateInfo structure must be set to VK_FALSE. Otherwise setting *alphaToOneEnable* to VK_TRUE will enable alpha-to-one behaviour.
- *multiViewport* indicates whether more than one viewport is supported. If this feature is not enabled, the *viewportCount* and *scissorCount* members of the VkPipelineViewportStateCreateInfo structure must be set to 1. Similarly, the *viewportCount* parameter to the **vkCmdSetViewport** command and the *scissorCount* parameter to the **vkCmdSetScissor** command must be 1, and the *firstViewport* parameter to the **vkCmdSetViewport** command and the *firstScissor* parameter to the **vkCmdSetScissor** command must be 0.
- *samplerAnisotropy* indicates whether anisotropic filtering is supported. If this feature is not enabled, the *maxAnisotropy* member of the VkSamplerCreateInfo structure must be 1.0.
- *textureCompressionETC2* indicates whether the ETC2 and EAC compressed texture formats are supported. If this feature is not enabled, the following formats must not be used to create images:
  - VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK
  - VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK
  - VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK
  - VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK
  - VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK
  - VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK
  - VK_FORMAT_EAC_R11_UNORM_BLOCK
  - VK_FORMAT_EAC_R11_SNORM_BLOCK
  - VK_FORMAT_EAC_R11G11_UNORM_BLOCK
  - VK_FORMAT_EAC_R11G11_SNORM_BLOCK

  `vkGetPhysicalDeviceFormatProperties` is used to check for the supported properties of individual formats.
- *textureCompressionASTC_LDR* indicates whether the ASTC LDR compressed texture formats are supported. If this feature is not enabled, the following formats must not be used to create images:
  - VK_FORMAT_ASTC_4x4_UNORM_BLOCK
  - VK_FORMAT_ASTC_4x4_SRGB_BLOCK
  - VK_FORMAT_ASTC_5x4_UNORM_BLOCK
  - VK_FORMAT_ASTC_5x4_SRGB_BLOCK
  - VK_FORMAT_ASTC_5x5_UNORM_BLOCK
  - VK_FORMAT_ASTC_5x5_SRGB_BLOCK
  - VK_FORMAT_ASTC_6x5_UNORM_BLOCK
  - VK_FORMAT_ASTC_6x5_SRGB_BLOCK

- VK_FORMAT_ASTC_6x6_UNORM_BLOCK
- VK_FORMAT_ASTC_6x6_SRGB_BLOCK
- VK_FORMAT_ASTC_8x5_UNORM_BLOCK
- VK_FORMAT_ASTC_8x5_SRGB_BLOCK
- VK_FORMAT_ASTC_8x6_UNORM_BLOCK
- VK_FORMAT_ASTC_8x6_SRGB_BLOCK
- VK_FORMAT_ASTC_8x8_UNORM_BLOCK
- VK_FORMAT_ASTC_8x8_SRGB_BLOCK
- VK_FORMAT_ASTC_10x5_UNORM_BLOCK
- VK_FORMAT_ASTC_10x5_SRGB_BLOCK
- VK_FORMAT_ASTC_10x6_UNORM_BLOCK
- VK_FORMAT_ASTC_10x6_SRGB_BLOCK
- VK_FORMAT_ASTC_10x8_UNORM_BLOCK
- VK_FORMAT_ASTC_10x8_SRGB_BLOCK
- VK_FORMAT_ASTC_10x10_UNORM_BLOCK
- VK_FORMAT_ASTC_10x10_SRGB_BLOCK
- VK_FORMAT_ASTC_12x10_UNORM_BLOCK
- VK_FORMAT_ASTC_12x10_SRGB_BLOCK
- VK_FORMAT_ASTC_12x12_UNORM_BLOCK
- VK_FORMAT_ASTC_12x12_SRGB_BLOCK

`vkGetPhysicalDeviceFormatProperties` is used to check for the supported properties of individual formats.

• *`textureCompressionBC`* indicates whether the BC compressed texture formats are supported. If this feature is not enabled, the following formats must not be used to create images:

- VK_FORMAT_BC1_RGB_UNORM_BLOCK
- VK_FORMAT_BC1_RGB_SRGB_BLOCK
- VK_FORMAT_BC1_RGBA_UNORM_BLOCK
- VK_FORMAT_BC1_RGBA_SRGB_BLOCK
- VK_FORMAT_BC2_UNORM_BLOCK
- VK_FORMAT_BC2_SRGB_BLOCK
- VK_FORMAT_BC3_UNORM_BLOCK
- VK_FORMAT_BC3_SRGB_BLOCK
- VK_FORMAT_BC4_UNORM_BLOCK
- VK_FORMAT_BC4_SNORM_BLOCK
- VK_FORMAT_BC5_UNORM_BLOCK
- VK_FORMAT_BC5_SNORM_BLOCK
- VK_FORMAT_BC6H_UFLOAT_BLOCK
- VK_FORMAT_BC6H_SFLOAT_BLOCK
- VK_FORMAT_BC7_UNORM_BLOCK

  - VK_FORMAT_BC7_SRGB_BLOCK

  `vkGetPhysicalDeviceFormatProperties` is used to check for the supported properties of individual formats.

- *`occlusionQueryPrecise`* indicates whether precise (non-conservative) occlusion queries are supported. Occlusion queries are created in a VkQueryPool by specifying the *`queryType`* of VK_QUERY_TYPE_OCCLUSION in the VkQueryPoolCreateInfo structure which is passed to **`vkCreateQueryPool`**. If this feature is enabled, queries of this type can enable VK_QUERY_CONTROL_PRECISE_BIT in the *`flags`* parameter to **`vkCmdBeginQuery`**. If this feature is not supported, the implementation supports only conservative occlusion queries. When any samples are passed, conservative queries will return between one and the actual number of samples passed. When this feature is enabled and VK_QUERY_CONTROL_PRECISE_BIT is set, occlusion queries will report the actual number of samples passed.
- *`pipelineStatisticsQuery`* indicates whether the pipeline statistics queries are supported. If this feature is not enabled, queries of type VK_QUERY_TYPE_PIPELINE_STATISTICS cannot be created, and none of the `VkQueryPipelineStatisticFlagBits` bits can be set in the *`pipelineStatistics`* member of the VkQueryPoolCreateInfo structure.
- *`vertexPipelineStoresAndAtomics`* indicates whether storage buffers and images support stores and atomic operations in the vertex, tessellation, and geometry shader stages. If this feature is not enabled, all storage image, storage texel buffers and storage buffer variables in shaders for these stages must be decorated with the **NonWriteable** SPIR-V decoration (or the **`readonly`** memory qualifier in GLSL).
- *`fragmentStoresAndAtomics`* indicates whether storage buffers and images support stores and atomic operations in the fragment shader stage. If this feature is not enabled, all storage image, storage texel buffers and storage buffer variables in shaders for the fragment stage must be decorated with the **NonWriteable** SPIR-V decoration (or the **`readonly`** memory qualifier in GLSL).
- *`shaderTessellationAndGeometryPointSize`* indicates whether the **`PointSize`** shader builtin is available in the tessellation control, tessellation evaluation, and geometry shader stages. If this feature is not enabled, the **`PointSize`** shader builtin is not available in these shader stages and all points written from a tessellation or geometry shader will have a size of 1.0. This also indicates whether the **TessellationPointSize** SPIR-V OpCapability can be used in shader code for tessellation control and evaluation shaders, or if the **GeometryPointSize** SPIR-V OpCapability can be used in shader code for geometry shaders. An implementation supporting this feature must also support one or both of the *tessellationShader* or *geometryShader* features.
- *`shaderImageGatherExtended`* indicates whether the extended set of image gather instructions are available in shader code. If this feature is not enabled, the *textureGatherOffset* shader instruction only supports offsets that are constant integer expressions and the *textureGatherOffsets* shader instruction is not supported. This also indicates whether the **ImageGatherExtended** SPIR-V OpCapability can be used in shader code.
- *`shaderStorageImageExtendedFormats`* indicates whether the extended storage image formats are available in shader code. If this feature is not enabled, the formats requiring the **StorageImageExtendedFormats** SPIR-V OpCapability are not supported for resources referenced by the VK_DESCRIPTOR_TYPE_STORAGE_IMAGE descriptor type. This also indicates whether the **StorageImageExtendedFormats** OpCapability can be used in shader code.
- *`shaderStorageImageMultisample`* indicates whether multisampled storage images are supported. If this feature is not enabled, images that are created with a *`usage`* that includes VK_IMAGE_USAGE_STORAGE_BIT must be created with *`samples`* equal to VK_SAMPLE_COUNT_1_BIT. This also indicates whether the **StorageImageMultisample** SPIR-V OpCapability can be used in shader code.
- *`shaderStorageImageReadWithoutFormat`* indicates whether storage images require a format qualifier to be specified when reading from storage images. If this feature is not enabled, the OpImageRead SPIR-V instruction must not have an OpImageType of **Unknown**. This also indicates whether the **StorageImageReadWithoutFormat** SPIR-V OpCapability can be used in shader code.

- `shaderStorageImageWriteWithoutFormat` indicates whether storage images require a format qualifier to be specified when writing to storage images. If this feature is not enabled, the OpImageWrite SPIR-V instruction must not have an OpImageType of **Unknown**. This also indicates whether the **StorageImageWriteWithoutFormat** SPIR-V OpCapability can be used in shader code.
- `shaderUniformBufferArrayDynamicIndexing` indicates whether arrays of uniform buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_ DYNAMIC must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether the **UniformBufferArrayDynamicIndexing** SPIR-V OpCapability can be used in shader code.
- `shaderSampledImageArrayDynamicIndexing` indicates whether arrays of samplers or sampled images can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_SAMPLER, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_ SAMPLER and VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether the **SampledImageArrayDynamicIndexing** SPIR-V OpCapability can be used in shader code.
- `shaderStorageBufferArrayDynamicIndexing` indicates whether arrays of storage buffers can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_ DYNAMIC must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether the **StorageBufferArrayDynamicIndexing** SPIR-V OpCapability can be used in shader code.
- `shaderStorageImageArrayDynamicIndexing` indicates whether arrays of storage images can be indexed by dynamically uniform integer expressions in shader code. If this feature is not enabled, resources with a descriptor type of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE must be indexed only by constant integral expressions when aggregated into arrays in shader code. This also indicates whether the **StorageImageArrayDynamicIndexing** SPIR-V OpCapability can be used in shader code.
- `shaderClipDistance` indicates whether clip distances are supported in shader code. If this feature is not enabled, the **`ClipDistance`** shader builtin is not available in the builtin shader input and output blocks. This also indicates whether the **ClipDistance** SPIR-V OpCapability can be used in shader code.
- `shaderCullDistance` indicates whether cull distances are suppored in shader code. If this feature is not enabled, the **`CullDistance`** shader builtin is not available in the builtin shader input and output blocks. This also indicates whether the **CullDistance** SPIR-V OpCapability can be used in shader code.
- `shaderFloat64` indicates whether 64-bit floats (doubles) are supported in shader code. If this feature is not enabled, 64-bit floating-point types must not be used in shader code. This also indicates whether the **Float64** SPIR-V OpCapability can be used in shader code.
- `shaderInt64` indicates whether 64-bit integers (signed and unsigned) are supported in shader code. If this feature is not enabled, 64-bit integer types must not be used in shader code. This also indicates whether the **Int64** SPIR-V OpCapability can be used in shader code.
- `shaderInt16` indicates whether 16-bit integers (signed and unsigned) are supported in shader code. If this feature is not enabled, 16-bit integer types must not be used in shader code. This also indicates whether the **Int16** SPIR-V OpCapability can be used in shader code.
- `shaderResourceResidency` indicates whether image operations that return resource residency information are supported in shader code. If this feature is not enabled, image operations which return resource residency information must not be used in shader code. This also indicates whether the **SparseResidency** SPIR-V OpCapability can be used in shader code. The feature requires at least one of the `sparseResidency*` features to be supported.

- *shaderResourceMinLod* indicates whether image operations that specify the minimum resource level-of-detail (LOD) are supported in shader code. If this feature is not enabled, the image operations which specify minimum resource LOD must not be used in shader code. This also indicates whether the **MinLod** SPIR-V OpCapability can be used in shader code.
- *sparseBinding* indicates whether resource memory can be managed at opaque page level instead of at the object level. If this feature is not enabled, resource memory must be bound only on a per-object basis using the **vkBindBufferMemory** and **vkBindImageMemory** commands. In this case, buffers and images must not be created with VK_BUFFER_CREATE_SPARSE_BINDING_BIT and VK_IMAGE_CREATE_SPARSE_BINDING_BIT set in the *flags* member of the VkBufferCreateInfo and VkImageCreateInfo structures, respectively. Otherwise resource memory can be managed as described in Sparse Resource Features.
- *sparseResidencyBuffer* indicates whether the device can access partially resident buffers. If this feature is not enabled, buffers must not be created with VK_BUFFER_CREATE_SPARSE_RESIDENCY_BIT set in the *flags* member of the VkBufferCreateInfo structure.
- *sparseResidencyImage2D* indicates whether the device can access partially resident 2D images with 1 sample per pixel. If this feature is not enabled, images with an *imageType* of VK_IMAGE_TYPE_2D and *samples* set to VK_SAMPLE_COUNT_1_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the *flags* member of the VkImageCreateInfo structure.
- *sparseResidencyImage3D* indicates whether the device can access partially resident 3D images. If this feature is not enabled, images with an *imageType* of VK_IMAGE_TYPE_3D must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the *flags* member of the VkImageCreateInfo structure.
- *sparseResidency2Samples* indicates whether the physical device can access partially resident 2D images with 2 samples per pixel. If this feature is not enabled, images with an *imageType* of VK_IMAGE_TYPE_2D and *samples* set to VK_SAMPLE_COUNT_2_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the *flags* member of the VkImageCreateInfo structure.
- *sparseResidency4Samples* indicates whether the physical device can access partially resident 2D images with 4 samples per pixel. If this feature is not enabled, images with an *imageType* of VK_IMAGE_TYPE_2D and *samples* set to VK_SAMPLE_COUNT_4_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the *flags* member of the VkImageCreateInfo structure.
- *sparseResidency8Samples* indicates whether the physical device can access partially resident 2D images with 8 samples per pixel. If this feature is not enabled, images with an *imageType* of VK_IMAGE_TYPE_2D and *samples* set to VK_SAMPLE_COUNT_8_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the *flags* member of the VkImageCreateInfo structure.
- *sparseResidency16Samples* indicates whether the physical device can access partially resident 2D images with 16 samples per pixel. If this feature is not enabled, images with an *imageType* of VK_IMAGE_TYPE_2D and *samples* set to VK_SAMPLE_COUNT_16_BIT must not be created with VK_IMAGE_CREATE_SPARSE_RESIDENCY_BIT set in the *flags* member of the VkImageCreateInfo structure.
- *sparseResidencyAliased* indicates whether the physical device can correctly access data aliased into multiple locations. If this feature is not enabled, the VK_BUFFER_CREATE_SPARSE_ALIASED_BIT and VK_IMAGE_CREATE_SPARSE_ALIASED_BIT enum values must not be used in *flags* members of the VkBufferCreateInfo and VkImageCreateInfo structures, respectively.
- *variableMultisampleRate* indicates whether all pipelines that will be bound to a command buffer during a subpass with no attachments must have the same value for VkPipelineMultisampleStateCreateInfo::*rasterizationSamples*. If set to VK_TRUE, the implementation supports variable multisample rates in a subpass with no attachments. If set to VK_FALSE, then all pipelines for such a subpass must have the same multisample rate. This has no effect in situations where a subpass has valid attachment references.

• *inheritedQueries* indicates whether a secondary command buffer may be executed while a query is active.

Valid Usage

• If the value of any member of this structure is VK_FALSE, as returned by `vkGetPhysicalDeviceFeatures`, then it must be VK_FALSE when passed as part of the VkDeviceCreateInfo struct when creating a device

29.1.1 Feature Requirements

All Vulkan graphics implementations must support the following features:

• robustBufferAccess.

All other features are not required by the Specification.

29.2 Limits

There are a variety of implementation-dependent limits.

The VkPhysicalDeviceLimits are properties of the physical device. These are available in the *limits* member of the `VkPhysicalDeviceProperties` structure which is returned from `vkGetPhysicalDeviceProperties`.

The definition of VkPhysicalDeviceLimits is:

```
typedef struct VkPhysicalDeviceLimits {
    uint32_t                                    maxImageDimension1D;
    uint32_t                                    maxImageDimension2D;
    uint32_t                                    maxImageDimension3D;
    uint32_t                                    maxImageDimensionCube;
    uint32_t                                    maxImageArrayLayers;
    uint32_t                                    maxTexelBufferElements;
    uint32_t                                    maxUniformBufferRange;
    uint32_t                                    maxStorageBufferRange;
    uint32_t                                    maxPushConstantsSize;
    uint32_t                                    maxMemoryAllocationCount;
    uint32_t                                    maxSamplerAllocationCount;
    VkDeviceSize                                bufferImageGranularity;
    VkDeviceSize                                sparseAddressSpaceSize;
    uint32_t                                    maxBoundDescriptorSets;
    uint32_t                                    maxPerStageDescriptorSamplers;
    uint32_t                                    maxPerStageDescriptorUniformBuffers;
    uint32_t                                    maxPerStageDescriptorStorageBuffers;
    uint32_t                                    maxPerStageDescriptorSampledImages;
    uint32_t                                    maxPerStageDescriptorStorageImages;
    uint32_t                                     ↩
        maxPerStageDescriptorInputAttachments;
    uint32_t                                    maxPerStageResources;
    uint32_t                                    maxDescriptorSetSamplers;
    uint32_t                                    maxDescriptorSetUniformBuffers;
    uint32_t                                     ↩
        maxDescriptorSetUniformBuffersDynamic;
    uint32_t                                    maxDescriptorSetStorageBuffers;
    uint32_t                                     ↩
        maxDescriptorSetStorageBuffersDynamic;
    uint32_t                                    maxDescriptorSetSampledImages;
    uint32_t                                    maxDescriptorSetStorageImages;
    uint32_t                                    maxDescriptorSetInputAttachments;
    uint32_t                                    maxVertexInputAttributes;
    uint32_t                                    maxVertexInputBindings;
    uint32_t                                    maxVertexInputAttributeOffset;
    uint32_t                                    maxVertexInputBindingStride;
    uint32_t                                    maxVertexOutputComponents;
    uint32_t                                    maxTessellationGenerationLevel;
    uint32_t                                    maxTessellationPatchSize;
    uint32_t                                     ↩
        maxTessellationControlPerVertexInputComponents;
    uint32_t                                     ↩
        maxTessellationControlPerVertexOutputComponents;
    uint32_t                                     ↩
        maxTessellationControlPerPatchOutputComponents;
    uint32_t                                     ↩
        maxTessellationControlTotalOutputComponents;
    uint32_t                                     ↩
        maxTessellationEvaluationInputComponents;
    uint32_t                                     ↩
        maxTessellationEvaluationOutputComponents;
    uint32_t                                    maxGeometryShaderInvocations;
    uint32_t                                    maxGeometryInputComponents;
    uint32_t                                    maxGeometryOutputComponents;
    uint32_t                                    maxGeometryOutputVertices;
    uint32_t                                    maxGeometryTotalOutputComponents;
    uint32_t                                    maxFragmentInputComponents;
    uint32_t                                    maxFragmentOutputAttachments;
    uint32_t                                    maxFragmentDualSrcAttachments;
    uint32_t                                    maxFragmentCombinedOutputResources;
    uint32_t                                    maxComputeSharedMemorySize;
    uint32_t                                    maxComputeWorkGroupCount[3];
    uint32_t                                    maxComputeWorkGroupInvocations;
    uint32_t                                    maxComputeWorkGroupSize[3];
    uint32_t                                    subPixelPrecisionBits;
    uint32_t                                    subTexelPrecisionBits;
    uint32_t                                    mipmapPrecisionBits;
    uint32_t                                    maxDrawIndexedIndexValue;
    uint32_t                                    maxDrawIndirectCount;
    float                                       maxSamplerLodBias;
    float                                       maxSamplerAnisotropy;
    uint32_t                                    maxViewports;
    uint32_t                                    maxViewportDimensions[2];
    float                                       viewportBoundsRange[2];
    uint32_t                                    viewportSubPixelBits;
    size_t                                      minMemoryMapAlignment;
    VkDeviceSize                                minTexelBufferOffsetAlignment;
    VkDeviceSize                                minUniformBufferOffsetAlignment;
    VkDeviceSize                                minStorageBufferOffsetAlignment;
    int32_t                                     minTexelOffset;
    uint32_t                                    maxTexelOffset;
    int32_t                                     minTexelGatherOffset;
    uint32_t                                    maxTexelGatherOffset;
    float                                       minInterpolationOffset;
    float                                       maxInterpolationOffset;
    uint32_t                                    subPixelInterpolationOffsetBits;
    uint32_t                                    maxFramebufferWidth;
    uint32_t                                    maxFramebufferHeight;
    uint32_t                                    maxFramebufferLayers;
    VkSampleCountFlags                          framebufferColorSampleCounts;
    VkSampleCountFlags                          framebufferDepthSampleCounts;
    VkSampleCountFlags                          framebufferStencilSampleCounts;
    VkSampleCountFlags                          framebufferNoAttachmentsSampleCounts ↩
        ;
    uint32_t                                    maxColorAttachments;
    VkSampleCountFlags                          sampledImageColorSampleCounts;
    VkSampleCountFlags                          sampledImageIntegerSampleCounts;
    VkSampleCountFlags                          sampledImageDepthSampleCounts;
    VkSampleCountFlags                          sampledImageStencilSampleCounts;
    VkSampleCountFlags                          storageImageSampleCounts;
    uint32_t                                    maxSampleMaskWords;
    VkBool32                                    timestampComputeAndGraphics;
    float                                       timestampPeriod;
    uint32_t                                    maxClipDistances;
    uint32_t                                    maxCullDistances;
    uint32_t                                    maxCombinedClipAndCullDistances;
    uint32_t                                    discreteQueuePriorities;
    float                                       pointSizeRange[2];
    float                                       lineWidthRange[2];
    float                                       pointSizeGranularity;
    float                                       lineWidthGranularity;
    VkBool32                                    strictLines;
    VkBool32                                    standardSampleLocations;
    VkDeviceSize                                optimalBufferCopyOffsetAlignment;
    VkDeviceSize                                optimalBufferCopyRowPitchAlignment;
    VkDeviceSize                                nonCoherentAtomSize;
} VkPhysicalDeviceLimits;
```

The members of the VkPhysicalDeviceLimits describe the following properties of the physical device:

- *maxImageDimension1D* is the maximum dimension (*width*) of an image created with an *imageType* of VK_IMAGE_TYPE_1D.
- *maxImageDimension2D* is the maximum dimension (*width* or *height*) of an image created with an *imageType* of VK_IMAGE_TYPE_2D and without VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT set in flags.
- *maxImageDimension3D* is the maximum dimension (*width*, *height*, or *depth*) of an image created with an *imageType* of VK_IMAGE_TYPE_3D.
- *maxImageDimensionCube* is the maximum dimension (*width* or *height*) of an image created with an *imageType* of VK_IMAGE_TYPE_2D and with VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT set in *flags*.
- *maxImageArrayLayers* is the maximum number of layers (*arrayLayers*) for an image.

- *maxTexelBufferElements* is the maximum number of addressable texels for a buffer view created on a buffer which was created with the VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT or VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT set in the *usage* member of the VkBufferCreateInfo structure.

- *maxUniformBufferRange* is the maximum `VkDescriptorBufferInfo`::*range*, in bytes, that can be specified in a call to `vkUpdateDescriptorSets` when a `VkWriteDescriptorSet`::*descriptorType* of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC is specified.

- *maxStorageBufferRange* is the maximum `VkDescriptorBufferInfo`::*range*, in bytes, that can be specified in a call to `vkUpdateDescriptorSets` when a `VkWriteDescriptorSet`::*descriptorType* of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC is specified.

- *maxPushConstantsSize* is the maximum size, in bytes, of the push constants pool that can be referenced by the `vkCmdPushConstants` commands. For each of the push constant ranges indicated by the *pPushConstantRanges* member of the VkPipelineLayoutCreateInfo structure, *offset* + *size* must be less than or equal to this limit.

- *maxMemoryAllocationCount* is the maximum number of device memory allocations, as created by `vkAllocateMemory`, which can simultaneously exist.

- *maxSamplerAllocationCount* is the maximum number of sampler objects, as created by `vkCreateSampler`, which can simultaneously be allocated on a device.

- *bufferImageGranularity* is the granularity, in bytes, at which buffers and images can be bound to adjacent memory for simultaneous usage. See Buffer-Image Granularity for more details.

- *sparseAddressSpaceSize* is the total amount of address space available, in bytes, for sparse memory resources. This is an upper bound on the sum of the size of all sparse resources, regardless of whether any pages are bound.

- *maxBoundDescriptorSets* is the maximum number of descriptor sets that can be simultaneously used by a pipeline. Set numbers used by all shaders must be less than *maxBoundDescriptorSets*. See Section 13.2.

- *maxPerStageDescriptorSamplers* is the maximum number of samplers that can be referenced in a pipeline layout for any single shader stage. Descriptors with a type of VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER count against this limit. A descriptor is referenced by a pipeline shader stage when the *stageFlags* member of the VkDescriptorSetLayoutBinding structure has the bit for that shader stage set. See Section 13.1.2 and Section 13.1.4.

- *maxPerStageDescriptorUniformBuffers* is the maximum number of uniform buffers that can be referenced in a pipeline layout for any single shader stage. Descriptors with a type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC count against this limit. A descriptor is referenced by a pipeline shader stage when the *stageFlags* member of the VkDescriptorSetLayoutBinding structure has the bit for that shader stage set. See Section 13.1.7 and Section 13.1.9.

- *maxPerStageDescriptorStorageBuffers* is the maximum number of storage buffers that can be referenced in a pipeline layout for any single shader stage. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC count against this limit. A descriptor is referenced by a pipeline shader stage when the *stageFlags* member of the VkDescriptorSetLayoutBinding structure has the bit for that shader stage set. See Section 13.1.8 and Section 13.1.10.

- *maxPerStageDescriptorSampledImages* is the maximum number of sampled images that can be referenced in a pipeline layout for any single shader stage. Descriptors with a type of VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, or VK_DESCRIPTOR_TYPE_

UNIFORM_TEXEL_BUFFER count against this limit. A descriptor is referenced by a pipeline shader stage when the *stageFlags* member of the VkDescriptorSetLayoutBinding structure has the bit for that shader stage set. See Section 13.1.4, Section 13.1.3, and Section 13.1.5.

- *maxPerStageDescriptorStorageImages* is the maximum number of storage images that can be referenced in a pipeline layout for any single shader stage. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER count against this limit. A descriptor is referenced by a pipeline shader stage when the *stageFlags* member of the VkDescriptorSetLayoutBinding structure has the bit for that shader stage set. See Section 13.1.1, and Section 13.1.6.

- *maxPerStageDescriptorInputAttachments* is the maximum number of input attachments that can be referenced in a pipeline layout for any single shader stage. Descriptors with a type of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT count against this limit. A descriptor is referenced by a pipeline shader stage when the *stageFlags* member of the VkDescriptorSetLayoutBinding structure has the bit for that shader stage set. These are only supported for the fragment stage. See Section 13.1.11.

- *maxPerStageResources* is the maximum number of resources that can be referenced in a pipeline layout by a single stage. Descriptors with a type of VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER, VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC, VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC or enume:VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT count against this limit. For the fragment shader stage the framebuffer color attachments also count against this limit.

- *maxDescriptorSetSamplers* is the maximum number of samplers that can be referenced in a pipeline layout across all pipeline shader stages and descriptor sets. Descriptors with a type of VK_DESCRIPTOR_TYPE_SAMPLER or VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER count against this limit. See Section 13.1.2 and Section 13.1.4.

- *maxDescriptorSetUniformBuffers* is the maximum number of uniform buffers that can be referenced in a pipeline layout across all pipeline shader stages and descriptor sets. Descriptors with a type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC count against this limit. See Section 13.1.7 and Section 13.1.9.

- *maxDescriptorSetUniformBuffersDynamic* is the maximum number of dynamic uniform buffers that can be referenced in a pipeline layout across all pipeline shader stages and descriptor sets. Descriptors with a type of VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC count against this limit. See Section 13.1.9.

- *maxDescriptorSetStorageBuffers* is the maximum number of storage buffers that can be referenced in a pipeline layout across all pipeline shader stages and descriptor sets. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC count against this limit. See Section 13.1.8 and Section 13.1.10.

- *maxDescriptorSetStorageBuffersDynamic* is the maximum number of dynamic storage buffers that can be referenced in a pipeline layout across all pipeline shader stages and descriptor sets. Descriptors with a type of VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC count against this limit. See Section 13.1.10.

- *maxDescriptorSetSampledImages* is the maximum number of sampled images that can be referenced in a pipeline layout across all pipeline shader stages and descriptor sets. Descriptors with a type of VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, or VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER count against this limit. See Section 13.1.4, Section 13.1.3, and Section 13.1.5.

- *maxDescriptorSetStorageImages* is the maximum number of storage images that can be referenced in a pipeline layout across all pipeline shader stages and descriptor sets. Descriptors with a type of VK_

DESCRIPTOR_TYPE_STORAGE_IMAGE, or VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER count against this limit. See Section 13.1.1, and Section 13.1.6.

- *maxDescriptorSetInputAttachments* is the maximum number of input attachments that can be referenced in a pipeline layout across all pipeline shader stages and descriptor sets. Descriptors with a type of VK_DESCRIPTOR_TYPE_INPUT_ATTACHMENT count against this limit. See Section 13.1.11.
- *maxVertexInputAttributes* is the maximum number of vertex input attributes that can be specified for a graphics pipeline. These are described in the VkVertexInputAttributeDescription structure that is provided at graphics pipeline creation time via the *pVertexAttributeDescriptions* member of the VkPipelineVertexInputStateCreateInfo structure. See Section 19.1 and Section 19.2.
- *maxVertexInputBindings* is the maximum number of vertex buffers that can be specified for providing vertex attributes to a graphics pipeline. These are described in the VkVertexInputBindingDescription structure that is provided at graphics pipeline creation time via the *pVertexBindingDescriptions* member of the VkPipelineVertexInputStateCreateInfo structure. The *binding* member of VkVertexInputBindingDescription must be less than this limit. See Section 19.2.
- *maxVertexInputAttributeOffset* is the maximum vertex input attribute offset that can be added to the vertex input binding stride. The *offset* member of the VkVertexInputAttributeDescription structure must be less than or equal to this limit. See Section 19.2.
- *maxVertexInputBindingStride* is the maximum vertex input binding stride that can be specified in a vertex input binding. The *stride* member of the VkVertexInputBindingDescription structure must be less than or equal to this limit. See Section 19.2.
- *maxVertexOutputComponents* is the maximum number of components of output variables which can be output by a vertex shader. See Section 8.5
- *maxTessellationGenerationLevel* is the maximum tessellation generation level supported by the fixed-function tessellation primitive generator. See Chapter 20
- *maxTessellationPatchSize* is the maximum patch size, in vertices, of patches that can be processed by the tessellation control shader and tessellation primitive generator. The tessellation patch sizes are controlled by the *patchControlPoints* member of the VkPipelineTessellationStateCreateInfo structure and the **OutputVertices** execution mode of the tessellation control shader. See Chapter 20
- *maxTessellationControlPerVertexInputComponents* is the maximum number of components of input variables which can be provided as per-vertex inputs to the tessellation control shader stage.
- *maxTessellationControlPerVertexOutputComponents* is the maximum number of components of per-vertex output variables which can be output from the tessellation control shader stage.
- *maxTessellationControlPerPatchOutputComponents* is the maximum number of components of per-patch output variables which can be output from the tessellation control shader stage.
- *maxTessellationControlTotalOutputComponents* is the maximum total number of components of per-vertex and per-patch output variables which can be output from the tessellation control shader stage.
- *maxTessellationEvaluationInputComponents* is the maximum number of components of input variables which can be provided as per-vertex inputs to the tessellation evaluation shader stage.
- *maxTessellationEvaluationOutputComponents* is the maximum number of components of per-vertex output variables which can be output from the tessellation evaluation shader stage.
- *maxGeometryShaderInvocations* is the maximum invocation count supported for instanced geometry shaders. See Chapter 21

- *maxGeometryInputComponents* is the maximum number of components of input variables which can be provided as inputs to the geometry shader stage.
- *maxGeometryOutputComponents* is the maximum number of components of output variables which can be output from the geometry shader stage.
- *maxGeometryOutputVertices* is the maximum number of vertices which can be emitted by any geometry shader.
- *maxGeometryTotalOutputComponents* is the maximum total number of components of output, across all emitted vertices, which can be output from the geometry shader stage.
- *maxFragmentInputComponents* is the maximum number of components of input variables which can be provided as inputs to the fragment shader stage.
- *maxFragmentOutputAttachments* is the maximum number of output attachments which can be written to by the fragment shader stage.
- *maxFragmentDualSrcAttachments* is the maximum number of output attachments which can be written to by the fragment shader stage when blending is enabled and one of the dual source blend modes is in use. See Section 25.1.2 and dualSrcBlend.
- *maxFragmentCombinedOutputResources* is the total number of storage buffers, storage images, and output buffers which can be used in the fragment shader stage.
- *maxComputeSharedMemorySize* is the maximum total storage size, in bytes, of all variables declared with the **WorkgroupLocal** SPIR-V Storage Class (the *shared* storage qualifier in GLSL) in the compute shader stage.
- *maxComputeWorkGroupCount*[3] is the maximum number of work groups that can be dispatched by a single dispatch command. These three values represent the maximum number of work groups for the X, Y, and Z dimensions, respectively. The *x*, *y*, and *z* parameters to the `vkCmdDispatch` command, or members of the `VkDispatchIndirectCommand` structure must be less than or equal to the corresponding limit. See Chapter 26.
- *maxComputeWorkGroupInvocations* is the maximum total number of compute shader invocations in a single local work group. The product of the *x*, *y*, and *z* sizes as specified by **LocalSize** in the SPIR-V Execution Mode must be no larger than this limit.
- *maxComputeWorkGroupSize*[3] is the maximum size of a local compute work group, per dimension. These three values represent the maximum local work group size in the X, Y, and Z dimensions, respectively. The *x*, *y*, and *z* sizes specified by the **LocalSize** Execution Mode in SPIR-V must be less than or equal to the corresponding limit.
- *subPixelPrecisionBits* is the number of bits of subpixel precision in framebuffer coordinates x_f and y_f. See Chapter 23.
- *subTexelPrecisionBits* is the number of bits of precision in the division along an axis of a texture used for minification and magnification filters. $2^{\text{subTexelPrecisionBits}}$ is the actual number of divisions along each axis of the texture represented. The filtering hardware will snap to these locations when computing the filtered results.
- *mipmapPrecisionBits* is the number of bits of division that the LOD calculation for mipmap fetching get snapped to when determining the contribution from each miplevel to the mip filtered results. $2^{\text{mipmapPrecisionBits}}$ is the actual number of divisions.

Note
For example, if this value is 2 bits then when linearly filtering between two levels, each level could: contribute: 0%, 33%, 66%, or 100% (this is just an example and the amount of contribution should be covered by different equations in the spec).

- *maxDrawIndexedIndexValue* is the maximum index value that can be used for indexed draw calls when using 32-bit indices. This excludes the primitive restart index value of 0xFFFFFFFF. See fullDrawIndexUint32.
- *maxDrawIndirectCount* is the maximum draw count that is supported for indirect draw calls. See multiDrawIndirect.
- *maxSamplerLodBias* is the maximum absolute sampler level of detail bias. The sum of the *mipLodBias* member of the VkSamplerCreateInfo structure and the bias provided by the SPIR-V image sampling functions (or 0 if no bias is provided) are clamped to the range [-maxSamplerLodBias,maxSamplerLodBias]. See [samplers-mipLodBias].
- *maxSamplerAnisotropy* is the maximum degree of sampler anisotropy. The maximum degree of anisotropic filtering used for an image sampling operation is the minimum of the *maxAnisotropy* member of the VkSamplerCreateInfo structure and this limit. See [samplers-maxAnisotropy].
- *maxViewports* is the maximum number of active viewports. The *viewportCount* member of the VkPipelineViewportStateCreateInfo structure that is provided at pipeline creation must be less than or equal to this limit.
- *maxViewportDimensions*[2] are the maximum viewport dimensions in the X (width) and Y (height) dimensions, respectively. The maximum viewport dimensions must be greater than or equal to the larger of: the visible dimensions of the display being rendered to (if a display exists), and the largest image which can be created and used as a framebuffer attachment. The viewport width and height (see Controlling the Viewport) are clamped to these limits.
- *viewportBoundsRange*[2] is the viewport bounds range [minimum, maximum]. The location of a viewport's upper-left corner (see Controlling the Viewport) is clamped to these limits.
- *viewportSubPixelBits* is the number of bits of subpixel precision for viewport bounds. The subpixel precision that floating-point viewport bounds are interpreted at is given by this limit.
- *minMemoryMapAlignment* is the minimum required alignment, in bytes, of host-visible memory allocations within the host address space. When mapping a memory allocation with `vkMapMemory`, subtracting *offset* bytes from the returned pointer will always produce a multiple of this limit. See Section 10.2.1.
- *minTexelBufferOffsetAlignment* is the minimum required alignment, in bytes, for the *offset* member of the VkBufferViewCreateInfo structure for texel buffers. When a buffer view is created for a buffer which was created with VK_BUFFER_USAGE_UNIFORM_TEXEL_BUFFER_BIT or VK_BUFFER_USAGE_STORAGE_TEXEL_BUFFER_BIT set in the *usage* member of the VkBufferCreateInfo structure, the *offset* must be an integer multiple of this limit. This limit is a maximum, not a minimum.
- *minUniformBufferOffsetAlignment* is the minimum required alignment, in bytes, for the *offset* member of the VkDescriptorBufferInfo structure for uniform buffers. When a descriptor of type VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER or VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC is updated to reference a buffer which was created with the VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT set in the *usage* member of the VkBufferCreateInfo structure, the *offset* must be an integer multiple of this limit. This limit is a maximum, not a minimum.
- *minStorageBufferOffsetAlignment* is the minimum required alignment, in bytes, for the *offset* member of the VkDescriptorBufferInfo structure for storage buffers. When a descriptor of type VK_DESCRIPTOR_TYPE_STORAGE_BUFFER or VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC is updated to reference a buffer which was created with the VK_BUFFER_USAGE_STORAGE_BUFFER_BIT set in the *usage* member of the VkBufferCreateInfo structure, the *offset* must be an integer multiple of this limit. This limit is a maximum, not a minimum.
- *minTexelOffset* is the minimum offset value for the **ConstOffset** image operand of any of the **OpImageSample*** or **OpImageFetch** SPIR-V image instructions.

- *`maxTexelOffset`* is the maximum offset value for the **ConstOffset** image operand of any of the **OpImageSample*** or **OpImageFetch** SPIR-V image instructions.
- *`minTexelGatherOffset`* is the minimum offset value for the **Offset** or **ConstOffsets** image operands of any of the **OpImageGather** or **OpImageDrefGather** SPIR-V image instructions.
- *`maxTexelGatherOffset`* is the maximum offset value for the **Offset** or **ConstOffsets** image operands of any of the **OpImageGather** or **OpImageDrefGather** SPIR-V image instructions.
- *`minInterpolationOffset`* is the minimum negative offset value for the *offset* operand of the **InterpolateAtOffset** SPIR-V extended instruction.
- *`maxInterpolationOffset`* is the maximum positive offset value for the *offset* operand of the **InterpolateAtOffset** SPIR-V extended instruction.
- *`subPixelInterpolationOffsetBits`* is the number of subpixel fractional bits that the *x* and *y* offsets to the **InterpolateAtOffset** SPIR-V extended instruction may be rounded to as fixed-point values.
- *`maxFramebufferWidth`* is the maximum width for a framebuffer. The *`width`* member of the VkFramebufferCreateInfo structure must be less than or equal to this limit.
- *`maxFramebufferHeight`* is the maximum height for a framebuffer. The *`height`* member of the VkFramebufferCreateInfo structure must be less than or equal to this limit.
- *`maxFramebufferLayers`* is the maximum layer count for a layered framebuffer. The *`layers`* member of the VkFramebufferCreateInfo structure must be less than or equal to this limit.
- *`framebufferColorSampleCounts`* is a bitmask[1] of `VkSampleCountFlagBits` bits indicating the supported color sample counts for a framebuffer color attachment.
- *`framebufferDepthSampleCounts`* is a bitmask[1] of `VkSampleCountFlagBits` bits indicating the supported depth sample counts for a framebuffer depth attachment.
- *`framebufferStencilSampleCounts`* is a bitmask[1] of `VkSampleCountFlagBits` bits indicating the supported stencil sample counts for a framebuffer stencil attachment.
- *`framebufferNoAttachmentsSampleCounts`* is a bitmask[1] of `VkSampleCountFlagBits` bits indicating the supported sample counts for a framebuffer with no attachments.
- *`maxColorAttachments`* is the maximum number of color attachments that can be referenced by a subpass in a render pass. The *`colorAttachmentCount`* member of the VkSubpassDescription structure must be less than or equal to this limit.
- *`sampledImageColorSampleCounts`* is a bitmask[1] of `VkSampleCountFlagBits` bits indicating the sample counts supported for all images with a non-integer color format.
- *`sampledImageIntegerSampleCounts`* is a bitmask[1] of `VkSampleCountFlagBits` bits indicating the sample counts supported for all images with a integer color format.
- *`sampledImageDepthSampleCounts`* is a bitmask[1] of `VkSampleCountFlagBits` bits indicating the sample counts supported for all images with a depth format.
- *`sampledImageStencilSampleCounts`* is a bitmask[1] of `VkSampleCountFlagBits` bits indicating the sample supported for all images with a stencil format.
- *`storageImageSampleCounts`* is a bitmask[1] of `VkSampleCountFlagBits` bits indicating the sample counts supported for all images used for storage operations.

- *maxSampleMaskWords* is the maximum number of components in the **SampleMask** or **SampleMaskIn** shader built-in.
- *timestampComputeAndGraphics* indicates support for timestamps on all graphics and compute queues. If this limit is set to VK_TRUE, all queues that advertise the VK_QUEUE_GRAPHICS_BIT or VK_QUEUE_COMPUTE_BIT in the VkQueueFamilyProperties::*queueFlags* support VkQueueFamilyProperties::*timestampValidBits* of at least 36. See Timestamp Queries.
- *timestampPeriod* is the number of nanoseconds required for a timestamp query to be incremented by 1. See Timestamp Queries.
- *maxClipDistances* is the maximum number of clip distances that can be written to via the **ClipDistance** shader built-in in a single shader stage.
- *maxCullDistances* is the maximum number of cull distances that can be written to via the **CullDistance** shader built-in in a single shader stage.
- *maxCombinedClipAndCullDistances* is the maximum combined number of clip and cull distances that can be written to via the **ClipDistance** and **CullDistances** shader built-ins in a single shader stage.
- *discreteQueuePriorities* is the number of discrete priorities that can be assigned to a queue based on the value of each member of VkDeviceQueueCreateInfo::*pQueuePriorities*. This must be at least 2, and levels must be spread evenly over the range, with at least one level at 1.0, and another at 0.0. See Section 4.3.4.
- *pointSizeRange*[2] is the range [minimum,maximum] of supported sizes for points. Values written to the **PointSize** shader built-in are clamped to this range.
- *lineWidthRange*[2] is the range [minimum,maximum] of supported widths for lines. Values specified by the *lineWidth* member of the VkPipelineRasterizationStateCreateInfo or the *lineWidth* parameter to **vkCmdSetLineWidth** are clamped to this range.
- *pointSizeGranularity* is the granularity of supported point sizes. Not all point sizes in the range defined by *pointSizeRange* are supported. This limit specifies the granularity (or increment) between successive supported point sizes.
- *lineWidthGranularity* is the granularity of supported line widths. Not all line widths in the range defined by *lineWidthRange* are supported. This limit specifies the granularity (or increment) between successive supported line widths.
- *strictLines* indicates whether lines are rasterized according to the preferred method of rasterization. If set to VK_FALSE, lines may be rasterized under a relaxed set of rules. If set to VK_TRUE, lines are rasterized as per the strict definition. See Basic Line Segment Rasterization.
- *standardSampleLocations* indicates whether rasterization uses the standard sample locations as documented in Multisampling. If set to VK_TRUE, the implementation uses the documented sample locations. If set to VK_FALSE, the implementation may use different sample locations.
- *optimalBufferCopyOffsetAlignment* is the optimal buffer view offset alignment in bytes for **vkCmdCopyBufferToImage** and **vkCmdCopyImageToBuffer**. The per texel alignment requirements are still enforced, this is just an additional alignment recommendation for optimal performance and power.
- *optimalBufferCopyRowPitchAlignment* is the optimal buffer row pitch alignment in bytes for **vkCmdCopyBufferToImage** and **vkCmdCopyImageToBuffer**. Row pitch is the number of bytes between texels with the same x coordinate in adjacent rows (y coordinates differ by one). The per texel alignment requirements are still enforced, this is just an additional alignment recommendation for optimal performance and power.

- `nonCoherentAtomSize` is the size and alignment in bytes that bounds concurrent access to host-mapped device memory.

1

For all bitmasks of type `VkSampleCountFlags` above, the bits which can be set include:

```
typedef enum VkSampleCountFlagBits {
    VK_SAMPLE_COUNT_1_BIT = 0x00000001,
    VK_SAMPLE_COUNT_2_BIT = 0x00000002,
    VK_SAMPLE_COUNT_4_BIT = 0x00000004,
    VK_SAMPLE_COUNT_8_BIT = 0x00000008,
    VK_SAMPLE_COUNT_16_BIT = 0x00000010,
    VK_SAMPLE_COUNT_32_BIT = 0x00000020,
    VK_SAMPLE_COUNT_64_BIT = 0x00000040,
} VkSampleCountFlagBits;
```

The sample count limits defined above represent the minimum supported sample counts for each image type. Individual images may support additional sample counts, which are queried using `vkGetPhysicalDeviceImageFormatProperties`. The sample count limits for images only apply to images created with the *tiling* set to VK_IMAGE_TILING_OPTIMAL. For VK_IMAGE_TILING_ LINEAR images the only supported sample count is VK_SAMPLE_COUNT_1_BIT.

29.2.1 Limit Requirements

The following table specifies the required minimum/maximum for all Vulkan graphics implementations. Where a limit corresponds to a fine-grained device feature which is optional, the feature name is listed with two required limits, one when the feature is supported and one when it is not supported. If an implementation supports a feature, the limits reported are the same whether or not the feature is enabled.

Table 29.1: Required Limit Types

Type	Limit	Feature
uint32_t	maxImageDimension1D	-
uint32_t	maxImageDimension2D	-
uint32_t	maxImageDimension3D	-
uint32_t	maxImageDimensionCube	-
uint32_t	maxImageArrayLayers	-
uint32_t	maxTexelBufferElements	-
uint32_t	maxUniformBufferRange	-
uint32_t	maxStorageBufferRange	-
uint32_t	maxPushConstantsSize	-
uint32_t	maxMemoryAllocationCount	-
uint32_t	maxSamplerAllocationCount	-
VkDeviceSize	bufferImageGranularity	-
VkDeviceSize	sparseAddressSpaceSize	sparseBinding
uint32_t	maxBoundDescriptorSets	-
uint32_t	maxPerStageDescriptorSamplers	-
uint32_t	maxPerStageDescriptorUniformBuffers	-
uint32_t	maxPerStageDescriptorStorageBuffers	-
uint32_t	maxPerStageDescriptorSampledImages	-
uint32_t	maxPerStageDescriptorStorageImages	-

Table 29.1: (continued)

Type	Limit	Feature
uint32_t	maxPerStageDescriptorInputAttachments	-
uint32_t	maxPerStageResources	-
uint32_t	maxDescriptorSetSamplers	-
uint32_t	maxDescriptorSetUniformBuffers	-
uint32_t	maxDescriptorSetUniformBuffersDynamic	-
uint32_t	maxDescriptorSetStorageBuffers	-
uint32_t	maxDescriptorSetStorageBuffersDynamic	-
uint32_t	maxDescriptorSetSampledImages	-
uint32_t	maxDescriptorSetStorageImages	-
uint32_t	maxDescriptorSetInputAttachments	-
uint32_t	maxVertexInputAttributes	-
uint32_t	maxVertexInputBindings	-
uint32_t	maxVertexInputAttributeOffset	-
uint32_t	maxVertexInputBindingStride	-
uint32_t	maxVertexOutputComponents	-
uint32_t	maxTessellationGenerationLevel	tessellationShader
uint32_t	maxTessellationPatchSize	tessellationShader
uint32_t	maxTessellationControlPerVertexInputComponents	tessellationShader
uint32_t	maxTessellationControlPerVertexOutputComponents	tessellationShader
uint32_t	maxTessellationControlPerPatchOutputComponents	tessellationShader
uint32_t	maxTessellationControlTotalOutputComponents	tessellationShader
uint32_t	maxTessellationEvaluationInputComponents	tessellationShader
uint32_t	maxTessellationEvaluationOutputComponents	tessellationShader
uint32_t	maxGeometryShaderInvocations	geometryShader
uint32_t	maxGeometryInputComponents	geometryShader
uint32_t	maxGeometryOutputComponents	geometryShader
uint32_t	maxGeometryOutputVertices	geometryShader
uint32_t	maxGeometryTotalOutputComponents	geometryShader
uint32_t	maxFragmentInputComponents	-
uint32_t	maxFragmentOutputAttachments	-
uint32_t	maxFragmentDualSrcAttachments	dualSrcBlend
uint32_t	maxFragmentCombinedOutputResources	-
uint32_t	maxComputeSharedMemorySize	-
3 × uint32_t	maxComputeWorkGroupCount	-
uint32_t	maxComputeWorkGroupInvocations	-
3 × uint32_t	maxComputeWorkGroupSize	-
uint32_t	subPixelPrecisionBits	-
uint32_t	subTexelPrecisionBits	-
uint32_t	mipmapPrecisionBits	-
uint32_t	maxDrawIndexedIndexValue	fullDrawIndexUint32
uint32_t	maxDrawIndirectCount	multiDrawIndirect
float	maxSamplerLodBias	-
float	maxSamplerAnisotropy	samplerAnisotropy
uint32_t	maxViewports	multiViewport
2 × uint32_t	maxViewportDimensions	-
2 × float	viewportBoundsRange	-
uint32_t	viewportSubPixelBits	-
size_t	minMemoryMapAlignment	-
VkDeviceSize	minTexelBufferOffsetAlignment	-

Table 29.1: (continued)

Type	**Limit**	**Feature**
VkDeviceSize	minUniformBufferOffsetAlignment	-
VkDeviceSize	minStorageBufferOffsetAlignment	-
int32_t	minTexelOffset	-
uint32_t	maxTexelOffset	-
int32_t	minTexelGatherOffset	shaderImageGatherExtended
uint32_t	maxTexelGatherOffset	shaderImageGatherExtended
float	minInterpolationOffset	sampleRateShading
float	maxInterpolationOffset	sampleRateShading
uint32_t	subPixelInterpolationOffsetBits	sampleRateShading
uint32_t	maxFramebufferWidth	-
uint32_t	maxFramebufferHeight	-
uint32_t	maxFramebufferLayers	-
VkSampleCountFlags	framebufferColorSampleCounts	-
VkSampleCountFlags	framebufferDepthSampleCounts	-
VkSampleCountFlags	framebufferStencilSampleCounts	-
VkSampleCountFlags	framebufferNoAttachmentsSampleCounts	-
uint32_t	maxColorAttachments	-
VkSampleCountFlags	sampledImageColorSampleCounts	-
VkSampleCountFlags	sampledImageIntegerSampleCounts	-
VkSampleCountFlags	sampledImageDepthSampleCounts	-
VkSampleCountFlags	sampledImageStencilSampleCounts	-
VkSampleCountFlags	storageImageSampleCounts	shaderStorageImageMultisample
uint32_t	maxSampleMaskWords	-
vkBool32	timestampComputeAndGraphics	-
float	timestampPeriod	-
uint32_t	maxClipDistances	shaderClipDistance
uint32_t	maxCullDistances	shaderCullDistance
uint32_t	maxCombinedClipAndCullDistances	shaderCullDistance
uint32_t	discreteQueuePriorities	-
2 × float	pointSizeRange	largePoints
2 × float	lineWidthRange	wideLines
float	pointSizeGranularity	largePoints
float	lineWidthGranularity	wideLines
VkBool32	strictLines	-
VkBool32	standardSampleLocations	-
VkDeviceSize	optimalBufferCopyOffsetAlignment	-
VkDeviceSize	optimalBufferCopyRowPitchAlignment	-
VkDeviceSize	nonCoherentAtomSize	-

Table 29.2: Required Limits

Limit	**Unsupported Limit**	**Supported Limit**	**Limit Type**[1]
maxImageDimension1D	-	4096	min
maxImageDimension2D	-	4096	min
maxImageDimension3D	-	256	min

Table 29.2: (continued)

Limit	**Unsupported Limit**	**Supported Limit**	**Limit Type**[1]
maxImageDimensionCube	-	4096	min
maxImageArrayLayers	-	256	min
maxTexelBufferElements	-	65536	min
maxUniformBufferRange	-	16384	min
maxStorageBufferRange	-	2^{27}	min
maxPushConstantsSize	-	128	min
maxMemoryAllocationCount	-	4096	min
maxSamplerAllocationCount	-	4000	min
bufferImageGranularity	-	131072	max
sparseAddressSpaceSize	0	2 GB	min
maxBoundDescriptorSets	-	4	min
maxPerStageDescriptorSamplers	-	16	min
maxPerStageDescriptorUniformBuffers	-	12	min
maxPerStageDescriptorStorageBuffers	-	4	min
maxPerStageDescriptorSampledImages	-	16	min
maxPerStageDescriptorStorageImages	-	4	min
maxPerStageDescriptorInputAttachments	-	4	min
maxPerStageResources	-	128 [2]	min
maxDescriptorSetSamplers	-	96	min, 6×PerStage
maxDescriptorSetUniformBuffers	-	72	min, 6×PerStage
maxDescriptorSetUniformBuffersDynamic	-	8	min
maxDescriptorSetStorageBuffers	-	24	min, 6×PerStage
maxDescriptorSetStorageBuffersDynamic	-	4	min
maxDescriptorSetSampledImages	-	96	min, 6×PerStage
maxDescriptorSetStorageImages	-	24	min, 6×PerStage
maxDescriptorSetInputAttachments	-	4	min
maxVertexInputAttributes	-	16	min
maxVertexInputBindings	-	16	min
maxVertexInputAttributeOffset	-	2047	min
maxVertexInputBindingStride	-	2048	min
maxVertexOutputComponents	-	64	min
maxTessellationGenerationLevel	0	64	min
maxTessellationPatchSize	0	32	min
maxTessellationControlPerVertexInputComponents	0	64	min
maxTessellationControlPerVertexOutputComponents	0	64	min
maxTessellationControlPerPatchOutputComponents	0	120	min
maxTessellationControlTotalOutputComponents	0	2048	min
maxTessellationEvaluationInputComponents	0	64	min
maxTessellationEvaluationOutputComponents	0	64	min
maxGeometryShaderInvocations	0	32	min
maxGeometryInputComponents	0	64	min
maxGeometryOutputComponents	0	64	min
maxGeometryOutputVertices	0	256	min

Table 29.2: (continued)

Limit	**Unsupported Limit**	**Supported Limit**	**Limit Type**[1]
maxGeometryTotalOutputComponents	0	1024	min
maxFragmentInputComponents	-	64	min
maxFragmentOutputAttachments	-	4	min
maxFragmentDualSrcAttachments	0	1	min
maxFragmentCombinedOutputResources	-	4	min
maxComputeSharedMemorySize	-	16384	min
maxComputeWorkGroupCount	-	(65536,65536,65536)	min
maxComputeWorkGroupInvocations	-	128	min
maxComputeWorkGroupSize	-	(128,128,64)	min
subPixelPrecisionBits	-	4	min
subTexelPrecisionBits	-	4	min
mipmapPrecisionBits	-	4	min
maxDrawIndexedIndexValue	2^{24}-1	2^{32}-1	min
maxDrawIndirectCount	1	2^{16}-1	min
maxSamplerLodBias	-	2	min
maxSamplerAnisotropy	1	16	min
maxViewports	1	16	min
maxViewportDimensions	-	(4096,4096) [3]	min
viewportBoundsRange	-	(-8192,8191) [4]	(max,min)
viewportSubPixelBits	-	0	min
minMemoryMapAlignment	-	64	min
minTexelBufferOffsetAlignment	-	256	max
minUniformBufferOffsetAlignment	-	256	max
minStorageBufferOffsetAlignment	-	256	max
minTexelOffset	-	-8	max
maxTexelOffset	-	7	min
minTexelGatherOffset	0	-8	max
maxTexelGatherOffset	0	7	min
minInterpolationOffset	0.0	-0.5 [5]	max
maxInterpolationOffset	0.0	0.5 - (1 ULP) [5]	min
subPixelInterpolationOffsetBits	0	4 [5]	min
maxFramebufferWidth	-	4096	min
maxFramebufferHeight	-	4096	min
maxFramebufferLayers	-	256	min
framebufferColorSampleCounts	-	(VK_SAMPLE_ COUNT_1_BIT | VK_SAMPLE_ COUNT_4_BIT)	min
framebufferDepthSampleCounts	-	(VK_SAMPLE_ COUNT_1_BIT | VK_SAMPLE_ COUNT_4_BIT)	min
framebufferStencilSampleCounts	-	(VK_SAMPLE_ COUNT_1_BIT | VK_SAMPLE_ COUNT_4_BIT)	min

Table 29.2: (continued)

Limit	**Unsupported Limit**	**Supported Limit**	**Limit Type**[1]
framebufferNoAttachmentsSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT | VK_SAMPLE_COUNT_4_BIT)	min
maxColorAttachments	-	4	min
sampledImageColorSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT | VK_SAMPLE_COUNT_4_BIT)	min
sampledImageIntegerSampleCounts	-	VK_SAMPLE_COUNT_1_BIT	min
sampledImageDepthSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT | VK_SAMPLE_COUNT_4_BIT)	min
sampledImageStencilSampleCounts	-	(VK_SAMPLE_COUNT_1_BIT | VK_SAMPLE_COUNT_4_BIT)	min
storageImageSampleCounts	VK_SAMPLE_COUNT_1_BIT	(VK_SAMPLE_COUNT_1_BIT | VK_SAMPLE_COUNT_4_BIT)	min
maxSampleMaskWords	-	1	min
timestampComputeAndGraphics	-	-	implementation dependent
timestampPeriod	-	-	duration
maxClipDistances	0	8	min
maxCullDistances	0	8	min
maxCombinedClipAndCullDistances	0	8	min
discreteQueuePriorities	-	2	min
pointSizeRange	(1.0,1.0)	(1.0,64.0 - ULP)[6]	(max,min)
lineWidthRange	(1.0,1.0)	(1.0,8.0 - ULP)[7]	(max,min)
pointSizeGranularity	0.0	1.0 [6]	max, fixed point increment
lineWidthGranularity	0.0	1.0 [7]	max, fixed point increment
strictLines	-	-	implementation dependent
standardSampleLocations	-	-	implementation dependent
optimalBufferCopyOffsetAlignment	-	-	recommendation
optimalBufferCopyRowPitchAlignment	-	-	recommendation
nonCoherentAtomSize	-	128	max

1

The **Limit Type** column indicates the limit is either the minimum limit all implementations must support or the maximum limit all implementations must support. For bitfields a minimum limit is the least bits all implementations must set, but they may have additional bits set beyond this minimum.

2

The *maxPerStageResources* must be at least the smallest of the following:

- the sum of the *maxPerStageDescriptorUniformBuffers*, *maxPerStageDescriptorStorageBuffers*, *maxPerStageDescriptorSampledImages*, *maxPerStageDescriptorStorageImages*, *maxPerStageDescriptorInputAttachments*, *maxColorAttachments* limits, or
- 128.

It may not be possible to reach this limit in every stage.

3

Maximum image attachment size or maximum display size

4

Double the *maxViewportDimensions*

5

The values *minInterpolationOffset* and *maxInterpolationOffset* describe the closed interval of supported interpolation offsets: [*minInterpolationOffset*, *maxInterpolationOffset*]. The ULP is determined by *subPixelInterpolationOffsetBits*. If *subPixelInterpolationOffsetBits* is 4, this provides increments of $(1/2^4) = 0.0625$, and thus the range of supported interpolation offsets would be [-0.5, 0.4375].

6

The point size ULP is determined by *pointSizeGranularity*. If the *pointSizeGranularity* is 0.125, the range of supported point sizes must be at least [1.0, 63.875].

7

The line width ULP is determined by *lineWidthGranularity*. If the *lineWidthGranularity* is 0.0625, the range of supported line widths must be at least [1.0, 7.9375].

29.3 Formats

The features for the set of formats (`VkFormat`) supported by the implementation are queried individually using the `vkGetPhysicalDeviceFormatProperties` command.

29.3.1 Format Definition

The available formats available are defined by the `VkFormat` enumeration:

```
typedef enum VkFormat {
    VK_FORMAT_UNDEFINED = 0,
    VK_FORMAT_R4G4_UNORM_PACK8 = 1,
    VK_FORMAT_R4G4B4A4_UNORM_PACK16 = 2,
    VK_FORMAT_B4G4R4A4_UNORM_PACK16 = 3,
    VK_FORMAT_R5G6B5_UNORM_PACK16 = 4,
    VK_FORMAT_B5G6R5_UNORM_PACK16 = 5,
    VK_FORMAT_R5G5B5A1_UNORM_PACK16 = 6,
    VK_FORMAT_B5G5R5A1_UNORM_PACK16 = 7,
    VK_FORMAT_A1R5G5B5_UNORM_PACK16 = 8,
    VK_FORMAT_R8_UNORM = 9,
    VK_FORMAT_R8_SNORM = 10,
    VK_FORMAT_R8_USCALED = 11,
    VK_FORMAT_R8_SSCALED = 12,
    VK_FORMAT_R8_UINT = 13,
    VK_FORMAT_R8_SINT = 14,
    VK_FORMAT_R8_SRGB = 15,
    VK_FORMAT_R8G8_UNORM = 16,
    VK_FORMAT_R8G8_SNORM = 17,
    VK_FORMAT_R8G8_USCALED = 18,
    VK_FORMAT_R8G8_SSCALED = 19,
    VK_FORMAT_R8G8_UINT = 20,
    VK_FORMAT_R8G8_SINT = 21,
    VK_FORMAT_R8G8_SRGB = 22,
    VK_FORMAT_R8G8B8_UNORM = 23,
    VK_FORMAT_R8G8B8_SNORM = 24,
    VK_FORMAT_R8G8B8_USCALED = 25,
    VK_FORMAT_R8G8B8_SSCALED = 26,
    VK_FORMAT_R8G8B8_UINT = 27,
    VK_FORMAT_R8G8B8_SINT = 28,
    VK_FORMAT_R8G8B8_SRGB = 29,
    VK_FORMAT_B8G8R8_UNORM = 30,
    VK_FORMAT_B8G8R8_SNORM = 31,
    VK_FORMAT_B8G8R8_USCALED = 32,
    VK_FORMAT_B8G8R8_SSCALED = 33,
    VK_FORMAT_B8G8R8_UINT = 34,
    VK_FORMAT_B8G8R8_SINT = 35,
    VK_FORMAT_B8G8R8_SRGB = 36,
    VK_FORMAT_R8G8B8A8_UNORM = 37,
    VK_FORMAT_R8G8B8A8_SNORM = 38,
    VK_FORMAT_R8G8B8A8_USCALED = 39,
    VK_FORMAT_R8G8B8A8_SSCALED = 40,
    VK_FORMAT_R8G8B8A8_UINT = 41,
    VK_FORMAT_R8G8B8A8_SINT = 42,
    VK_FORMAT_R8G8B8A8_SRGB = 43,
    VK_FORMAT_B8G8R8A8_UNORM = 44,
    VK_FORMAT_B8G8R8A8_SNORM = 45,
    VK_FORMAT_B8G8R8A8_USCALED = 46,
    VK_FORMAT_B8G8R8A8_SSCALED = 47,
    VK_FORMAT_B8G8R8A8_UINT = 48,
    VK_FORMAT_B8G8R8A8_SINT = 49,
    VK_FORMAT_B8G8R8A8_SRGB = 50,
    VK_FORMAT_A8B8G8R8_UNORM_PACK32 = 51,
    VK_FORMAT_A8B8G8R8_SNORM_PACK32 = 52,
    VK_FORMAT_A8B8G8R8_USCALED_PACK32 = 53,
    VK_FORMAT_A8B8G8R8_SSCALED_PACK32 = 54,
    VK_FORMAT_A8B8G8R8_UINT_PACK32 = 55,
    VK_FORMAT_A8B8G8R8_SINT_PACK32 = 56,
    VK_FORMAT_A8B8G8R8_SRGB_PACK32 = 57,
    VK_FORMAT_A2R10G10B10_UNORM_PACK32 = 58,
    VK_FORMAT_A2R10G10B10_SNORM_PACK32 = 59,
    VK_FORMAT_A2R10G10B10_USCALED_PACK32 = 60,
    VK_FORMAT_A2R10G10B10_SSCALED_PACK32 = 61,
    VK_FORMAT_A2R10G10B10_UINT_PACK32 = 62,
    VK_FORMAT_A2R10G10B10_SINT_PACK32 = 63,
    VK_FORMAT_A2B10G10R10_UNORM_PACK32 = 64,
    VK_FORMAT_A2B10G10R10_SNORM_PACK32 = 65,
    VK_FORMAT_A2B10G10R10_USCALED_PACK32 = 66,
    VK_FORMAT_A2B10G10R10_SSCALED_PACK32 = 67,
    VK_FORMAT_A2B10G10R10_UINT_PACK32 = 68,
    VK_FORMAT_A2B10G10R10_SINT_PACK32 = 69,
    VK_FORMAT_R16_UNORM = 70,
    VK_FORMAT_R16_SNORM = 71,
    VK_FORMAT_R16_USCALED = 72,
    VK_FORMAT_R16_SSCALED = 73,
    VK_FORMAT_R16_UINT = 74,
    VK_FORMAT_R16_SINT = 75,
    VK_FORMAT_R16_SFLOAT = 76,
    VK_FORMAT_R16G16_UNORM = 77,
    VK_FORMAT_R16G16_SNORM = 78,
    VK_FORMAT_R16G16_USCALED = 79,
    VK_FORMAT_R16G16_SSCALED = 80,
    VK_FORMAT_R16G16_UINT = 81,
    VK_FORMAT_R16G16_SINT = 82,
    VK_FORMAT_R16G16_SFLOAT = 83,
    VK_FORMAT_R16G16B16_UNORM = 84,
    VK_FORMAT_R16G16B16_SNORM = 85,
    VK_FORMAT_R16G16B16_USCALED = 86,
    VK_FORMAT_R16G16B16_SSCALED = 87,
    VK_FORMAT_R16G16B16_UINT = 88,
    VK_FORMAT_R16G16B16_SINT = 89,
    VK_FORMAT_R16G16B16_SFLOAT = 90,
    VK_FORMAT_R16G16B16A16_UNORM = 91,
    VK_FORMAT_R16G16B16A16_SNORM = 92,
    VK_FORMAT_R16G16B16A16_USCALED = 93,
    VK_FORMAT_R16G16B16A16_SSCALED = 94,
    VK_FORMAT_R16G16B16A16_UINT = 95,
    VK_FORMAT_R16G16B16A16_SINT = 96,
    VK_FORMAT_R16G16B16A16_SFLOAT = 97,
    VK_FORMAT_R32_UINT = 98,
    VK_FORMAT_R32_SINT = 99,
    VK_FORMAT_R32_SFLOAT = 100,
    VK_FORMAT_R32G32_UINT = 101,
    VK_FORMAT_R32G32_SINT = 102,
    VK_FORMAT_R32G32_SFLOAT = 103,
    VK_FORMAT_R32G32B32_UINT = 104,
    VK_FORMAT_R32G32B32_SINT = 105,
    VK_FORMAT_R32G32B32_SFLOAT = 106,
    VK_FORMAT_R32G32B32A32_UINT = 107,
    VK_FORMAT_R32G32B32A32_SINT = 108,
    VK_FORMAT_R32G32B32A32_SFLOAT = 109,
    VK_FORMAT_R64_UINT = 110,
    VK_FORMAT_R64_SINT = 111,
    VK_FORMAT_R64_SFLOAT = 112,
    VK_FORMAT_R64G64_UINT = 113,
    VK_FORMAT_R64G64_SINT = 114,
    VK_FORMAT_R64G64_SFLOAT = 115,
    VK_FORMAT_R64G64B64_UINT = 116,
    VK_FORMAT_R64G64B64_SINT = 117,
    VK_FORMAT_R64G64B64_SFLOAT = 118,
    VK_FORMAT_R64G64B64A64_UINT = 119,
    VK_FORMAT_R64G64B64A64_SINT = 120,
    VK_FORMAT_R64G64B64A64_SFLOAT = 121,
    VK_FORMAT_B10G11R11_UFLOAT_PACK32 = 122,
    VK_FORMAT_E5B9G9R9_UFLOAT_PACK32 = 123,
    VK_FORMAT_D16_UNORM = 124,
    VK_FORMAT_X8_D24_UNORM_PACK32 = 125,
    VK_FORMAT_D32_SFLOAT = 126,
    VK_FORMAT_S8_UINT = 127,
    VK_FORMAT_D16_UNORM_S8_UINT = 128,
    VK_FORMAT_D24_UNORM_S8_UINT = 129,
    VK_FORMAT_D32_SFLOAT_S8_UINT = 130,
    VK_FORMAT_BC1_RGB_UNORM_BLOCK = 131,
    VK_FORMAT_BC1_RGB_SRGB_BLOCK = 132,
    VK_FORMAT_BC1_RGBA_UNORM_BLOCK = 133,
    VK_FORMAT_BC1_RGBA_SRGB_BLOCK = 134,
    VK_FORMAT_BC2_UNORM_BLOCK = 135,
    VK_FORMAT_BC2_SRGB_BLOCK = 136,
    VK_FORMAT_BC3_UNORM_BLOCK = 137,
    VK_FORMAT_BC3_SRGB_BLOCK = 138,
    VK_FORMAT_BC4_UNORM_BLOCK = 139,
    VK_FORMAT_BC4_SNORM_BLOCK = 140,
    VK_FORMAT_BC5_UNORM_BLOCK = 141,
    VK_FORMAT_BC5_SNORM_BLOCK = 142,
    VK_FORMAT_BC6H_UFLOAT_BLOCK = 143,
    VK_FORMAT_BC6H_SFLOAT_BLOCK = 144,
    VK_FORMAT_BC7_UNORM_BLOCK = 145,
    VK_FORMAT_BC7_SRGB_BLOCK = 146,
    VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK = 147,
    VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK = 148,
    VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK = 149,
    VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK = 150,
    VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK = 151,
    VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK = 152,
    VK_FORMAT_EAC_R11_UNORM_BLOCK = 153,
    VK_FORMAT_EAC_R11_SNORM_BLOCK = 154,
    VK_FORMAT_EAC_R11G11_UNORM_BLOCK = 155,
    VK_FORMAT_EAC_R11G11_SNORM_BLOCK = 156,
    VK_FORMAT_ASTC_4x4_UNORM_BLOCK = 157,
    VK_FORMAT_ASTC_4x4_SRGB_BLOCK = 158,
    VK_FORMAT_ASTC_5x4_UNORM_BLOCK = 159,
    VK_FORMAT_ASTC_5x4_SRGB_BLOCK = 160,
    VK_FORMAT_ASTC_5x5_UNORM_BLOCK = 161,
    VK_FORMAT_ASTC_5x5_SRGB_BLOCK = 162,
    VK_FORMAT_ASTC_6x5_UNORM_BLOCK = 163,
    VK_FORMAT_ASTC_6x5_SRGB_BLOCK = 164,
    VK_FORMAT_ASTC_6x6_UNORM_BLOCK = 165,
    VK_FORMAT_ASTC_6x6_SRGB_BLOCK = 166,
    VK_FORMAT_ASTC_8x5_UNORM_BLOCK = 167,
    VK_FORMAT_ASTC_8x5_SRGB_BLOCK = 168,
    VK_FORMAT_ASTC_8x6_UNORM_BLOCK = 169,
    VK_FORMAT_ASTC_8x6_SRGB_BLOCK = 170,
    VK_FORMAT_ASTC_8x8_UNORM_BLOCK = 171,
    VK_FORMAT_ASTC_8x8_SRGB_BLOCK = 172,
    VK_FORMAT_ASTC_10x5_UNORM_BLOCK = 173,
    VK_FORMAT_ASTC_10x5_SRGB_BLOCK = 174,
    VK_FORMAT_ASTC_10x6_UNORM_BLOCK = 175,
    VK_FORMAT_ASTC_10x6_SRGB_BLOCK = 176,
    VK_FORMAT_ASTC_10x8_UNORM_BLOCK = 177,
    VK_FORMAT_ASTC_10x8_SRGB_BLOCK = 178,
    VK_FORMAT_ASTC_10x10_UNORM_BLOCK = 179,
    VK_FORMAT_ASTC_10x10_SRGB_BLOCK = 180,
    VK_FORMAT_ASTC_12x10_UNORM_BLOCK = 181,
    VK_FORMAT_ASTC_12x10_SRGB_BLOCK = 182,
    VK_FORMAT_ASTC_12x12_UNORM_BLOCK = 183,
    VK_FORMAT_ASTC_12x12_SRGB_BLOCK = 184,
} VkFormat;
```

VK_FORMAT_UNDEFINED
The format is not specified.

VK_FORMAT_R4G4_UNORM_PACK8
A two-component, 8-bit packed unsigned normalized format that has a 4-bit R component in bits 4..7, and a 4-bit G component in bits 0..3.

VK_FORMAT_R4G4B4A4_UNORM_PACK16
A four-component, 16-bit packed unsigned normalized format that has a 4-bit R component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit B component in bits 4..7, and a 4-bit A component in bits 0..3.

VK_FORMAT_B4G4R4A4_UNORM_PACK16
A four-component, 16-bit packed unsigned normalized format that has a 4-bit B component in bits 12..15, a 4-bit G component in bits 8..11, a 4-bit R component in bits 4..7, and a 4-bit A component in bits 0..3.

VK_FORMAT_R5G6B5_UNORM_PACK16
A three-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit B component in bits 0..4.

VK_FORMAT_B5G6R5_UNORM_PACK16
A three-component, 16-bit packed unsigned normalized format that has a 5-bit B component in bits 11..15, a 6-bit G component in bits 5..10, and a 5-bit R component in bits 0..4.

VK_FORMAT_R5G5B5A1_UNORM_PACK16
A four-component, 16-bit packed unsigned normalized format that has a 5-bit R component in bits 11..15, a 5-bit G component in bits 6..10, a 5-bit B component in bits 1..5, and a 1-bit A component in bit 0.

VK_FORMAT_B5G5R5A1_UNORM_PACK16
A four-component, 16-bit packed unsigned normalized format that has a 5-bit B component in bits 11..15, a 5-bit G component in bits 6..10, a 5-bit R component in bits 1..5, and a 1-bit A component in bit 0.

VK_FORMAT_A1R5G5B5_UNORM_PACK16
A four-component, 16-bit packed unsigned normalized format that has a 1-bit A component in bit 15, a 5-bit R component in bits 10..14, a 5-bit G component in bits 5..9, and a 5-bit B component in bits 0..4.

VK_FORMAT_R8_UNORM
A one-component, 8-bit unsigned normalized format that has a single 8-bit R component.

VK_FORMAT_R8_SNORM
A one-component, 8-bit signed normalized format that has a single 8-bit R component.

VK_FORMAT_R8_USCALED
A one-component, 8-bit unsigned scaled integer format that has a single 8-bit R component.

VK_FORMAT_R8_SSCALED
A one-component, 8-bit signed scaled integer format that has a single 8-bit R component.

VK_FORMAT_R8_UINT
A one-component, 8-bit unsigned integer format that has a single 8-bit R component.

VK_FORMAT_R8_SINT
A one-component, 8-bit signed integer format that has a single 8-bit R component.

VK_FORMAT_R8_SRGB
A one-component, 8-bit unsigned normalized format that has a single 8-bit R component stored with sRGB nonlinear encoding.

VK_FORMAT_R8G8_UNORM
A two-component, 16-bit unsigned normalized format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

VK_FORMAT_R8G8_SNORM
A two-component, 16-bit signed normalized format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

VK_FORMAT_R8G8_USCALED
A two-component, 16-bit unsigned scaled integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

VK_FORMAT_R8G8_SSCALED
A two-component, 16-bit signed scaled integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

VK_FORMAT_R8G8_UINT
A two-component, 16-bit unsigned integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

VK_FORMAT_R8G8_SINT
A two-component, 16-bit signed integer format that has an 8-bit R component in byte 0, and an 8-bit G component in byte 1.

VK_FORMAT_R8G8_SRGB
A two-component, 16-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, and an 8-bit G component stored with sRGB nonlinear encoding in byte 1.

VK_FORMAT_R8G8B8_UNORM
A three-component, 24-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

VK_FORMAT_R8G8B8_SNORM
A three-component, 24-bit signed normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

VK_FORMAT_R8G8B8_USCALED
A three-component, 24-bit unsigned scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

VK_FORMAT_R8G8B8_SSCALED
A three-component, 24-bit signed scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

VK_FORMAT_R8G8B8_UINT
A three-component, 24-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

VK_FORMAT_R8G8B8_SINT
A three-component, 24-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, and an 8-bit B component in byte 2.

VK_FORMAT_R8G8B8_SRGB
A three-component, 24-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, and an 8-bit B component stored with sRGB nonlinear encoding in byte 2.

VK_FORMAT_B8G8R8_UNORM
A three-component, 24-bit unsigned normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

VK_FORMAT_B8G8R8_SNORM
A three-component, 24-bit signed normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

VK_FORMAT_B8G8R8_USCALED
A three-component, 24-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

VK_FORMAT_B8G8R8_SSCALED
A three-component, 24-bit signed scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

VK_FORMAT_B8G8R8_UINT
A three-component, 24-bit unsigned integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

VK_FORMAT_B8G8R8_SINT
A three-component, 24-bit signed integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, and an 8-bit R component in byte 2.

VK_FORMAT_B8G8R8_SRGB
A three-component, 24-bit unsigned normalized format that has an 8-bit B component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, and an 8-bit R component stored with sRGB nonlinear encoding in byte 2.

VK_FORMAT_R8G8B8A8_UNORM
A four-component, 32-bit unsigned normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_R8G8B8A8_SNORM
A four-component, 32-bit signed normalized format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_R8G8B8A8_USCALED
A four-component, 32-bit unsigned scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_R8G8B8A8_SSCALED
A four-component, 32-bit signed scaled format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_R8G8B8A8_UINT
A four-component, 32-bit unsigned integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_R8G8B8A8_SINT
A four-component, 32-bit signed integer format that has an 8-bit R component in byte 0, an 8-bit G component in byte 1, an 8-bit B component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_R8G8B8A8_SRGB
A four-component, 32-bit unsigned normalized format that has an 8-bit R component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, an 8-bit B component stored with sRGB nonlinear encoding in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_B8G8R8A8_UNORM
A four-component, 32-bit unsigned normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_B8G8R8A8_SNORM
A four-component, 32-bit signed normalized format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_B8G8R8A8_USCALED
A four-component, 32-bit unsigned scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_B8G8R8A8_SSCALED
A four-component, 32-bit signed scaled format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_B8G8R8A8_UINT
A four-component, 32-bit unsigned integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_B8G8R8A8_SINT
A four-component, 32-bit signed integer format that has an 8-bit B component in byte 0, an 8-bit G component in byte 1, an 8-bit R component in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_B8G8R8A8_SRGB
A four-component, 32-bit unsigned normalized format that has an 8-bit B component stored with sRGB nonlinear encoding in byte 0, an 8-bit G component stored with sRGB nonlinear encoding in byte 1, an 8-bit R component stored with sRGB nonlinear encoding in byte 2, and an 8-bit A component in byte 3.

VK_FORMAT_A8B8G8R8_UNORM_PACK32
A four-component, 32-bit packed unsigned normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

VK_FORMAT_A8B8G8R8_SNORM_PACK32
A four-component, 32-bit packed signed normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

VK_FORMAT_A8B8G8R8_USCALED_PACK32
A four-component, 32-bit packed unsigned scaled integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

VK_FORMAT_A8B8G8R8_SSCALED_PACK32
A four-component, 32-bit packed signed scaled integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

VK_FORMAT_A8B8G8R8_UINT_PACK32
A four-component, 32-bit packed unsigned integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

VK_FORMAT_A8B8G8R8_SINT_PACK32
A four-component, 32-bit packed signed integer format that has an 8-bit A component in bits 24..31, an 8-bit B component in bits 16..23, an 8-bit G component in bits 8..15, and an 8-bit R component in bits 0..7.

VK_FORMAT_A8B8G8R8_SRGB_PACK32
A four-component, 32-bit packed unsigned normalized format that has an 8-bit A component in bits 24..31, an 8-bit B component stored with sRGB nonlinear encoding in bits 16..23, an 8-bit G component stored with sRGB nonlinear encoding in bits 8..15, and an 8-bit R component stored with sRGB nonlinear encoding in bits 0..7.

VK_FORMAT_A2R10G10B10_UNORM_PACK32
A four-component, 32-bit packed unsigned normalized format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

VK_FORMAT_A2R10G10B10_SNORM_PACK32
A four-component, 32-bit packed signed normalized format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

VK_FORMAT_A2R10G10B10_USCALED_PACK32
A four-component, 32-bit packed unsigned scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

VK_FORMAT_A2R10G10B10_SSCALED_PACK32
A four-component, 32-bit packed signed scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

VK_FORMAT_A2R10G10B10_UINT_PACK32
A four-component, 32-bit packed unsigned integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

VK_FORMAT_A2R10G10B10_SINT_PACK32
A four-component, 32-bit packed signed integer format that has a 2-bit A component in bits 30..31, a 10-bit R component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit B component in bits 0..9.

VK_FORMAT_A2B10G10R10_UNORM_PACK32
A four-component, 32-bit packed unsigned normalized format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

VK_FORMAT_A2B10G10R10_SNORM_PACK32
A four-component, 32-bit packed signed normalized format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

VK_FORMAT_A2B10G10R10_USCALED_PACK32
A four-component, 32-bit packed unsigned scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

VK_FORMAT_A2B10G10R10_SSCALED_PACK32
A four-component, 32-bit packed signed scaled integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

VK_FORMAT_A2B10G10R10_UINT_PACK32
A four-component, 32-bit packed unsigned integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

VK_FORMAT_A2B10G10R10_SINT_PACK32
A four-component, 32-bit packed signed integer format that has a 2-bit A component in bits 30..31, a 10-bit B component in bits 20..29, a 10-bit G component in bits 10..19, and a 10-bit R component in bits 0..9.

VK_FORMAT_R16_UNORM
A one-component, 16-bit unsigned normalized format that has a single 16-bit R component.

VK_FORMAT_R16_SNORM
A one-component, 16-bit signed normalized format that has a single 16-bit R component.

VK_FORMAT_R16_USCALED
A one-component, 16-bit unsigned scaled integer format that has a single 16-bit R component.

VK_FORMAT_R16_SSCALED
A one-component, 16-bit signed scaled integer format that has a single 16-bit R component.

VK_FORMAT_R16_UINT
A one-component, 16-bit unsigned integer format that has a single 16-bit R component.

VK_FORMAT_R16_SINT
A one-component, 16-bit signed integer format that has a single 16-bit R component.

VK_FORMAT_R16_SFLOAT
A one-component, 16-bit signed floating-point format that has a single 16-bit R component.

VK_FORMAT_R16G16_UNORM
A two-component, 32-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

VK_FORMAT_R16G16_SNORM
A two-component, 32-bit signed normalized format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

VK_FORMAT_R16G16_USCALED
A two-component, 32-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

VK_FORMAT_R16G16_SSCALED
A two-component, 32-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

VK_FORMAT_R16G16_UINT
A two-component, 32-bit unsigned integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

VK_FORMAT_R16G16_SINT
A two-component, 32-bit signed integer format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

VK_FORMAT_R16G16_SFLOAT
A two-component, 32-bit signed floating-point format that has a 16-bit R component in bytes 0..1, and a 16-bit G component in bytes 2..3.

VK_FORMAT_R16G16B16_UNORM
A three-component, 48-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

VK_FORMAT_R16G16B16_SNORM
A three-component, 48-bit signed normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

VK_FORMAT_R16G16B16_USCALED
A three-component, 48-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

VK_FORMAT_R16G16B16_SSCALED
A three-component, 48-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

VK_FORMAT_R16G16B16_UINT
A three-component, 48-bit unsigned integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

VK_FORMAT_R16G16B16_SINT
A three-component, 48-bit signed integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

VK_FORMAT_R16G16B16_SFLOAT
A three-component, 48-bit signed floating-point format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, and a 16-bit B component in bytes 4..5.

VK_FORMAT_R16G16B16A16_UNORM
A four-component, 64-bit unsigned normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

VK_FORMAT_R16G16B16A16_SNORM
A four-component, 64-bit signed normalized format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

VK_FORMAT_R16G16B16A16_USCALED
A four-component, 64-bit unsigned scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

VK_FORMAT_R16G16B16A16_SSCALED
A four-component, 64-bit signed scaled integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

VK_FORMAT_R16G16B16A16_UINT
A four-component, 64-bit unsigned integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

VK_FORMAT_R16G16B16A16_SINT
A four-component, 64-bit signed integer format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

VK_FORMAT_R16G16B16A16_SFLOAT
A four-component, 64-bit signed floating-point format that has a 16-bit R component in bytes 0..1, a 16-bit G component in bytes 2..3, a 16-bit B component in bytes 4..5, and a 16-bit A component in bytes 6..7.

VK_FORMAT_R32_UINT
A one-component, 32-bit unsigned integer format that has a single 32-bit R component.

VK_FORMAT_R32_SINT
A one-component, 32-bit signed integer format that has a single 32-bit R component.

VK_FORMAT_R32_SFLOAT
A one-component, 32-bit signed floating-point format that has a single 32-bit R component.

VK_FORMAT_R32G32_UINT
A two-component, 64-bit unsigned integer format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

VK_FORMAT_R32G32_SINT
A two-component, 64-bit signed integer format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

VK_FORMAT_R32G32_SFLOAT
A two-component, 64-bit signed floating-point format that has a 32-bit R component in bytes 0..3, and a 32-bit G component in bytes 4..7.

VK_FORMAT_R32G32B32_UINT
A three-component, 96-bit unsigned integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.

VK_FORMAT_R32G32B32_SINT
A three-component, 96-bit signed integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.

VK_FORMAT_R32G32B32_SFLOAT
A three-component, 96-bit signed floating-point format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, and a 32-bit B component in bytes 8..11.

VK_FORMAT_R32G32B32A32_UINT
A four-component, 128-bit unsigned integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

VK_FORMAT_R32G32B32A32_SINT
A four-component, 128-bit signed integer format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

VK_FORMAT_R32G32B32A32_SFLOAT
A four-component, 128-bit signed floating-point format that has a 32-bit R component in bytes 0..3, a 32-bit G component in bytes 4..7, a 32-bit B component in bytes 8..11, and a 32-bit A component in bytes 12..15.

VK_FORMAT_R64_UINT
A one-component, 64-bit unsigned integer format that has a single 64-bit R component.

VK_FORMAT_R64_SINT
A one-component, 64-bit signed integer format that has a single 64-bit R component.

VK_FORMAT_R64_SFLOAT
A one-component, 64-bit signed floating-point format that has a single 64-bit R component.

VK_FORMAT_R64G64_UINT
A two-component, 128-bit unsigned integer format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

VK_FORMAT_R64G64_SINT
A two-component, 128-bit signed integer format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

VK_FORMAT_R64G64_SFLOAT
A two-component, 128-bit signed floating-point format that has a 64-bit R component in bytes 0..7, and a 64-bit G component in bytes 8..15.

VK_FORMAT_R64G64B64_UINT
A three-component, 192-bit unsigned integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

VK_FORMAT_R64G64B64_SINT
A three-component, 192-bit signed integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

VK_FORMAT_R64G64B64_SFLOAT
A three-component, 192-bit signed floating-point format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, and a 64-bit B component in bytes 16..23.

VK_FORMAT_R64G64B64A64_UINT
A four-component, 256-bit unsigned integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.

VK_FORMAT_R64G64B64A64_SINT
A four-component, 256-bit signed integer format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.

VK_FORMAT_R64G64B64A64_SFLOAT
A four-component, 256-bit signed floating-point format that has a 64-bit R component in bytes 0..7, a 64-bit G component in bytes 8..15, a 64-bit B component in bytes 16..23, and a 64-bit A component in bytes 24..31.

VK_FORMAT_B10G11R11_UFLOAT_PACK32
A three-component, 32-bit packed unsigned floating-point format that has a 10-bit B component in bits 22..31, an 11-bit G component in bytes 11..21, an 11-bit R component in bytes 0..10. See Section 2.6.4 and Section 2.6.3.

VK_FORMAT_E5B9G9R9_UFLOAT_PACK32
A three-component, 32-bit packed unsigned floating-point format that has a 5-bit shared exponent in bits 27..31, a 9-bit B component mantissa in bits 18..26, a 9-bit G component mantissa in bits 9..17, and a 9-bit R component mantissa in bits 0..8.

VK_FORMAT_D16_UNORM
A one-component, 16-bit unsigned normalized format that has a single 16-bit depth component.

VK_FORMAT_X8_D24_UNORM_PACK32
A two-component, 32-bit format that has 24 unsigned normalized bits in the depth component and, optionally, 8 bits that are unused.

VK_FORMAT_D32_SFLOAT
A one-component, 32-bit signed floating-point format that has 32-bits in the depth component.

VK_FORMAT_S8_UINT
A one-component, 8-bit unsigned integer format that has 8-bits in the stencil component.

VK_FORMAT_D16_UNORM_S8_UINT
A two-component, 24-bit format that has 16 unsigned normalized bits in the depth component and 8 unsigned integer bits in the stencil component.

VK_FORMAT_D24_UNORM_S8_UINT
A two-component, 32-bit packed format that has 8 unsigned integer bits in the stencil component, and 24 unsigned normalized bits in the depth component.

VK_FORMAT_D32_SFLOAT_S8_UINT
A two-component format that has 32 signed float bits in the depth component and 8 unsigned integer bits in the stencil component. There are optionally 24-bits that are unused.

VK_FORMAT_BC1_RGB_UNORM_BLOCK
A three-component, block compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized RGB image data. This format has no alpha and is considered opaque.

VK_FORMAT_BC1_RGB_SRGB_BLOCK
A three-component, block compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized RGB image data with sRGB nonlinear encoding. This format has no alpha and is considered opaque.

VK_FORMAT_BC1_RGBA_UNORM_BLOCK
A four-component, block compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized RGB image data, and provides 1 bit of alpha.

VK_FORMAT_BC1_RGBA_SRGB_BLOCK
A four-component, block compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized RGB image data with sRGB nonlinear encoding, and provides 1 bit of alpha.

VK_FORMAT_BC2_UNORM_BLOCK
A four-component, block compressed format where each 4x4 block consists of 64-bits of unsigned normalized alpha image data followed by 64-bits of encoded unsigned normalized RGB image data.

VK_FORMAT_BC2_SRGB_BLOCK
A four-component, block compressed format where each 4x4 block consists of 64-bits of unsigned normalized alpha image data followed by 64-bits of encoded unsigned normalized RGB image data with sRGB nonlinear encoding.

VK_FORMAT_BC3_UNORM_BLOCK
A four-component, block compressed format where each 4x4 block consists of 64-bits of encoded alpha image data followed by 64-bits of encoded RGB image data. Both blocks are decoded as unsigned normalized values.

VK_FORMAT_BC3_SRGB_BLOCK
A four-component, block compressed format where each 4x4 block consists of 64-bits of encoded alpha image data followed by 64-bits of encoded RGB image data with sRGB nonlinear encoding. Both blocks are decoded as unsigned normalized values.

VK_FORMAT_BC4_UNORM_BLOCK
A one-component, block compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized red image data.

VK_FORMAT_BC4_SNORM_BLOCK
A one-component, block compressed format where each 4x4 block consists of 64-bits of encoded signed normalized red image data.

VK_FORMAT_BC5_UNORM_BLOCK
A two-component, block compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized red image data followed by 64-bits of encoded unsigned normalized green image data.

VK_FORMAT_BC5_SNORM_BLOCK
A two-component, block compressed format where each 4x4 block consists of 64-bits of encoded signed normalized red image data followed by 64-bits of encoded signed normalized green image data.

VK_FORMAT_BC6H_UFLOAT_BLOCK
A three-component, block compressed format where each 4x4 block consists of 128-bits of encoded unsigned floating-point RGB image data.

VK_FORMAT_BC6H_SFLOAT_BLOCK
A three-component, block compressed format where each 4x4 block consists of 128-bits of encoded signed floating-point RGB image data.

VK_FORMAT_BC7_UNORM_BLOCK
A four-component, block compressed format where each 4x4 block consists of 128-bits of encoded unsigned normalized RGBA image data.

VK_FORMAT_BC7_SRGB_BLOCK
A four-component, block compressed format where each 4x4 block consists of 128-bits of encoded unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK
A three-component, ETC2 compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized RGB image data.

VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK
A three-component, ETC2 compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized RGB image data with sRGB nonlinear encoding.

VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK
A four-component, ETC2 compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized RGB image data, and provides 1 bit of alpha.

VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK
A four-component, ETC2 compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized RGB image data with sRGB nonlinear encoding, and provides 1 bit of alpha.

VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK
A four-component, ETC2 compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized RGB image data, and 64-bits of encoded unsigned normalized alpha image data.

VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK
A four-component, ETC2 compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized RGB image data with sRGB nonlinear encoding, and 64-bits of encoded unsigned normalized alpha image data.

VK_FORMAT_EAC_R11_UNORM_BLOCK
A one-component, ETC2 compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized red image data.

VK_FORMAT_EAC_R11_SNORM_BLOCK
A one-component, ETC2 compressed format where each 4x4 block consists of 64-bits of encoded signed normalized red image data.

VK_FORMAT_EAC_R11G11_UNORM_BLOCK
A two-component, ETC2 compressed format where each 4x4 block consists of 64-bits of encoded unsigned normalized red image data followed by 64-bits of encoded unsigned normalized green image data.

VK_FORMAT_EAC_R11G11_SNORM_BLOCK
A two-component, ETC2 compressed format where each 4x4 block consists of 64-bits of encoded signed normalized red image data followed by 64-bits of encoded signed normalized green image data.

VK_FORMAT_ASTC_4x4_UNORM_BLOCK
A four-component, ASTC compressed format where each 4x4 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_4x4_SRGB_BLOCK
A four-component, ASTC compressed format where each 4x4 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_5x4_UNORM_BLOCK
: A four-component, ASTC compressed format where each 5x4 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_5x4_SRGB_BLOCK
: A four-component, ASTC compressed format where each 5x4 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_5x5_UNORM_BLOCK
: A four-component, ASTC compressed format where each 5x5 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_5x5_SRGB_BLOCK
: A four-component, ASTC compressed format where each 5x5 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_6x5_UNORM_BLOCK
: A four-component, ASTC compressed format where each 6x5 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_6x5_SRGB_BLOCK
: A four-component, ASTC compressed format where each 6x5 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_6x6_UNORM_BLOCK
: A four-component, ASTC compressed format where each 6x6 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_6x6_SRGB_BLOCK
: A four-component, ASTC compressed format where each 6x6 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_8x5_UNORM_BLOCK
: A four-component, ASTC compressed format where each 8x5 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_8x5_SRGB_BLOCK
: A four-component, ASTC compressed format where each 8x5 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_8x6_UNORM_BLOCK
: A four-component, ASTC compressed format where each 8x6 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_8x6_SRGB_BLOCK
: A four-component, ASTC compressed format where each 8x6 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_8x8_UNORM_BLOCK
: A four-component, ASTC compressed format where each 8x8 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_8x8_SRGB_BLOCK
: A four-component, ASTC compressed format where each 8x8 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_10x5_UNORM_BLOCK
A four-component, ASTC compressed format where each 10x5 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_10x5_SRGB_BLOCK
A four-component, ASTC compressed format where each 10x5 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_10x6_UNORM_BLOCK
A four-component, ASTC compressed format where each 10x6 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_10x6_SRGB_BLOCK
A four-component, ASTC compressed format where each 10x6 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_10x8_UNORM_BLOCK
A four-component, ASTC compressed format where each 10x8 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_10x8_SRGB_BLOCK
A four-component, ASTC compressed format where each 10x8 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_10x10_UNORM_BLOCK
A four-component, ASTC compressed format where each 10x10 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_10x10_SRGB_BLOCK
A four-component, ASTC compressed format where each 10x10 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_12x10_UNORM_BLOCK
A four-component, ASTC compressed format where each 12x10 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_12x10_SRGB_BLOCK
A four-component, ASTC compressed format where each 12x10 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

VK_FORMAT_ASTC_12x12_UNORM_BLOCK
A four-component, ASTC compressed format where each 12x12 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data.

VK_FORMAT_ASTC_12x12_SRGB_BLOCK
A four-component, ASTC compressed format where each 12x12 block consists of 128-bits of encoded image data which is decoded as unsigned normalized RGBA image data with sRGB nonlinear encoding.

29.3.1.1 Packed Formats

For the purposes of address alignment when referencing buffer memory containing vertex attribute or texel data, the following formats are considered *packed* - whole texels or attributes are stored in a single data element, rather than individual components occupying a single data element:

- Packed into 8-bit data types:

  - VK_FORMAT_R4G4_UNORM_PACK8
- Packed into 16-bit data types:
  - VK_FORMAT_R4G4B4A4_UNORM_PACK16
  - VK_FORMAT_B4G4R4A4_UNORM_PACK16
  - VK_FORMAT_R5G6B5_UNORM_PACK16
  - VK_FORMAT_B5G6R5_UNORM_PACK16
  - VK_FORMAT_R5G5B5A1_UNORM_PACK16
  - VK_FORMAT_B5G5R5A1_UNORM_PACK16
  - VK_FORMAT_A1R5G5B5_UNORM_PACK16
- Packed into 32-bit data types:
  - VK_FORMAT_A8B8G8R8_UNORM_PACK32
  - VK_FORMAT_A8B8G8R8_SNORM_PACK32
  - VK_FORMAT_A8B8G8R8_USCALED_PACK32
  - VK_FORMAT_A8B8G8R8_SSCALED_PACK32
  - VK_FORMAT_A8B8G8R8_UINT_PACK32
  - VK_FORMAT_A8B8G8R8_SINT_PACK32
  - VK_FORMAT_A8B8G8R8_SRGB_PACK32
  - VK_FORMAT_A2R10G10B10_UNORM_PACK32
  - VK_FORMAT_A2R10G10B10_SNORM_PACK32
  - VK_FORMAT_A2R10G10B10_USCALED_PACK32
  - VK_FORMAT_A2R10G10B10_SSCALED_PACK32
  - VK_FORMAT_A2R10G10B10_UINT_PACK32
  - VK_FORMAT_A2R10G10B10_SINT_PACK32
  - VK_FORMAT_A2B10G10R10_UNORM_PACK32
  - VK_FORMAT_A2B10G10R10_SNORM_PACK32
  - VK_FORMAT_A2B10G10R10_USCALED_PACK32
  - VK_FORMAT_A2B10G10R10_SSCALED_PACK32
  - VK_FORMAT_A2B10G10R10_UINT_PACK32
  - VK_FORMAT_A2B10G10R10_SINT_PACK32
  - VK_FORMAT_B10G11R11_UFLOAT_PACK32
  - VK_FORMAT_E5B9G9R9_UFLOAT_PACK32
  - VK_FORMAT_X8_D24_UNORM_PACK32

29.3.1.2 Identification of formats

A *format* is represented by a single enum value. The name of a format is usually built up by using the following pattern:

`VK_FORMAT_{channel-format|compression-scheme}_{numeric-format}`

The channel-format specifies either the size of the R, G, B, and A components (if they are present) in the case of a color format, or the size of the depth (D) and stencil (S) components (if they are present) in the case of a depth/stencil format (see below). An X indicates a component that is unused, but may be present for padding.

Table 29.3: Interpretation of Numeric Format

Numeric format	Description
UNORM	The components are unsigned normalized values in the range [0,1]
SNORM	The components are signed normalized values in the range [-1,1]
USCALED	The components are unsigned integer values that get converted to floating-point in the range $[0,2^{n}-1]$
SSCALED	The components are signed integer values that get converted to floating-point in the range $[-2^{n-1},2^{n-1}-1]$
UINT	The components are unsigned integer values in the range $[0,2^{n}-1]$
SINT	The components are signed integer values in the range $[-2^{n-1},2^{n-1}-1]$
UFLOAT	The components are unsigned floating-point numbers (used by packed, shared exponent, and some compressed formats)
SFLOAT	The components are signed floating-point numbers
SRGB	The R, G, and B components are unsigned normalized values that represent values using sRGB nonlinear encoding, while the A component (if one exists) is a regular unsigned normalized value

The suffix _PACKnn indicates that the format is packed into an underlying type with nn bits.

The suffix _BLOCK indicates that the format is a block compressed format, with the representation of multiple pixels encoded interdependently within a region.

Table 29.4: Interpretation of Compression Scheme

Compression scheme	Description
BC	Block Compression. See Section B.1.
ETC2	Ericsson Texture Compression. See Section B.2.
EAC	ETC2 Alpha Compression. See Section B.2.
ASTC	Adaptive Scalable Texture Compression (LDR Profile). See Section B.3.

29.3.1.3 Representation

Color formats must be represented in memory in exactly the form indicated by the format's name. This means that promoting one format to another with more bits per component and/or additional channels must not occur for color formats. Depth and stencil formats have more relaxed requirements as discussed below.

The representation of non-packed formats is that the first component specified in the name of the format is in the lowest memory addresses and the last component specified is in the highest memory addresses. See Byte mappings for non-packed/compressed color formats. The ordering of bytes within a component is determined by the endianness of the platform.

Table 29.5: Byte mappings for non-packed/compressed color formats

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**← Byte**
R																**VK_FORMAT_R8_***
R	G															**VK_FORMAT_R8G8_***
R	G	B														**VK_FORMAT_R8G8B8_***
B	G	R														**VK_FORMAT_B8G8R8_***
R	G	B	A													**VK_FORMAT_R8G8B8A8_***
B	G	R	A													**VK_FORMAT_B8G8R8A8_***
R																**VK_FORMAT_R16_***
R		G														**VK_FORMAT_R16G16_***
R		G		B												**VK_FORMAT_R16G16B16_***
R		G		B		A										**VK_FORMAT_R16G16B16A16_***
R																**VK_FORMAT_R32_***
R				G												**VK_FORMAT_R32G32_***
R				G				B								**VK_FORMAT_R32G32B32_***
R				G				B				A				**VK_FORMAT_R32G32B32A32_***
R																**VK_FORMAT_R64_***
R								G								**VK_FORMAT_R64G64_***
VK_FORMAT_R64G64B64_* as VK_FORMAT_R64G64_* but with B in bytes 16-23																
VK_FORMAT_R64G64B64A64_* as VK_FORMAT_R64G64B64_* but with A in bytes 24-31																

Packed formats store multiple channels within one underlying type. The bit representation is that the first component specified in the name of the format is in the most-significant bits and the last component specified is in the least-significant bits of the underlying type. The in-memory ordering of bytes comprising the underlying type is determined by the endianness of the platform.

Table 29.6: Bit mappings for packed 8-bit formats

Bit →	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
VK_FORMAT_R4G4_UNORM_PACK8	R_3	R_2	R_1	R_0	G_3	G_2	G_1	G_0

Table 29.7: Bit mappings for packed 16-bit VK_FORMAT_* formats

Bit →	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
R4G4B4A4_UNORM_PACK16	R_3	R_2	R_1	R_0	G_3	G_2	G_1	G_0	B_3	B_2	B_1	B_0	A_3	A_2	A_1	A_0
B4G4R4A4_UNORM_PACK16	B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0	R_3	R_2	R_1	R_0	A_3	A_2	A_1	A_0
R5G6B5_UNORM_PACK16	R_4	R_3	R_2	R_1	R_0	G_5	G_4	G_3	G_2	G_1	G_0	B_4	B_3	B_2	B_1	B_0
B5G6R5_UNORM_PACK16	B_4	B_3	B_2	B_1	B_0	G_5	G_4	G_3	G_2	G_1	G_0	R_4	R_3	R_2	R_1	R_0
R5G5B5A1_UNORM_PACK16	R_4	R_3	R_2	R_1	R_0	G_4	G_3	G_2	G_1	G_0	B_4	B_3	B_2	B_1	B_0	A_0
B5G5R5A1_UNORM_PACK16	B_4	B_3	B_2	B_1	B_0	G_4	G_3	G_2	G_1	G_0	R_4	R_3	R_2	R_1	R_0	A_0
A1R5G5B5_UNORM_PACK16	A_0	R_4	R_3	R_2	R_1	R_0	G_4	G_3	G_2	G_1	G_0	B_4	B_3	B_2	B_1	B_0

Table 29.8: Bit mappings for packed 32-bit formats

31	30	29	28	27	26	25	25	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VK_FORMAT_A8B8G8R8_*_PACK32																															
A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
VK_FORMAT_A2R10G10B10_*_PACK32																															
A_1	A_0	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0	G_9	G_8	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0	B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
VK_FORMAT_A2B10G10R10_*_PACK32																															
A_1	A_0	B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	G_9	G_8	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
VK_FORMAT_B10G11R11_UFLOAT_PACK32																															
B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	G_{10}	G_9	G_8	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0	R_{10}	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
VK_FORMAT_E5B9G9R9_UFLOAT_PACK32																															
E_4	E_3	E_2	E_1	E_0	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0	G_8	G_7	G_6	G_5	G_4	G_3	G_2	G_1	G_0	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
VK_FORMAT_XB_D24_UNORM_PACK32																															
X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0	D_{23}	D_{22}	D_{21}	D_{20}	D_{19}	D_{18}	D_{17}	D_{16}	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

29.3.1.4 Depth and Stencil Formats

Depth and stencil formats are considered opaque and need not be stored in the exact number of bits per texel or component ordering indicated by the format enum. However, implementations must not substitute a different depth or stencil precision than that described in the format (e.g. D16 must not be implemented as D24 or D32).

29.3.1.5 Format Compatibility Classes

Uncompressed color formats are *compatible* with each other if they occupy the same number of bits per data element. Compressed color formats are compatible with each other if the only difference between them is the numerical type of the uncompressed pixels (e.g. signed vs. unsigned, or SRGB vs. UNORM encoding). Each depth/stencil format is only compatible with itself. In the following table, all the formats in the same row are compatible.

Table 29.9: Compatible formats

Class	Formats
8-bit	VK_FORMAT_R4G4_UNORM_PACK8, VK_FORMAT_R8_UNORM, VK_FORMAT_R8_SNORM, VK_FORMAT_R8_USCALED, VK_FORMAT_R8_SSCALED, VK_FORMAT_R8_UINT, VK_FORMAT_R8_SINT, VK_FORMAT_R8_SRGB

Table 29.9: (continued)

Class	Formats
16-bit	VK_FORMAT_R4G4B4A4_UNORM_PACK16, VK_FORMAT_B4G4R4A4_UNORM_PACK16, VK_FORMAT_R5G6B5_UNORM_PACK16, VK_FORMAT_B5G6R5_UNORM_PACK16, VK_FORMAT_R5G5B5A1_UNORM_PACK16, VK_FORMAT_B5G5R5A1_UNORM_PACK16, VK_FORMAT_A1R5G5B5_UNORM_PACK16 VK_FORMAT_R8G8_UNORM, VK_FORMAT_R8G8_SNORM, VK_FORMAT_R8G8_USCALED, VK_FORMAT_R8G8_SSCALED, VK_FORMAT_R8G8_UINT, VK_FORMAT_R8G8_SINT, VK_FORMAT_R8G8_SRGB VK_FORMAT_R16_UNORM, VK_FORMAT_R16_SNORM, VK_FORMAT_R16_USCALED, VK_FORMAT_R16_SSCALED, VK_FORMAT_R16_UINT, VK_FORMAT_R16_SINT, VK_FORMAT_R16_SFLOAT
24-bit	VK_FORMAT_R8G8B8_UNORM, VK_FORMAT_R8G8B8_SNORM, VK_FORMAT_R8G8B8_USCALED, VK_FORMAT_R8G8B8_SSCALED, VK_FORMAT_R8G8B8_UINT, VK_FORMAT_R8G8B8_SINT, VK_FORMAT_R8G8B8_SRGB, VK_FORMAT_B8G8R8_UNORM, VK_FORMAT_B8G8R8_SNORM, VK_FORMAT_B8G8R8_USCALED, VK_FORMAT_B8G8R8_SSCALED, VK_FORMAT_B8G8R8_UINT, VK_FORMAT_B8G8R8_SINT, VK_FORMAT_B8G8R8_SRGB

Table 29.9: (continued)

Class	Formats
32-bit	VK_FORMAT_R8G8B8A8_UNORM, VK_FORMAT_R8G8B8A8_SNORM, VK_FORMAT_R8G8B8A8_USCALED, VK_FORMAT_R8G8B8A8_SSCALED, VK_FORMAT_R8G8B8A8_UINT, VK_FORMAT_R8G8B8A8_SINT, VK_FORMAT_R8G8B8A8_SRGB, VK_FORMAT_B8G8R8A8_UNORM, VK_FORMAT_B8G8R8A8_SNORM, VK_FORMAT_B8G8R8A8_USCALED, VK_FORMAT_B8G8R8A8_SSCALED, VK_FORMAT_B8G8R8A8_UINT, VK_FORMAT_B8G8R8A8_SINT, VK_FORMAT_B8G8R8A8_SRGB, VK_FORMAT_A8B8G8R8_UNORM_PACK32, VK_FORMAT_A8B8G8R8_SNORM_PACK32, VK_FORMAT_A8B8G8R8_USCALED_PACK32, VK_FORMAT_A8B8G8R8_SSCALED_PACK32, VK_FORMAT_A8B8G8R8_UINT_PACK32, VK_FORMAT_A8B8G8R8_SINT_PACK32, VK_FORMAT_A8B8G8R8_SRGB_PACK32 VK_FORMAT_A2R10G10B10_UNORM_PACK32, VK_FORMAT_A2R10G10B10_SNORM_PACK32, VK_FORMAT_A2R10G10B10_USCALED_PACK32, VK_FORMAT_A2R10G10B10_SSCALED_PACK32, VK_FORMAT_A2R10G10B10_UINT_PACK32, VK_FORMAT_A2R10G10B10_SINT_PACK32, VK_FORMAT_A2B10G10R10_UNORM_PACK32, VK_FORMAT_A2B10G10R10_SNORM_PACK32, VK_FORMAT_A2B10G10R10_USCALED_PACK32, VK_FORMAT_A2B10G10R10_SSCALED_PACK32, VK_FORMAT_A2B10G10R10_UINT_PACK32, VK_FORMAT_A2B10G10R10_SINT_PACK32, VK_FORMAT_R16G16_UNORM, VK_FORMAT_R16G16_SNORM, VK_FORMAT_R16G16_USCALED, VK_FORMAT_R16G16_SSCALED, VK_FORMAT_R16G16_UINT, VK_FORMAT_R16G16_SINT, VK_FORMAT_R16G16_SFLOAT, VK_FORMAT_R32_UINT, VK_FORMAT_R32_SINT, VK_FORMAT_R32_SFLOAT, VK_FORMAT_B10G11R11_UFLOAT_PACK32, VK_FORMAT_E5B9G9R9_UFLOAT_PACK32

Table 29.9: (continued)

Class	Formats
48-bit	VK_FORMAT_R16G16B16_UNORM, VK_FORMAT_R16G16B16_SNORM, VK_FORMAT_R16G16B16_USCALED, VK_FORMAT_R16G16B16_SSCALED, VK_FORMAT_R16G16B16_UINT, VK_FORMAT_R16G16B16_SINT, VK_FORMAT_R16G16B16_SFLOAT
64-bit	VK_FORMAT_R16G16B16A16_UNORM, VK_FORMAT_R16G16B16A16_SNORM, VK_FORMAT_R16G16B16A16_USCALED, VK_FORMAT_R16G16B16A16_SSCALED, VK_FORMAT_R16G16B16A16_UINT, VK_FORMAT_R16G16B16A16_SINT, VK_FORMAT_R16G16B16A16_SFLOAT, VK_FORMAT_R32G32_UINT, VK_FORMAT_R32G32_SINT, VK_FORMAT_R32G32_SFLOAT, VK_FORMAT_R64_UINT, VK_FORMAT_R64_SINT, VK_FORMAT_R64_SFLOAT,
96-bit	VK_FORMAT_R32G32B32_UINT, VK_FORMAT_R32G32B32_SINT, VK_FORMAT_R32G32B32_SFLOAT
128-bit	VK_FORMAT_R32G32B32A32_UINT, VK_FORMAT_R32G32B32A32_SINT, VK_FORMAT_R32G32B32A32_SFLOAT, VK_FORMAT_R64G64_UINT, VK_FORMAT_R64G64_SINT, VK_FORMAT_R64G64_SFLOAT
192-bit	VK_FORMAT_R64G64B64_UINT, VK_FORMAT_R64G64B64_SINT, VK_FORMAT_R64G64B64_SFLOAT
256-bit	VK_FORMAT_R64G64B64A64_UINT, VK_FORMAT_R64G64B64A64_SINT, VK_FORMAT_R64G64B64A64_SFLOAT
BC1_RGB	VK_FORMAT_BC1_RGB_UNORM_BLOCK, VK_FORMAT_BC1_RGB_SRGB_BLOCK
BC1_RGBA	VK_FORMAT_BC1_RGBA_UNORM_BLOCK, VK_FORMAT_BC1_RGBA_SRGB_BLOCK
BC2	VK_FORMAT_BC2_UNORM_BLOCK, VK_FORMAT_BC2_SRGB_BLOCK
BC3	VK_FORMAT_BC3_UNORM_BLOCK, VK_FORMAT_BC3_SRGB_BLOCK
BC4	VK_FORMAT_BC4_UNORM_BLOCK, VK_FORMAT_BC4_SNORM_BLOCK
BC5	VK_FORMAT_BC5_UNORM_BLOCK, VK_FORMAT_BC5_SNORM_BLOCK
BC6H	VK_FORMAT_BC6H_UFLOAT_BLOCK, VK_FORMAT_BC6H_SFLOAT_BLOCK

Table 29.9: (continued)

Class	**Formats**
BC7	VK_FORMAT_BC7_UNORM_BLOCK, VK_FORMAT_BC7_SRGB_BLOCK
ETC2_RGB	VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK
ETC2_RGBA	VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK
ETC2_EAC_RGBA	VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK, VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK
EAC_R	VK_FORMAT_EAC_R11_UNORM_BLOCK, VK_FORMAT_EAC_R11_SNORM_BLOCK
EAC_RG	VK_FORMAT_EAC_R11G11_UNORM_BLOCK, VK_FORMAT_EAC_R11G11_SNORM_BLOCK
ASTC_4x4	VK_FORMAT_ASTC_4x4_UNORM_BLOCK, VK_FORMAT_ASTC_4x4_SRGB_BLOCK
ASTC_5x4	VK_FORMAT_ASTC_5x4_UNORM_BLOCK, VK_FORMAT_ASTC_5x4_SRGB_BLOCK
ASTC_5x5	VK_FORMAT_ASTC_5x5_UNORM_BLOCK, VK_FORMAT_ASTC_5x5_SRGB_BLOCK
ASTC_6x5	VK_FORMAT_ASTC_6x5_UNORM_BLOCK, VK_FORMAT_ASTC_6x5_SRGB_BLOCK
ASTC_6x6	VK_FORMAT_ASTC_6x6_UNORM_BLOCK, VK_FORMAT_ASTC_6x6_SRGB_BLOCK
ASTC_8x5	VK_FORMAT_ASTC_8x5_UNORM_BLOCK, VK_FORMAT_ASTC_8x5_SRGB_BLOCK
ASTC_8x6	VK_FORMAT_ASTC_8x6_UNORM_BLOCK, VK_FORMAT_ASTC_8x6_SRGB_BLOCK
ASTC_8x8	VK_FORMAT_ASTC_8x8_UNORM_BLOCK, VK_FORMAT_ASTC_8x8_SRGB_BLOCK
ASTC_10x5	VK_FORMAT_ASTC_10x5_UNORM_BLOCK, VK_FORMAT_ASTC_10x5_SRGB_BLOCK
ASTC_10x6	VK_FORMAT_ASTC_10x6_UNORM_BLOCK, VK_FORMAT_ASTC_10x6_SRGB_BLOCK
ASTC_10x8	VK_FORMAT_ASTC_10x8_UNORM_BLOCK, VK_FORMAT_ASTC_10x8_SRGB_BLOCK
ASTC_10x10	VK_FORMAT_ASTC_10x10_UNORM_BLOCK, VK_FORMAT_ASTC_10x10_SRGB_BLOCK
ASTC_12x10	VK_FORMAT_ASTC_12x10_UNORM_BLOCK, VK_FORMAT_ASTC_12x10_SRGB_BLOCK
ASTC_12x12	VK_FORMAT_ASTC_12x12_UNORM_BLOCK, VK_FORMAT_ASTC_12x12_SRGB_BLOCK
D16	VK_FORMAT_D16_UNORM
D24	VK_FORMAT_X8_D24_UNORM_PACK32
D32	VK_FORMAT_D32_SFLOAT
S8	VK_FORMAT_S8_UINT
D16S8	VK_FORMAT_D16_UNORM_S8_UINT

Table 29.9: (continued)

Class	Formats
D24S8	VK_FORMAT_D24_UNORM_S8_UINT
D32S8	VK_FORMAT_D32_SFLOAT_S8_UINT

29.3.2 Format Properties

Supported format features are properties of the physical device, and are queried with the command:

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

physicalDevice is the physical device from which to query the format properties. *format* selects the format for which the properties are to be queried. *pFormatProperties* is a pointer to a VkFormatProperties structure to be filled in.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *format* must be a valid `VkFormat` value
- *pFormatProperties* must be a pointer to a VkFormatProperties structure

vkGetPhysicalDeviceFormatProperties returns VkFormatProperties:

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

The features are described as a set of `VkFormatFeatureFlagBits`:

```
typedef enum VkFormatFeatureFlagBits {
    VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT = 0x00000001,
    VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT = 0x00000002,
    VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT = 0x00000004,
    VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT = 0x00000008,
    VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT = 0x00000010,
    VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT = 0x00000020,
    VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT = 0x00000040,
    VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT = 0x00000080,
    VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT = 0x00000100,
    VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000200,
    VK_FORMAT_FEATURE_BLIT_SRC_BIT = 0x00000400,
    VK_FORMAT_FEATURE_BLIT_DST_BIT = 0x00000800,
} VkFormatFeatureFlagBits;
```

The *linearTilingFeatures* and *optimalTilingFeatures* members of the VkFormatProperties structure describe what features are supported by VK_IMAGE_TILING_LINEAR and VK_IMAGE_TILING_OPTIMAL images, respectively.

The following features may be supported by images or image views created with *format*:

VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT
VkImageView can be sampled from. See sampled images section.

VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT
VkImageView can be used as storage image. See storage images section.

VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT
VkImageView can be used as storage image that supports atomic operations.

VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT
VkImageView can be used as a framebuffer color attachment and as an input attachment.

VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT
VkImageView can be used as a framebuffer color attachment that supports blending and as an input attachment.

VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT
VkImageView can be used as a framebuffer depth/stencil attachment and as an input attachment.

VK_FORMAT_FEATURE_BLIT_SRC_BIT
VkImage can be used as *srcImage* for the **vkCmdBlitImage** command.

VK_FORMAT_FEATURE_BLIT_DST_BIT
VkImage can be used as *dstImage* for the **vkCmdBlitImage** command.

The *bufferFeatures* member of the VkFormatProperties structure describes what features are supported by buffers.

The following features may be supported by buffers or buffer views created with *format*:

VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT
Format can be used to create a VkBufferView that can be bound to a VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER descriptor.

VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT
Format can be used to create a VkBufferView that can be bound to a VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER descriptor.

VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT
Atomic operations are supported on VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER with this format.

VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT
Format can be used as a vertex attribute format (VkVertexInputAttributeDescription.format).

If *format* is a block-compression format, then buffers must not support any features for the format.

29.3.3 Required Format Support

Implementations must support at least the following set of features on the listed formats. These features are supported on existing formats without needing to advertise an extension or needing to explicitly enable them. Support for additional format features is queried using the `vkGetPhysicalDeviceFormatProperties` command.

The VK_FORMAT_FEATURE_SAMPLED_IMAGE_BIT and VK_FORMAT_FEATURE_BLIT_SRC_BIT features must be supported in *`optimalTilingFeatures`* for the following formats:

- VK_FORMAT_B4G4R4A4_UNORM_PACK16
- VK_FORMAT_R5G5B5A1_UNORM_PACK16
- VK_FORMAT_B5G5R5A1_UNORM_PACK16
- VK_FORMAT_B5G6R5_UNORM_PACK16
- VK_FORMAT_R5G6B5_UNORM_PACK16
- VK_FORMAT_R8_UNORM
- VK_FORMAT_R8_SNORM
- VK_FORMAT_R8_UINT
- VK_FORMAT_R8_SINT
- VK_FORMAT_R8G8_UNORM
- VK_FORMAT_R8G8_SNORM
- VK_FORMAT_R8G8_UINT
- VK_FORMAT_R8G8_SINT
- VK_FORMAT_R8G8B8A8_UNORM
- VK_FORMAT_R8G8B8A8_SNORM
- VK_FORMAT_R8G8B8A8_UINT
- VK_FORMAT_R8G8B8A8_SINT
- VK_FORMAT_R8G8B8A8_SRGB
- VK_FORMAT_A8B8G8R8_UNORM_PACK32
- VK_FORMAT_A8B8G8R8_SNORM_PACK32
- VK_FORMAT_A8B8G8R8_UINT_PACK32
- VK_FORMAT_A8B8G8R8_SINT_PACK32
- VK_FORMAT_A8B8G8R8_SRGB_PACK32
- VK_FORMAT_B8G8R8A8_UNORM
- VK_FORMAT_B8G8R8A8_SRGB
- VK_FORMAT_A2R10G10B10_UNORM_PACK32
- VK_FORMAT_A2R10G10B10_UINT_PACK32

- VK_FORMAT_A2B10G10R10_UNORM_PACK32
- VK_FORMAT_R16_UNORM
- VK_FORMAT_R16_SNORM
- VK_FORMAT_R16_UINT
- VK_FORMAT_R16_SINT
- VK_FORMAT_R16_SFLOAT
- VK_FORMAT_R16G16_UNORM
- VK_FORMAT_R16G16_SNORM
- VK_FORMAT_R16G16_UINT
- VK_FORMAT_R16G16_SINT
- VK_FORMAT_R16G16_SFLOAT
- VK_FORMAT_R16G16B16A16_UNORM
- VK_FORMAT_R16G16B16A16_SNORM
- VK_FORMAT_R16G16B16A16_UINT
- VK_FORMAT_R16G16B16A16_SINT
- VK_FORMAT_R16G16B16A16_SFLOAT
- VK_FORMAT_R32_UINT
- VK_FORMAT_R32_SINT
- VK_FORMAT_R32_SFLOAT
- VK_FORMAT_R32G32_UINT
- VK_FORMAT_R32G32_SINT
- VK_FORMAT_R32G32_SFLOAT
- VK_FORMAT_R32G32B32A32_UINT
- VK_FORMAT_R32G32B32A32_SINT
- VK_FORMAT_R32G32B32A32_SFLOAT
- VK_FORMAT_B10G11R11_UFLOAT_PACK32
- VK_FORMAT_E5B9G9R9_UFLOAT_PACK32
- VK_FORMAT_D16_UNORM
- VK_FORMAT_D32_SFLOAT
- all the compressed image formats from at least one of the following sets of compressed image formats:
  - BC compressed formats
    - VK_FORMAT_BC1_RGB_UNORM_BLOCK

    * VK_FORMAT_BC1_RGB_SRGB_BLOCK
    * VK_FORMAT_BC1_RGBA_UNORM_BLOCK
    * VK_FORMAT_BC1_RGBA_SRGB_BLOCK
    * VK_FORMAT_BC2_UNORM_BLOCK
    * VK_FORMAT_BC2_SRGB_BLOCK
    * VK_FORMAT_BC3_UNORM_BLOCK
    * VK_FORMAT_BC3_SRGB_BLOCK
    * VK_FORMAT_BC4_UNORM_BLOCK
    * VK_FORMAT_BC4_SNORM_BLOCK
    * VK_FORMAT_BC5_UNORM_BLOCK
    * VK_FORMAT_BC5_SNORM_BLOCK
    * VK_FORMAT_BC6H_UFLOAT_BLOCK
    * VK_FORMAT_BC6H_SFLOAT_BLOCK
    * VK_FORMAT_BC7_UNORM_BLOCK
    * VK_FORMAT_BC7_SRGB_BLOCK
  - ETC2 and EAC compressed formats
    * VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK
    * VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK
    * VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK
    * VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK
    * VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK
    * VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK
    * VK_FORMAT_EAC_R11_UNORM_BLOCK
    * VK_FORMAT_EAC_R11_SNORM_BLOCK
    * VK_FORMAT_EAC_R11G11_UNORM_BLOCK
    * VK_FORMAT_EAC_R11G11_SNORM_BLOCK
  - ASTC LDR compressed formats
    * VK_FORMAT_ASTC_4x4_UNORM_BLOCK
    * VK_FORMAT_ASTC_4x4_SRGB_BLOCK
    * VK_FORMAT_ASTC_5x4_UNORM_BLOCK
    * VK_FORMAT_ASTC_5x4_SRGB_BLOCK
    * VK_FORMAT_ASTC_5x5_UNORM_BLOCK
    * VK_FORMAT_ASTC_5x5_SRGB_BLOCK
    * VK_FORMAT_ASTC_6x5_UNORM_BLOCK
    * VK_FORMAT_ASTC_6x5_SRGB_BLOCK
    * VK_FORMAT_ASTC_6x6_UNORM_BLOCK
    * VK_FORMAT_ASTC_6x6_SRGB_BLOCK
    * VK_FORMAT_ASTC_8x5_UNORM_BLOCK
    * VK_FORMAT_ASTC_8x5_SRGB_BLOCK
    * VK_FORMAT_ASTC_8x6_UNORM_BLOCK
    * VK_FORMAT_ASTC_8x6_SRGB_BLOCK
    * VK_FORMAT_ASTC_8x8_UNORM_BLOCK
    * VK_FORMAT_ASTC_8x8_SRGB_BLOCK
    * VK_FORMAT_ASTC_10x5_UNORM_BLOCK

  - VK_FORMAT_ASTC_10x5_SRGB_BLOCK
  - VK_FORMAT_ASTC_10x6_UNORM_BLOCK
  - VK_FORMAT_ASTC_10x6_SRGB_BLOCK
  - VK_FORMAT_ASTC_10x8_UNORM_BLOCK
  - VK_FORMAT_ASTC_10x8_SRGB_BLOCK
  - VK_FORMAT_ASTC_10x10_UNORM_BLOCK
  - VK_FORMAT_ASTC_10x10_SRGB_BLOCK
  - VK_FORMAT_ASTC_12x10_UNORM_BLOCK
  - VK_FORMAT_ASTC_12x10_SRGB_BLOCK
  - VK_FORMAT_ASTC_12x12_UNORM_BLOCK
  - VK_FORMAT_ASTC_12x12_SRGB_BLOCK

The VK_FORMAT_FEATURE_STORAGE_IMAGE_BIT feature must be supported in *optimalTilingFeatures* for the following formats:

- VK_FORMAT_R8G8B8A8_UNORM
- VK_FORMAT_R8G8B8A8_SNORM
- VK_FORMAT_R8G8B8A8_UINT
- VK_FORMAT_R8G8B8A8_SINT
- VK_FORMAT_R16G16B16A16_UINT
- VK_FORMAT_R16G16B16A16_SINT
- VK_FORMAT_R16G16B16A16_SFLOAT
- VK_FORMAT_R32_UINT
- VK_FORMAT_R32_SINT
- VK_FORMAT_R32_SFLOAT
- VK_FORMAT_R32G32_UINT
- VK_FORMAT_R32G32_SINT
- VK_FORMAT_R32G32_SFLOAT
- VK_FORMAT_R32G32B32A32_UINT
- VK_FORMAT_R32G32B32A32_SINT
- VK_FORMAT_R32G32B32A32_SFLOAT

The VK_FORMAT_FEATURE_STORAGE_IMAGE_ATOMIC_BIT feature must be supported in *optimalTilingFeatures* for the following formats:

- VK_FORMAT_R32_UINT
- VK_FORMAT_R32_SINT

The VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BIT and VK_FORMAT_FEATURE_BLIT_DST_BIT features must be supported in *optimalTilingFeatures* for the following formats:

- VK_FORMAT_R5G6B5_UNORM_PACK16
- VK_FORMAT_R8_UNORM
- VK_FORMAT_R8_SNORM
- VK_FORMAT_R8_UINT
- VK_FORMAT_R8_SINT
- VK_FORMAT_R8G8_UNORM
- VK_FORMAT_R8G8_SNORM
- VK_FORMAT_R8G8_UINT
- VK_FORMAT_R8G8_SINT
- VK_FORMAT_R8G8B8A8_UNORM
- VK_FORMAT_R8G8B8A8_SNORM
- VK_FORMAT_R8G8B8A8_UINT
- VK_FORMAT_R8G8B8A8_SINT
- VK_FORMAT_R8G8B8A8_SRGB
- VK_FORMAT_A8B8G8R8_UNORM_PACK32
- VK_FORMAT_A8B8G8R8_SNORM_PACK32
- VK_FORMAT_A8B8G8R8_UINT_PACK32
- VK_FORMAT_A8B8G8R8_SINT_PACK32
- VK_FORMAT_A8B8G8R8_SRGB_PACK32
- VK_FORMAT_B8G8R8A8_UNORM
- VK_FORMAT_B8G8R8A8_SRGB
- VK_FORMAT_A2R10G10B10_UNORM_PACK32
- VK_FORMAT_A2B10G10R10_UNORM_PACK32
- VK_FORMAT_A2B10G10R10_UINT_PACK32
- VK_FORMAT_R16_UNORM
- VK_FORMAT_R16_SNORM
- VK_FORMAT_R16_UINT
- VK_FORMAT_R16_SINT
- VK_FORMAT_R16_SFLOAT
- VK_FORMAT_R16G16_UNORM
- VK_FORMAT_R16G16_SNORM
- VK_FORMAT_R16G16_UINT

- VK_FORMAT_R16G16_SINT
- VK_FORMAT_R16G16_SFLOAT
- VK_FORMAT_R16G16B16A16_UNORM
- VK_FORMAT_R16G16B16A16_SNORM
- VK_FORMAT_R16G16B16A16_UINT
- VK_FORMAT_R16G16B16A16_SINT
- VK_FORMAT_R16G16B16A16_SFLOAT
- VK_FORMAT_R32_UINT
- VK_FORMAT_R32_SINT
- VK_FORMAT_R32_SFLOAT
- VK_FORMAT_R32G32_UINT
- VK_FORMAT_R32G32_SINT
- VK_FORMAT_R32G32_SFLOAT
- VK_FORMAT_R32G32B32A32_UINT
- VK_FORMAT_R32G32B32A32_SINT
- VK_FORMAT_R32G32B32A32_SFLOAT

The VK_FORMAT_FEATURE_COLOR_ATTACHMENT_BLEND_BIT feature must be supported in *optimalTilingFeatures* for the following formats:

- VK_FORMAT_R5G6B5_UNORM_PACK16
- VK_FORMAT_R8_UNORM
- VK_FORMAT_R8_SNORM
- VK_FORMAT_R8G8_UNORM
- VK_FORMAT_R8G8_SNORM
- VK_FORMAT_R8G8B8A8_UNORM
- VK_FORMAT_R8G8B8A8_SNORM
- VK_FORMAT_R8G8B8A8_SRGB
- VK_FORMAT_A8B8G8R8_UNORM_PACK32
- VK_FORMAT_A8B8G8R8_SNORM_PACK32
- VK_FORMAT_A8B8G8R8_SRGB_PACK32
- VK_FORMAT_B8G8R8A8_UNORM
- VK_FORMAT_B8G8R8A8_SRGB
- VK_FORMAT_A2B10G10R10_UNORM_PACK32

- VK_FORMAT_A2R10G10B10_UNORM_PACK32
- VK_FORMAT_R16_UNORM
- VK_FORMAT_R16_SNORM
- VK_FORMAT_R16_SFLOAT
- VK_FORMAT_R16G16_UNORM
- VK_FORMAT_R16G16_SNORM
- VK_FORMAT_R16G16_SFLOAT
- VK_FORMAT_R16G16B16A16_UNORM
- VK_FORMAT_R16G16B16A16_SNORM
- VK_FORMAT_R16G16B16A16_SFLOAT

The VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT feature must be supported in *optimalTilingFeatures* for the following formats:

- VK_FORMAT_D16_UNORM
- at least one of VK_FORMAT_X8_D24_UNORM_PACK32 or VK_FORMAT_D32_SFLOAT
- at least one of VK_FORMAT_D24_UNORM_S8_UINT or VK_FORMAT_D32_SFLOAT_S8_UINT

The VK_FORMAT_FEATURE_VERTEX_BUFFER_BIT feature must be supported in *bufferFeatures* for the following formats:

- VK_FORMAT_R8_UNORM
- VK_FORMAT_R8_SNORM
- VK_FORMAT_R8_UINT
- VK_FORMAT_R8_SINT
- VK_FORMAT_R8G8_UNORM
- VK_FORMAT_R8G8_SNORM
- VK_FORMAT_R8G8_UINT
- VK_FORMAT_R8G8_SINT
- VK_FORMAT_R8G8B8A8_UNORM
- VK_FORMAT_R8G8B8A8_SNORM
- VK_FORMAT_R8G8B8A8_UINT
- VK_FORMAT_R8G8B8A8_SINT
- VK_FORMAT_A8B8G8R8_UNORM_PACK32
- VK_FORMAT_A8B8G8R8_SNORM_PACK32
- VK_FORMAT_A8B8G8R8_UINT_PACK32

- VK_FORMAT_A8B8G8R8_SINT_PACK32
- VK_FORMAT_B8G8R8A8_UNORM
- VK_FORMAT_A2R10G10B10_UNORM_PACK32
- VK_FORMAT_A2R10G10B10_SNORM_PACK32
- VK_FORMAT_A2R10G10B10_UINT_PACK32
- VK_FORMAT_A2R10G10B10_SINT_PACK32
- VK_FORMAT_A2B10G10R10_UNORM_PACK32
- VK_FORMAT_A2B10G10R10_UINT_PACK32
- VK_FORMAT_R16_UNORM
- VK_FORMAT_R16_SNORM
- VK_FORMAT_R16_UINT
- VK_FORMAT_R16_SINT
- VK_FORMAT_R16_SFLOAT
- VK_FORMAT_R16G16_UNORM
- VK_FORMAT_R16G16_SNORM
- VK_FORMAT_R16G16_UINT
- VK_FORMAT_R16G16_SINT
- VK_FORMAT_R16G16_SFLOAT
- VK_FORMAT_R16G16B16A16_UNORM
- VK_FORMAT_R16G16B16A16_SNORM
- VK_FORMAT_R16G16B16A16_UINT
- VK_FORMAT_R16G16B16A16_SINT
- VK_FORMAT_R16G16B16A16_SFLOAT
- VK_FORMAT_R32_UINT
- VK_FORMAT_R32_SINT
- VK_FORMAT_R32_SFLOAT
- VK_FORMAT_R32G32_UINT
- VK_FORMAT_R32G32_SINT
- VK_FORMAT_R32G32_SFLOAT
- VK_FORMAT_R32G32B32_UINT
- VK_FORMAT_R32G32B32_SINT
- VK_FORMAT_R32G32B32_SFLOAT

- VK_FORMAT_R32G32B32A32_UINT
- VK_FORMAT_R32G32B32A32_SINT
- VK_FORMAT_R32G32B32A32_SFLOAT

The VK_FORMAT_FEATURE_UNIFORM_TEXEL_BUFFER_BIT feature must be supported in *bufferFeatures* for the following formats:

- VK_FORMAT_R8_UNORM
- VK_FORMAT_R8_SNORM
- VK_FORMAT_R8_UINT
- VK_FORMAT_R8_SINT
- VK_FORMAT_R8G8_UNORM
- VK_FORMAT_R8G8_SNORM
- VK_FORMAT_R8G8_UINT
- VK_FORMAT_R8G8_SINT
- VK_FORMAT_R8G8B8A8_UNORM
- VK_FORMAT_R8G8B8A8_SNORM
- VK_FORMAT_R8G8B8A8_UINT
- VK_FORMAT_R8G8B8A8_SINT
- VK_FORMAT_A8B8G8R8_UNORM_PACK32
- VK_FORMAT_A8B8G8R8_SNORM_PACK32
- VK_FORMAT_A8B8G8R8_UINT_PACK32
- VK_FORMAT_A8B8G8R8_SINT_PACK32
- VK_FORMAT_B8G8R8A8_UNORM
- VK_FORMAT_A2B10G10R10_UNORM_PACK32
- VK_FORMAT_A2B10G10R10_UINT_PACK32
- VK_FORMAT_R16_UNORM
- VK_FORMAT_R16_SNORM
- VK_FORMAT_R16_UINT
- VK_FORMAT_R16_SINT
- VK_FORMAT_R16_SFLOAT
- VK_FORMAT_R16G16_UNORM
- VK_FORMAT_R16G16_SNORM
- VK_FORMAT_R16G16_UINT

- VK_FORMAT_R16G16_SINT
- VK_FORMAT_R16G16_SFLOAT
- VK_FORMAT_R16G16B16A16_UNORM
- VK_FORMAT_R16G16B16A16_SNORM
- VK_FORMAT_R16G16B16A16_UINT
- VK_FORMAT_R16G16B16A16_SINT
- VK_FORMAT_R16G16B16A16_SFLOAT
- VK_FORMAT_R32_UINT
- VK_FORMAT_R32_SINT
- VK_FORMAT_R32_SFLOAT
- VK_FORMAT_R32G32_UINT
- VK_FORMAT_R32G32_SINT
- VK_FORMAT_R32G32_SFLOAT
- VK_FORMAT_R32G32B32A32_UINT
- VK_FORMAT_R32G32B32A32_SINT
- VK_FORMAT_R32G32B32A32_SFLOAT
- VK_FORMAT_B10G11R11_UFLOAT_PACK32

The VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_BIT feature must be supported in *bufferFeatures* for the following formats:

- VK_FORMAT_R8G8B8A8_UNORM
- VK_FORMAT_R8G8B8A8_SNORM
- VK_FORMAT_R8G8B8A8_UINT
- VK_FORMAT_R8G8B8A8_SINT
- VK_FORMAT_A8B8G8R8_UNORM_PACK32
- VK_FORMAT_A8B8G8R8_SNORM_PACK32
- VK_FORMAT_A8B8G8R8_UINT_PACK32
- VK_FORMAT_A8B8G8R8_SINT_PACK32
- VK_FORMAT_R16G16B16A16_UINT
- VK_FORMAT_R16G16B16A16_SINT
- VK_FORMAT_R16G16B16A16_SFLOAT
- VK_FORMAT_R32_UINT
- VK_FORMAT_R32_SINT

- VK_FORMAT_R32_SFLOAT
- VK_FORMAT_R32G32_UINT
- VK_FORMAT_R32G32_SINT
- VK_FORMAT_R32G32_SFLOAT
- VK_FORMAT_R32G32B32A32_UINT
- VK_FORMAT_R32G32B32A32_SINT
- VK_FORMAT_R32G32B32A32_SFLOAT

The VK_FORMAT_FEATURE_STORAGE_TEXEL_BUFFER_ATOMIC_BIT properties must be supported in *bufferFeatures* for the following formats:

- VK_FORMAT_R32_UINT
- VK_FORMAT_R32_SINT

29.4 Additional Image Capabilities

In addition to the minimum capabilities described in the previous sections (Limits and Formats), implementations may support additional capabilities for certain types of images. For example, larger dimensions or additional sample counts for certain image types, or additional capabilities for *linear* tiling format images.

Additional capabilities specific to image types are queried with the command:

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

physicalDevice is the physical device from which to query the image capabilities. *format*, *type*, *tiling*, *usage*, and *flags* correspond to VkImageCreateInfo members that would be consumed by `vkCreateImage`. *pImageFormatProperties* points to an instance of the VkImageFormatProperties structure.

Valid Usage

- *physicalDevice* must be a valid VkPhysicalDevice handle
- *format* must be a valid `VkFormat` value
- *type* must be a valid `VkImageType` value
- *tiling* must be a valid `VkImageTiling` value

- *usage* must be a valid combination of `VkImageUsageFlagBits` values
- *usage* must not be `0`
- *flags* must be a valid combination of `VkImageCreateFlagBits` values
- *pImageFormatProperties* must be a pointer to a VkImageFormatProperties structure

The definition of this structure is:

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

- *maxExtent* are the maximum image dimensions. See the Allowed extent values based on imageType table below for how these values are constrained by *type*.
- *maxMipLevels* is the maximum number of mipmap levels.
- *maxArrayLayers* is the maximum number of array layers.
- *sampleCounts* is a bitmask of `VkSampleCountFlagBits` specifying all the supported sample counts for this image. When *tiling* is VK_IMAGE_TILING_LINEAR the *sampleCounts* will be set to VK_SAMPLE_COUNT_1_BIT. Otherwise the bits set here are a superset of the corresponding limits for the image type in the VkPhysicalDeviceLimits struct. For non-integer color images this is *sampledImageColorSampleCounts*, for integer format color images this is *sampledImageIntegerSampleCounts*, for depth format images this is *sampledImageDepthSampleCounts*, for stencil format images this is *sampledImageStencilSampleCounts*, and if *usage* has VK_IMAGE_USAGE_STORAGE_BIT set this is *storageImageSampleCounts*.
- *maxResourceSize* is the maximum total image size. This may be used to limit total resource size, since it may not be possible to maximize all dimensions at once.

Table 29.10: Allowed extent values based on imageType

VkImageType	**maxExtent values**

`VK_IMAGE_TYPE_1D`	width >= 1
	height = 1
	depth = 1

`VK_IMAGE_TYPE_2D`	width >= 1 height >= 1 depth = 1
`VK_IMAGE_TYPE_3D`	width >= 1 height >= 1 depth >= 1

If *format* is not a supported image format, or if the combination of *format*, *type*, *tiling*, *usage*, and *flags* is not supported for images, then **vkGetPhysicalDeviceImageFormatProperties** returns VK_ERROR_FORMAT_NOT_SUPPORTED.

The limitations on an image format that are reported by **vkGetPhysicalDeviceImageFormatProperties** have the following property: if **usage1** and **usage2** of type VkImageUsageFlags and **flags1** and **flags2** of type VkImageCreateFlags are such that **usage1** contains a subset of the bits set in **usage2** and **flags1** contains a subset of the bits set in **flags2**, then the limitations for **usage1** and **flags1** must be no more strict than the limitations for **usage2** and **flags2**, for all values of *format*, *type*, and *tiling*.

Chapter 30

Glossary

The terms defined in this section are used consistently throughout this Specification and may be used with or without capitalization.

Adjacent Vertex
A vertex in an adjacency primitive topology that is not part of a given primitive, but is accessible in geometry shaders.

Aliased Range (Memory)
A range of a device memory allocation which is bound to multiple resources simultaneously.

API Order
A set of ordering rules which govern how primitives in draw commands affect the framebuffer.

Attachment (Render Pass)
A zero-based integer index name used in render pass creation to refer to a framebuffer attachment that is accessed by one or more subpasses. The index also refers to an attachment description which includes information about the properties of the image view that will later be attached.

Available
See Memory Dependency.

Back-Facing
See Facingness.

Batch
A collection of command buffers which are submitted to a queue, and corresponding semaphores to wait and signal. Corresponds to an instance of the `VkSubmitInfo` structure.

Blend Constant
Four floating point (RGBA) values used as an input to blending.

Blending
Arithmetic operations between a fragment color value and a value in a color attachment, which produce a final color value to be written to the attachment.

Buffer
A resource which represents a linear array of data in device memory. Represented by a VkBuffer object.

Buffer View
An object which represents a range of a specific buffer, and state that controls how the contents are interpreted. Represented by a VkBufferView object.

Built-In Variable
A variable decorated in a shader, where the decoration makes the variable take values provided by the execution environment or values that are generated by fixed-function pipeline stages.

Built-In Interface Block
A block defined in a shader which contains only variables decorated with built-in decorations, and which is used to match against other shader stages.

Clip Coordinates
The homogeneous coordinate space that vertex positions (**`Position`** decoration) are written in by vertex processing stages.

Clip Distance
A built-in output from vertex processing stages, which defines a clip half-space against which the primitive is clipped.

Clip Volume
The intersection of the view volume with all clip half-spaces.

Color Attachment
A subpass attachment point, or image view, which is the target of fragment color outputs and blending.

Combined Image Sampler
A descriptor type that includes both a sampled image and a sampler.

Command Buffer
An object which records commands, to be submitted to a queue. Represented by a VkCommandBuffer object.

Command Pool
An object that command buffer memory is allocated from, and which owns that memory. Command pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use. Represented by a VkCommandPool object.

Compatible Allocator
When allocators are compatible, allocations from each allocator can be freed by the other allocator.

Compatible Image Formats
When formats are compatible, images created with one of the formats can have image views created from it using any of the compatible formats.

Compatible Queues
Queues within a queue family. Compatible queues have identical properties.

Cull Distance
A built-in output from vertex processing stages, which defines a cull half-space where the primitive is rejected if all vertices have a negative value for the same cull distance.

Cull Volume
The intersection of the view volume with all cull half-spaces.

Decoration (SPIR-V)
Auxiliary information such as built-in variables, stream numbers, invariance, interpolation type, relaxed precision, etc., added to variables or structure-type members through decorations.

Depth/Stencil Attachment
A subpass attachment point, or image view, which is the target of depth and/or stencil test operations and writes.

Descriptor
Information about a resource or resource view written into a descriptor set which is used to access the resource or view from a shader.

Descriptor Pool
An object which descriptor sets are allocated from, and which owns the storage of those descriptor sets. Descriptor pools aid multithreaded performance by enabling different threads to use different allocators, without internal synchronization on each use. Represented by a VkDescriptorPool object.

Descriptor Set
An object that resource descriptors are written into via the API, and which can be bound to a command buffer such that the descriptors contained within it can be accessed from shaders. Represented by a VkDescriptorSet object.

Descriptor Set Layout
An object which defines the set of resources (types and counts) and their relative arrangement (in the binding namespace) within a descriptor set. Used when allocating descriptor sets and when creating pipeline layouts. Represented by a VkDescriptorSetLayout object.

Device
The processor(s) and execution environment that performs tasks requested by the application via the Vulkan API.

Device Memory
Memory accessible to the device. Represented by a VkDeviceMemory object.

Device-Local Memory
Memory that is connected to the device, and may be more performant for device access than host-local memory.

Dispatchable Handle
A handle of a pointer handle type which may be used by layers as part of intercepting API commands. The first argument to each Vulkan command is a dispatchable handle type.

Dispatching Commands
Commands that provoke work using a compute pipeline. Includes `vkCmdDispatch` and `vkCmdDispatchIndirect`.

Drawing Commands
Commands that provoke work using a graphics pipeline. Includes `vkCmdDraw`, `vkCmdDrawIndexed`, `vkCmdDrawIndirect`, and `vkCmdDrawIndexedIndirect`.

Dynamic Storage Buffer
A storage buffer whose offset is specified each time the storage buffer is bound to a command buffer via a descriptor set.

Dynamic Uniform Buffer
A uniform buffer whose offset is specified each time the uniform buffer is bound to a command buffer via a descriptor set.

Event
A synchronization primitive which is signalled when execution of previous commands complete through a specified set of pipeline stages. Events can be waited on by the device and polled by the host. Represented by a VkEvent object.

Execution Dependency
A dependency which guarantees that certain pipeline stages' work for a first set of commands has completed execution before certain pipeline stages' work for a second set of commands begins execution. This is accomplished via pipeline barriers, subpass dependencies, events, or implicit ordering operations.

Execution Dependency Chain
A sequence of execution dependencies which transitively act as an execution dependency.

External synchronization
A type of synchronization required of the application, where parameters defined to be externally synchronized must not be used simultaneously in multiple threads.

Facingness (Polygon)
A classification of a polygon as either front-facing or back-facing, depending on the orientation (winding order) of its vertices.

Fence
A synchronization primitive which is signaled when a set of batches or sparse binding operations complete execution on a queue. Fences can be waited on by the host. Represented by a VkFence object.

Flatshading
A property of a vertex attribute which causes the value from a single vertex (the provoking vertex) to be used for all vertices in a primitive, and for interpolation of that attribute to return that single value unaltered.

Fragment Input Attachment Interface
A fragment shader entry point's variables with **`Input`** storage class and a decoration of **`InputAttachmentIndex`**, which receive values from input attachments.

Fragment Output Interface
A fragment shader entry point's variables with **`Output`** storage class, which output to color and/or depth/stencil attachments.

Framebuffer
A collection of image views and a set of dimensions which, in conjunction with a render pass, define the inputs and outputs used by drawing commands. Represented by a VkFramebuffer object.

Framebuffer Attachment
One of the image views used in a framebuffer.

Framebuffer Coordinates
A coordinate system in which adjacent pixels' coordinates differ by 1 in x and/or y, with $(0,0)$ in the upper left corner and pixel centers at half-integers.

Front-Facing
See Facingness.

Handle
An opaque integer or pointer value used to refer to a Vulkan object. Each object type has a unique handle type.

Helper Invocation
A fragment shader invocation that is created solely for the purposes of evaluating derivatives for use in non-helper fragment shader invocations, and which does not have side effects.

Host
The processor(s) and execution environment that the application runs on, and that the Vulkan API is exposed on.

Host Memory
Memory not accessible to the device, used to store implementation data structures.

Host-Accessible Subresource
A buffer, or a linear image subresource in either the VK_IMAGE_LAYOUT_PREINITIALIZED or VK_IMAGE_LAYOUT_GENERAL layout. Host-accessible subresources have a well-defined addressing scheme which can be used by the host.

Host-Local Memory
Memory that is not local to the device, and may be less performant for device access than device-local memory.

Host-Visible Memory
Device memory that can be mapped on the host and can be read and written by the host.

Image
A resource which represents a multi-dimensional formatted interpretation of device memory. Represented by a VkImage object.

Image Subresource
A specific mipmap level and layer of an image.

Image Subresource Range
A set of image subresources which are contiguous mipmap levels and layers.

Image View
An object which represents a subresource range of a specific image, and state that controls how the contents are interpreted. Represented by a VkImageView object.

Immutable Sampler
A sampler descriptor provided at descriptor set layout creation time, and which is used for that binding in all descriptor sets created from the layout, and cannot be changed.

Index Buffer
A buffer bound via `vkCmdBindIndexBuffer` which is the source of index values used to fetch vertex attributes for a `vkCmdDrawIndexed` or `vkCmdDrawIndexedIndirect` command.

Indirect Commands
Drawing or dispatching commands which source some of their parameters from structures in buffer memory. Includes `vkCmdDrawIndirect`, `vkCmdDrawIndexedIndirect`, and `vkCmdDispatchIndirect`.

Input Attachment
A descriptor type that represents an image view, and supports unfiltered read-only access in a shader, only at the fragment's location in the view.

Instance
The top-level Vulkan object, which represents the application's connection to the implementation. Represented by a VkInstance object.

Internal Synchronization
A type of synchronization required of the implementation, where parameters not defined to be externally synchronized may require internal mutexing to avoid multithreaded race conditions.

Invocation (Shader)
A single execution of an entry point in a SPIR-V module. For example, a single vertex's execution of a vertex shader or a single fragment's execution of a fragment shader.

Logical Device
An object that represents the application's interface to the physical device. The logical device is the parent of most Vulkan objects. Represented by a VkDevice object.

Logical Operation
Bitwise operations between a fragment color value and a value in a color attachment, which produce a final color value to be written to the attachment.

Lost Device
A state which a logical device may be in as a result of hardware errors or other exceptional conditions.

Mappable
See Host-Visible Memory.

Memory Dependency
A sequence of operations that makes writes available, performs an execution dependency between the writes and subsequent accesses, and makes available writes visible to later accesses. In order for the effects of a write to be coherent with later accesses, it must be made available from the old access type and then made visible to the new access type.

Memory Heap
A region of memory from which device memory allocations can be made.

Memory Type
An index used to select a set of memory properties (e.g. mappable, cached) for a device memory allocation.

Mip Tail Region
The set of mipmap levels of a sparse residency texture which are too small to fill a block, and which must all be bound to memory collectively and opaquely.

Non-Dispatchable Handle
A handle of an integer handle type. Handle values may not be unique, even for two objects of the same type.

Normalized
A value which is interpreted as being in the range $[0,1]$ as a result of being implicitly divided by some other value.

Normalized Device Coordinates
A coordinate space after perspective division is applied to clip coordinates, and before the viewport transformation converts to framebuffer coordinates.

Overlapped Range (Aliased Range)
The aliased range of a device memory allocation which intersects a given subresource of an image or range of a buffer.

Packed Format
A format whose components are stored as a single data element in memory, with their relative locations defined within that element.

Physical Device
An object that represents a single device in the system. Represented by a VkPhysicalDevice object.

Pipeline
An object which controls how graphics or compute work is executed on the device. A pipeline includes one or more shaders, as well as state controlling any non-programmable stages of the pipeline. Represented by a VkPipeline object.

Pipeline Barrier
An execution and/or memory dependency recorded as an explicit command in a command buffer, which forms a dependency between the previous and subsequent commands.

Pipeline Cache
An object which can be used to collect and retrieve information from pipelines as they are created, and can be populated with previously retrieved information in order to accelerate pipeline creation. Represented by a VkPipelineCache object.

Pipeline Layout
An object which defines the set of resources (via a collection of descriptor set layouts) and push constants used by pipelines that are created using the layout. Used when creating a pipeline and when binding descriptor sets and setting push constant values. Represented by a VkPipelineLayout object.

Point Sampling (Rasterization)
A rule which determines whether a fragment sample location is covered by a polygon primitive by testing whether the sample location is in the interior of the polygon in framebuffer-space, or on the boundary of the polygon according to the tie-breaking rules.

Preserve Attachment
One of a list of attachments in a subpass description which is not read or written by the subpass, but which is read or written on earlier and later subpasses and whose contents must be preserved through this subpass.

Primitive Topology
State that controls how vertices are assembled into primitives, e.g. as lists of triangles, strips of lines, etc..

Provoking Vertex
The vertex in a primitive from which flatshaded attribute values are taken. This is generally the "first" vertex in the primitive, and depends on the primitive topology.

Push Constants
A small bank of values writable via the API and accessible in shaders. Push constants allow the application to set values used in shaders without creating buffers or modifying and binding descriptor sets for each update.

Push Constant Interface
The set of variables with **`PushConstant`** storage class which are statically referenced by a shader entry point, and which receive values from push constant commands.

Query Pool
An object which contains a number of query entries and their associated state and results. Represented by a VkQueryPool object.

Queue
An object which executes command buffers and sparse binding operations on a device. Represented by a VkQueue object.

Queue Family
A set of queues which have common properties and support the same functionality, as advertised in `VkQueueFamilyProperties`.

Queue Submission
An operation that is enqueued for execution on a queue typically as a result of the issue of commands of the form **`vkQueue*`**.

Render Pass
An object which represents a set of framebuffer attachments and phases of rendering using those attachments. Represented by a VkRenderPass object.

Render Pass Instance
A use of a render pass in a command buffer.

Resolve Attachment
A subpass attachment point, or image view, which is the target of a multisample resolve operation from the corresponding color attachment at the end of the subpass.

Sampled Image
A descriptor type that represents an image view, and supports filtered (sampled) and unfiltered read-only acccess in a shader.

Sampler
An object which contains state that controls how sampled image data is sampled (or filtered) when accessed in a shader. Also a descriptor type describing the object. Represented by a VkSampler object.

Self-Dependency
A subpass dependency from a subpass to itself, i.e. with *`srcSubpass`* equal to *`dstSubpass`*. A self-dependency is not automatically performed during a render pass instance, rather a subset of it can be performed via `vkCmdPipelineBarrier` during the subpass.

Semaphore
A synchronization primitive which supports signal and wait operations, and can be used to synchronize operations within a queue or across queues. Represented by a VkSemaphore object.

Shader
Instructions selected (via an entry point) from a shader module, which are executed in a shader stage.

Shader Code
A stream of instructions used to describe the operation of a shader.

Shader Module
A collection of shader code, potentially including several functions and entry points, which is used to create shaders in pipelines. Represented by a VkShaderModule object.

Shader Stage
A stage of the graphics or compute pipeline which executes shader code.

Side Effect
A store to memory or atomic operation on memory from a shader invocation.

Storage Buffer
A descriptor type that represents a buffer, and supports reads, writes, and atomics in a shader.

Storage Image
A descriptor type that represents an image view, and supports unfiltered loads, stores, and atomics in a shader.

Storage Texel Buffer
A descriptor type that represents a buffer view, and supports unfiltered, formatted reads, writes, and atomics in a shader.

Subpass
A phase of rendering within a render pass, which reads and writes a subset of the attachments.

Subpass Dependency
An execution and/or memory dependency between two subpasses described as part of render pass creation, and automatically performed between subpasses in a render pass instance. A subpass dependency limits the overlap of execution of the pair of subpasses, and can provide guarantees of memory coherence between accesses in the subpasses.

Subpass Description
Lists of attachment indices for input attachments, color attachments, depth/stencil attachment, resolve attachments, and preserve attachments used by the subpass in a render pass.

Subset (Self-Dependency)
A subset of a self-dependency is a pipeline barrier performed during the subpass of the self-dependency, and whose stage masks and access masks each contain a subset of the bits set in the identically named mask in the self-dependency.

Texel Coordinate System
One of three coordinate systems (normalized, unnormalized, integer) which define how texel coordinates are interpreted in an image or a specific mipmap level of an image.

Uniform Texel Buffer
A descriptor type that represents a buffer view, and supports unfiltered, formatted, read-only access in a shader.

Uniform Buffer
A descriptor type that represents a buffer, and supports read-only access in a shader.

Unnormalized
A value which is interpreted according to its conventional interpretation, and is not normalized.

User-Defined Variable Interface
A shader entry point's variables with `Input` or `Output` storage class, which are not built-in variables.

Vertex Input Attribute
A graphics pipeline resource which produces input values for the vertex shader by reading data from a vertex input binding and converting it to the attribute's format.

Vertex Input Binding
A graphics pipeline resource which is bound to a buffer and includes state which affects addressing calculations within that buffer.

Vertex Input Interface
A vertex shader entry point's variables with `Input` storage class, which receive values from vertex input attributes.

Vertex Processing Stages
A set of shader stages which comprises the vertex shader, tessellation control shader, tessellation evaluation shader, and geometry shader stages.

View Volume
A subspace in homogeneous coordinates, corresponding to post-projection x and y values between -1 and +1, and z values between 0 and +1.

Viewport Transformation
A transformation from normalized device coordinates to framebuffer coordinates, based on a viewport rectangle and depth range.

Visible
See Memory Dependency.

Chapter 31

Common Abbreviations

Abbreviations and acronyms are sometimes used in the Specification and the API where they are considered clear and commonplace, and are defined here:

Src
Source

Dst
Destination

Min
Minimum

Max
Maximum

Rect
Rectangle

Info
Information

LOD
Level of Detail

ID
Identifier

UUID
Universally Unique Identifier

Op
Operation

R
Red color channel

G
Green color channel

B

Blue color channel

A

Alpha color channel

Chapter 32

Prefixes

Prefixes are used in the API to denote specific semantic meaning of Vulkan names, or as a label to avoid name clashes, and are explained here:

VK/Vk/vk
Vulkan namespace
All types, commands, enumerants and defines in this specification are prefixed with these two characters.

PFN/pfn
Function Pointer
Denotes that a type is a function pointer, or that a variable is of a pointer type.

p
Pointer
Variable is a pointer.

vkCmd
Commands which record commands in command buffers
These API commands do not result in immediate processing on the device. Instead, they record the requested action in a command buffer for execution when the command buffer is submitted to a queue.

s
Structure
Used to denote the VK_STRUCTURE_TYPE* member of each structure in *sType*

Appendix A

Vulkan Environment for SPIR-V

Shaders for Vulkan are defined by the [Khronos SPIR-V Specification] as well as the [Khronos SPIR-V Extended Instructions for GLSL Specification]. This appendix defines additional SPIR-V requirements applying to Vulkan shaders.

A.1 Required Versions and Formats

A Vulkan 1.0 implementation must support the 1.0 version of SPIR-V and the 1.0 version of the SPIR-V Extended Instructions for GLSL.

A SPIR-V module passed into `vkCreateShaderModule` is interpreted as a series of 32-bit words in platform endianness, with literal strings packed as described in section 2.2 of the SPIR-V Specification. The first few words of the SPIR-V module must be a magic number and a SPIR-V version number, as described in section 2.3 of the SPIR-V Specification.

A.2 Capabilities

Implementations must support the following capability operands declared by **OpCapability**:

- Matrix
- Shader
- InputAttachment
- Sampled1D
- Image1D
- SampledBuffer
- ImageBuffer
- ImageQuery
- DerivativeControl

Implementations may support features that are not required by the Specification, as described in the Features chapter. If such a feature is supported, then any capability operand(s) corresponding to that feature must also be supported.

Table A.1: SPIR-V Capabilities which are not required, and corresponding feature names

SPIR-V OpCapability	**Vulkan feature name**
Geometry	geometryShader
Tessellation	tessellationShader
Float64	shaderFloat64
Int64	shaderInt64
Int16	shaderInt16
TessellationPointSize	shaderTessellationAndGeometryPointSize
GeometryPointSize	shaderTessellationAndGeometryPointSize
ImageGatherExtended	shaderImageGatherExtended
StorageImageMultisample	shaderStorageImageMultisample
UniformBufferArrayDynamicIndexing	shaderUniformBufferArrayDynamicIndexing
SampledImageArrayDynamicIndexing	shaderSampledImageArrayDynamicIndexing
StorageBufferArrayDynamicIndexing	shaderStorageBufferArrayDynamicIndexing
StorageImageArrayDynamicIndexing	shaderStorageImageArrayDynamicIndexing
ClipDistance	shaderClipDistance
CullDistance	shaderCullDistance
ImageCubeArray	imageCubeArray
SampleRateShading	sampleRateShading
SparseResidency	shaderResourceResidency
MinLod	shaderResourceMinLod
SampledCubeArray	imageCubeArray
ImageMSArray	shaderStorageImageMultisample
StorageImageExtendedFormats	shaderStorageImageExtendedFormats
InterpolationFunction	sampleRateShading
StorageImageReadWithoutFormat	shaderStorageImageReadWithoutFormat
StorageImageWriteWithoutFormat	shaderStorageImageWriteWithoutFormat
MultiViewport	multiViewport

The application must not pass a SPIR-V module containing any of the following to `vkCreateShaderModule`:

- any OpCapability not listed above,
- an unsupported capability, or
- a capability which corresponds to a Vulkan feature which has not been enabled.

A.3 Validation Rules within a Module

A SPIR-V module passed to `vkCreateShaderModule` must conform to the following rules:

- Every entry point must have no return value and accept no arguments.
- The **Logical** addressing model must be selected.
- **Scope** for execution must be limited to:
  - **Workgroup**

- **Subgroup**

- **Scope** for memory must be limited to:
  - **Device**
  - **Workgroup**
  - **Invocation**
- The **OriginLowerLeft** execution mode must not be used; fragment entry points must declare **OriginUpperLeft**.
- The **PixelCenterInteger** execution mode must not be used. Pixels are always centered at half-integer coordinates.
- Images
  - **OpTypeImage** must declare a scalar 32-bit float or 32-bit integer type for the *Sampled Type*. (RelaxedPrecision can be applied to a sampling instruction and to the variable holding the result of a sampling instruction.)
  - **OpSampledImage** must only consume an *Image* operand whose type has its *Sampled* operand set to 1.
  - The *(u, v)* coordinates used for a **SubpassData** must be the <id> of a constant vector (0, 0), or if a layer coordinate is used, must be a vector that was formed with constant 0 for the *u* and *v* components.
  - The *Depth* operand of **OpTypeImage** is ignored.
- Decorations
  - The **GLSLShared** and **GLSLPacked** decorations must not be used.
  - The **`Flat`**, **`NoPerspective`**, **`Sample`**, and **`Centroid`** decorations must not be used on variables with storage class other than **`Input`** or on variables used in the interface of non-fragment shader entry points.
  - The **`Patch`** decoration must not be used on variables in the interface of a vertex, geometry, or fragment shader stage's entry point.
- **OpTypeRuntimeArray** must only be used for the last member of an **OpTypeStruct** in the **Uniform** storage class.
- Linkage: See Section 9.3 for additional linking and validation rules.

A.4 Precision and Operation of SPIR-V Floating-Point Operations

The following rules apply to both single and double-precision operations:

- Positive and negative infinities and positive and negative zeros are generated as dictated by [IEEE 754], but subject to the precisions allowed in the following table.
- Dividing a non-zero by a zero results in the appropriately signed [IEEE 754] infinity.
- Any denormalized value input into a shader or potentially generated by any operation in a shader may be flushed to 0.
- The rounding mode cannot be set and is undefined.
- NaNs may not be generated. Instructions that operate on a NaN may not result in a NaN.
- Support for signaling NaNs is optional and exceptions are never raised.

The precision of double-precision operations is at least that of single precision. For single precision (32 bit) operations, precisions are required to be at least as follows, unless decorated with RelaxedPrecision:

Table A.2: Precision of core SPIRV Operations

Operation	Precision
OpFAdd	Correctly rounded.
OpFSub	Correctly rounded.
OpFMul	Correctly rounded.
OpFOrdEqual, **OpFUnordEqual**	Correct result.
OpFOrdLessThan, **OpFUnordLessThan**	Correct result.
OpFOrdGreaterThan, **OpFUnordGreaterThan**	Correct result.
OpFOrdLessThanEqual, **OpFUnordLessThanEqual**	Correct result.
OpFOrdGreaterThanEqual, **OpFUnordGreaterThanEqual**	Correct result.
OpFDiv, **OpFRem**, and **OpFMod**	2.5 ULP for b in the range [2^{-126}, 2^{126}].
conversions between types	Correctly rounded.

Table A.3: Precision of GLSL.std.450 Operations

Operation	Precision
fma()	Inherited from **OpFMul** followed by **OpFAdd**
exp(x), **exp2**(x)	(3 + 2 * |x|) ULP.
log(), **log2**()	3 ULP outside the range [0.5, 2.0]. Absolute error < 2^{-21} inside the range [0.5, 2.0].
pow(x, y)	Inherited from **exp2** (x * **log2** (y)).
sqrt()	Inherited from 1.0 **/** **inversesqrt**().
inversesqrt()	2 ULP.

GLSL.std.450 extended instructions specifically defined in terms of the above operations inherit the above errors. GLSL.std.450 extended instructions not listed above and not defined in terms of the above have undefined precision. These include, for example, the trigonometric functions and determinant.

A.5 Compatibility Between SPIR-V Image Formats And Vulkan Formats

SPIR-V Image Format	Vulkan Format
`Rgba32f`	VK_FORMAT_R32G32B32A32_SFLOAT
`Rgba16f`	VK_FORMAT_R16G16B16A16_SFLOAT
`R32f`	VK_FORMAT_R32_SFLOAT
`Rgba8`	VK_FORMAT_R8G8B8A8_UNORM
`Rgba8Snorm`	VK_FORMAT_R8G8B8A8_SNORM
`Rg32f`	VK_FORMAT_R32G32_SFLOAT
`Rg16f`	VK_FORMAT_R16G16_SFLOAT
`R11fG11fB10f`	VK_FORMAT_B10G11R11_UFLOAT_PACK32
`R16f`	VK_FORMAT_R16_SFLOAT
`Rgba16`	VK_FORMAT_R16G16B16A16_UNORM

SPIR-V Image Format	**Vulkan Format**
`Rgb10A2`	VK_FORMAT_A2B10G10R10_UNORM_PACK32
`Rg16`	VK_FORMAT_R16G16_UNORM
`Rg8`	VK_FORMAT_R8G8_UNORM
`R16`	VK_FORMAT_R16_UNORM
`R8`	VK_FORMAT_R8_UNORM
`Rgba16Snorm`	VK_FORMAT_R16G16B16A16_SNORM
`Rg16Snorm`	VK_FORMAT_R16G16_SNORM
`Rg8Snorm`	VK_FORMAT_R8G8_SNORM
`R16Snorm`	VK_FORMAT_R16_SNORM
`R8Snorm`	VK_FORMAT_R8_SNORM
`Rgba32i`	VK_FORMAT_R32G32B32A32_SINT
`Rgba16i`	VK_FORMAT_R16G16B16A16_SINT
`Rgba8i`	VK_FORMAT_R8G8B8A8_SINT
`R32i`	VK_FORMAT_R32_SINT
`Rg32i`	VK_FORMAT_R32G32_SINT
`Rg16i`	VK_FORMAT_R16G16_SINT
`Rg8i`	VK_FORMAT_R8G8_SINT
`R16i`	VK_FORMAT_R16_SINT
`R8i`	VK_FORMAT_R8_SINT
`Rgba32ui`	VK_FORMAT_R32G32B32A32_UINT
`Rgba16ui`	VK_FORMAT_R16G16B16A16_UINT
`Rgba8ui`	VK_FORMAT_R8G8B8A8_UINT
`R32ui`	VK_FORMAT_R32_UINT
`Rgb10a2ui`	VK_FORMAT_A2B10G10R10_UINT_PACK32
`Rg32ui`	VK_FORMAT_R32G32_UINT
`Rg16ui`	VK_FORMAT_R16G16_UINT
`Rg8ui`	VK_FORMAT_R8G8_UINT
`R16ui`	VK_FORMAT_R16_UINT
`R8ui`	VK_FORMAT_R8_UINT

Appendix B

Compressed Image Formats

The compressed texture formats used by Vulkan are described in the specifically identified sections of the [Khronos Data Format Specification], version 1.1.

Unless otherwise described, the quantities encoded in these compressed formats are treated as normalized, unsigned values.

B.1 Block Compressed Image Formats

Those formats listed as “sRGB-encoded” have in-memory representations of R, G and B channels which are nonlinearly-encoded as R', G' and B' according to the formula in Section 14.2.2; any alpha channel is unchanged. As part of filtering, the decoded nonlinear R', G', B' values are converted to linear R, G and B channels according to Section 14.2.1; any alpha channel is unchanged.

Table B.1: Mapping of Vulkan BC formats to descriptions

VkFormat	**Data Format Specification description**
Formats described in the “S3TC Compressed Texture Image Formats” chapter	
VK_FORMAT_BC1_RGB_UNORM_BLOCK	BC1 with no alpha
VK_FORMAT_BC1_RGB_SRGB_BLOCK	BC1 with no alpha, sRGB-encoded
VK_FORMAT_BC1_RGBA_UNORM_BLOCK	BC1 with alpha
VK_FORMAT_BC1_RGBA_SRGB_BLOCK	BC1 with alpha, sRGB-encoded
VK_FORMAT_BC2_UNORM_BLOCK	BC2
VK_FORMAT_BC2_SRGB_BLOCK	BC2, sRGB-encoded
VK_FORMAT_BC3_UNORM_BLOCK	BC3
VK_FORMAT_BC3_SRGB_BLOCK	BC3, sRGB-encoded
Formats described in the “RGTC Compressed Texture Image Formats” chapter	
VK_FORMAT_BC4_UNORM_BLOCK	BC4 unsigned
VK_FORMAT_BC4_SNORM_BLOCK	BC4 signed
VK_FORMAT_BC5_UNORM_BLOCK	BC5 unsigned
VK_FORMAT_BC5_SNORM_BLOCK	BC5 signed
Formats described in the “BPTC Compressed Texture Image Formats” chapter	
VK_FORMAT_BC6H_UFLOAT_BLOCK	BC6H (unsigned version)
VK_FORMAT_BC6H_SFLOAT_BLOCK	BC6H (signed version)
VK_FORMAT_BC7_UNORM_BLOCK	BC7
VK_FORMAT_BC7_SRGB_BLOCK	BC7, sRGB-encoded

B.2 ETC Compressed Image Formats

The following formats are described in the “ETC2 Compressed Texture Image Formats” chapter of the Khronos Data Format Specification.

Table B.2: Mapping of Vulkan ETC formats to descriptions

VkFormat	**Data Format Specification description**
VK_FORMAT_ETC2_R8G8B8_UNORM_BLOCK	RGB ETC2
VK_FORMAT_ETC2_R8G8B8_SRGB_BLOCK	RGB ETC2 w/ sRGB encoding
VK_FORMAT_ETC2_R8G8B8A1_UNORM_BLOCK	RGB ETC2 w/ punchthrough alpha
VK_FORMAT_ETC2_R8G8B8A1_SRGB_BLOCK	RGB ETC2 w/ punchthrough alpha & sRGB
VK_FORMAT_ETC2_R8G8B8A8_UNORM_BLOCK	RGBA ETC2
VK_FORMAT_ETC2_R8G8B8A8_SRGB_BLOCK	RGBA ETC2 w/ sRGB encoding
VK_FORMAT_EAC_R11_UNORM_BLOCK	Unsigned R11 EAC
VK_FORMAT_EAC_R11_SNORM_BLOCK	Signed R11 EAC
VK_FORMAT_EAC_R11G11_UNORM_BLOCK	Unsigned RG11 EAC
VK_FORMAT_EAC_R11G11_SNORM_BLOCK	Signed RG11 EAC

B.3 ASTC Compressed Image Formats

ASTC formats are described in the “ASTC Compressed Texture Image Formats” chapter of the Khronos Data Format Specification.

Table B.3: Mapping of Vulkan ASTC formats to descriptions

VkFormat	**Block size**	**sRGB output**
VK_FORMAT_ASTC_4x4_UNORM_BLOCK	4×4	No
VK_FORMAT_ASTC_4x4_SRGB_BLOCK	4×4	Yes
VK_FORMAT_ASTC_5x4_UNORM_BLOCK	5×4	No
VK_FORMAT_ASTC_5x4_SRGB_BLOCK	5×4	Yes
VK_FORMAT_ASTC_5x5_UNORM_BLOCK	5×5	No
VK_FORMAT_ASTC_5x5_SRGB_BLOCK	5×5	Yes
VK_FORMAT_ASTC_6x5_UNORM_BLOCK	6×5	No
VK_FORMAT_ASTC_6x5_SRGB_BLOCK	6×5	Yes
VK_FORMAT_ASTC_6x6_UNORM_BLOCK	6×6	No
VK_FORMAT_ASTC_6x6_SRGB_BLOCK	6×6	Yes
VK_FORMAT_ASTC_8x5_UNORM_BLOCK	8×5	No
VK_FORMAT_ASTC_8x5_SRGB_BLOCK	8×5	Yes
VK_FORMAT_ASTC_8x6_UNORM_BLOCK	8×6	No
VK_FORMAT_ASTC_8x6_SRGB_BLOCK	8×6	Yes
VK_FORMAT_ASTC_8x8_UNORM_BLOCK	8×8	No

Table B.3: (continued)

VkFormat	**Block size**	**sRGB output**
VK_FORMAT_ASTC_8x8_SRGB_BLOCK	8×8	Yes
VK_FORMAT_ASTC_10x5_UNORM_BLOCK	10×5	No
VK_FORMAT_ASTC_10x5_SRGB_BLOCK	10×5	Yes
VK_FORMAT_ASTC_10x6_UNORM_BLOCK	10×6	No
VK_FORMAT_ASTC_10x6_SRGB_BLOCK	10×6	Yes
VK_FORMAT_ASTC_10x8_UNORM_BLOCK	10×8	No
VK_FORMAT_ASTC_10x8_SRGB_BLOCK	10×8	Yes
VK_FORMAT_ASTC_10x10_UNORM_BLOCK	10×10	No
VK_FORMAT_ASTC_10x10_SRGB_BLOCK	10×10	Yes
VK_FORMAT_ASTC_12x10_UNORM_BLOCK	12×10	No
VK_FORMAT_ASTC_12x10_SRGB_BLOCK	12×10	Yes
VK_FORMAT_ASTC_12x12_UNORM_BLOCK	12×12	No
VK_FORMAT_ASTC_12x12_SRGB_BLOCK	12×12	Yes

Appendix C

Layers & Extensions

Extensions to the Vulkan API can be defined by authors, groups of authors, and the Khronos Vulkan Working Group. In order not to compromise the readability of the Vulkan Specification, the core Specification does not incorporate most extensions. The online registry of extensions is available at URL

http://www.khronos.org/registry/vulkan/

and allows generating versions of the Specification incorporating different extensions.

Note
We're still sorting through details of extension branches, how they relate to the master spec branch, how spec+multiple extension documents will be generated, and so on. The process isn't fully defined yet.

C.1 Introduction

The Khronos extension registries and extension naming conventions serve several purposes:

- Avoiding naming collisions between extensions developed by mutually unaware parties, both in the extension names themselves, as well as their token, command, and type names.
- Allocating enumerant values for tokens added by extensions
- Creating a defined order between extensions. Extensions with higher numbers may reference extensions with lower numbers, and must define any relevant interactions with lower-numbered extensions.
- Provides a central repository for documentation and header changes associated with extensions

Vulkan's design and general software development trends introduces two new paradigms that require rethinking the existing mechanisms:

- Layers, and with them a focus on a more open ecosystem where non-Khronos members are expected to extend a Khronos API using the Layer mechanism.
- Namespaced constants (enumerations) that don't necessarily draw from a single global set of token values.

C.2 General Rules/Guidelines

Some general rules to simplify the specific rules below:

- Extensions and layers must each have a globally unique name.
- All commands and tokens must have a globally unique name.
- Extensions can expose new commands, types, and/or tokens, but layers must not.
  - However, layers can expose their own extensions, which in turn are allowed to expose new commands and tokens.
- All extensions must be registered with Khronos.
- Extensions must be strictly additive and backwards-compatible. That is, extensions must not remove existing functionality, or change existing default behaviors. A Vulkan implementation may support any combination of extensions, but applications written using only the core API, or a subset of the supported extensions, must continue to work in such an implementation without changes in behavior.

C.3 Extension and Layer Naming Conventions

- Extensions are named with the syntax: `VK_AUTHOR_<name>`.
- Layers are named with the syntax: `VK_LAYER_{AUTHOR|FQDN}_<name>`.

Both extensions and layer names include a `VK_` prefix. In addition, layers add a `LAYER_` prefix. Extension and layer names also contain an *author prefix* identifying the author of the extension/layer. This prefix is a short, capitalized, registered string identifying an author, such as a Khronos member developing Vulkan implementations for their devices, or a non-Khronos developer creating Vulkan layers.

Some authors have platform communities they wish to distinguish between, and can register additional author prefixes for that purpose. For example, Google has separate Android and Chrome communities.

Details on how to register an author prefix are provided below. Layer authors not wishing to register an author prefix with Khronos can instead use a fully-qualified domain name (FQDN) as the prefix. The FQDN should be a domain name owned by the author. FQDNs cannot be used for extensions, only for layers.

- The following are examples of extension and layer names, demonstrating the above syntax:
  - Extension names all use the base prefix `VK_`.
  - Khronos-ratified extensions add the special author prefix `KHR`, and will use the prefix `VK_KHR_`.
  - The following author prefixes are reserved and must not be used:
    * `VK` - To avoid confusion with the top-level `VK_` prefix.
    * `VULKAN` - To avoid confusion with the name of the Vulkan API.
    * `LAYER` - To avoid confusion with the higher-level "LAYER" prefix.
    * `KHRONOS` - To avoid confusion with the Khronos organization.
  - Multi-author extensions that have not been ratified by Khronos (those developed via cooperation between, and intended to be supported by two or more registered authors) add the special author prefix `EXT` to the base prefix, and will use the prefix `VK_EXT_`.

- Traditional author-specific extensions developed by one author (or one author in cooperation with non-authors) add the author prefix to the base prefix. For example, NVIDIA will use the prefix `VK_NV_`, and Valve will use the prefix `VK_VALVE_`. Some authors can have additional registered author prefixes for special purposes. For example, an Android extension developed by Google - but part of an Android open-source community project, and so not a proprietary Google extension - will use the prefix `VK_ANDROID_`.
- Layer names follow the same conventions as extensions, but use the base prefix `VK_LAYER_`.
- Because layers need not be registered with Khronos, an alternative prefix mechanism is needed to allow creating unique layer names without registering an author prefix. Layer authors that prefer not to register an author prefix can instead use a fully-qualified domain name (FQDN) in reverse-order as an author prefix, using all lower-case characters, and replacing `.` (period) with _ (underscore) characters. For example, a layer written by the owner of www.3dxcl.com would use the prefix `VK_LAYER_com_3dxcl_www_`. FQDNs must be encoded in UTF-8.

C.4 Extension Command, Token, and Type Naming Conventions

Extensions may add new commands, tokens, and types, or collectively "objects" to the Vulkan API. These objects also require globally unique names. These names shall conform to the following template:

```
vk<CommandName><ExtensionAuthorPrefix>
```

Where `ExtensionAuthorPrefix` is equal to that of the prefixes defined above for extension names with the exception that a leading _ (underscore) is added for non-Khronos extensions. For example, a Khronos-blessed extension could expose the following command:

```
void vkDoSomethingKHR(void);
```

A Google extension could expose the following command:

```
void vkDoSomethingGOOGLE(void);
```

A multi-author extension could expose the following type:

```
typedef struct VkSomeDataEXT;
```

And a non-Khronos extension could expose this enumerant:

```
enum VkSomeValuesGRPHX {
    VK_SOME_VALUE_0_GRPHX = 0,
    VK_SOME_VALUE_1_GRPHX = 1,
    VK_SOME_VALUE_2_GRPHX = 2,
};
```

C.5 Registering an Author Prefix with Khronos

Previous Khronos APIs could only officially be modified by Khronos members. In an effort to build a more flexible platform, Vulkan allows non-Khronos developers to extend and modify the API via layers and extensions in the same manner as Khronos members. However, for both technical and non-technical reasons, extensions must still be registered with Khronos. Therefore, a mechanism for non-members to register layers and extensions is provided.

Extension authors will be able to create an account on the Khronos github vulkan project and, using this account, register an author prefix with Khronos. This string must be used as the author prefix in any extensions the user registers. The same account will be used to request registration of extensions or layers with Khronos, as described below.

To reserve an author prefix, propose a merge request against `src/spec/vk.xml` in the master branch of the vulkan project, adding a `<tag>` entry and filling in the `name`, `author` and `contact` attributes. The prefix will be reserved once this merge request is accepted into the master branch.

C.6 Registering Extensions and Layers

Extensions must be registered with Khronos. Layers may be registered, and registration is strongly recommended. Registration means receiving an extension number; having the extension or layer name appear on the Khronos registry website and link to some associated documentation hosted on Khronos; and, for extensions including any new objects added by the extension, added to the official XML specification of the Vulkan API hosted by Khronos, used to generate vulkan.h among other purposes.

Registration for Khronos members will be handled using the existing ad-hoc mechanism of filing a bug or pull request on the internal bugzilla or gitlab instance pointing to the associated extension spec. This will be a portion of the Vulkan XML file reserved for extensions, reducing the workload on the registry maintainer from importing a text spec document to the XML spec to essentially just validating and accepting a merge, and reporting the extension number back to the authors.

Since the above process could in theory be completely automated, this suggests a scalable mechanism for accepting registration of non-Khronos extensions. Non-Khronos members who want to create extensions must register with Khronos by creating a github account, and registering their author prefix and/or FQDNs to that account. They can then submit new extension registration requests by proposing merges to the *master* branch of the Vulkan repository in the form of modifications to the `vk.xml` file defining the registry. On acceptance of the merge, the extension will be registered, though its specification need not be checked into the github repository at that point.

The registration process will be split into several steps to accommodate extension number assignment prior to extension publication:

- Acquire an extension number. This is done by proposing a merge request against `vk.xml` similarly to how author prefixes are reserved. The merge should add a new `<extension>` tag at the end of the file with attributes specifying the proposed extension `name`, the next unused sequential extension `number`, the `author` and `contact` information (if different than that already specified for the author prefix used in the extension name), and finally, specifying `supported="disabled"`. The extension number will be reserved only once this merge request is accepted into the master branch.
- Develop & test the extension using the registered extension number.
- Publish the extension to Khronos using the previously registered extension number, by creating a branch of the repository with appropriate changes relative to the master branch.
- Mark the extension as enabled, by proposing a merge to master changing the `supported` attribute value of the `<extension>` to `supported="vulkan"`. This should be completely automated and under the control of the publishers, to allow them to align publication on Khronos with product releases. However, complete automation might be difficult, since steps such as regenerating and validating vulkan.h are involved. Once the merge is accepted and the corresponding updated header with the new extension interface is committed to the master branch, publication is complete.

This will require significant investment in infrastructure to support the process on the Khronos servers.

C.7 Documenting Extensions

Extensions are documented as modifications to the Vulkan specification. These modifications will be on Git branches that are named with the following syntax: `<extension-name>_<spec_version>`

For example, the VK_LUNARG_debug_report extension will be documented relative to version 1.0 of the Vulkan specification. As such, the branch name will be: `VK_LUNARG_debug_report_1_0`

If the extension modifies an existing section of the Vulkan specification, those modifications are made in-place. Since the changes are on a branch, the core-only specification can be easily produced. A specification with an extension is created by merging in the extension's branch contents.

Whenever a new version of the Vulkan API is released, and whenever a new extension is registered, Khronos will create a "full" branch, which contains all extension merged in. For example, after version 1.0 is released, a `Vulkan-1-0-full` branch will be created.

Extensions should be merged according to their registered extension number. If two extensions both modify the same portion of the specification, the higher-numbered extension should take care to deal with any conflicts.

The WSI extensions are a short-term exception to the branch-per-extension rule. There are 10 tightly-coupled extensions (e.g. VK_KHR_surface) that are temporarily living on one branch while automated tools are created for merging branches.

The WSI extensions were used to help pioneer what should be done for extensions. This includes the following:

- All extensions should add to the appendix of the Vulkan specification. This should be modeled after what was done for the VK_KHR_surface extension, which contains some high-level information about the extension (as well as code examples, and revision history) in the `appendices/vk_khr_surface.txt` file.
- Each extension's appendix file is included by adding an `include` statement to the `vkspec.txt` file. Since most extensions will all put their `include` line at the same place in this file, they should add this statement on the master branch, even though the file won't actually exist on the master branch. This will avoid merge conflicts when multiple extensions' branches are merged in order to create the "full" branch specification.
- If there are any other places where 2 or more extensions will extend the Vulkan specification, it is best to put that content in a file, and use an `include` statement to put that content into the spec. Again, this `include` line should be put on the master branch in order to avoid merge conflicts.
- If an extension is more of an addition to the Vulkan specification, the extension should add a chapter to the Vulkan specification.

C.8 Assigning Extension Token Values

Extensions can define their own enumeration types and assign any values to their enumerants that they like. Each enumeration has a private namespace, so collisions are not a problem. However, when extending existing enumeration object with new values, care must be taken to preserve global uniqueness of values. Enumerations which define new bitfields are treated specially as described in Reserving Bitfield Values below.

Each extension is assigned a range of values that can be used to create globally-unique enum values. Most values will be negative numbers, but positive numbers are also reserved. The ability to create both positive and negative extension values is necessary to enable extending enumerations such as VkResult that assign special meaning to negative and positive values. Therefore, 1000 positive and 1000 negative values are reserved for each extension.

Typically, an extension will use a unique offset for each enumeration constant it adds, yielding 1000 values per extension. Since each enumeration object has its own namespace, if an extension needs to add many enumeration constant values, it can reuse offsets on a per-type basis.

The information needed to add new values to the XML are as follows:

- The **extension** name that is adding the new enumeration constant (e.g. "vk_ext_khr_swapchain").
- The existing enumeration **type** being extended (e.g. VkStructureType).
- The name of the new enumeration **token** being added (e.g. VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR).
- The **offset**, which is an integer between 0 and 999 relative to the base being used for the extension.
- The **direction** may be specified to indicate a negative value (dir="-") when needed for negative VkResult values indicating errors, like VK_ERROR_SURFACE_LOST_KHR. The default direction is positive, if not specified.

Implicit is the registered number of an extension, which is used to create a block of unused values offset against a global extension base value. Individual enumerant values are calculated as offsets in that block. Values are calculated as follows:

- base_value = 1000000000
- block_size = 1000
- enum_offset(extension_number,offset) = base_value
  (extension_number - 1) * block_size + offset
- Positive values: enum_offset(extension_number,offset)
- Negative values: -enum_offset(extension_number,offset)

The exact syntax for specifying extension enumerant values is defined in the *readme.pdf* specifying the format of `vk.xml`, and extension authors can also refer to existing extensions for examples.

C.9 Required Extension Tokens

In addition to any tokens specific to the functionality of an extension, all extensions must define two additional tokens.

- VK_EXTNAME_SPEC_VERSION is an integer constant which is the revision of the extension named "VK_extname" in vulkan.h. This value begins at 1 with the initial version of an extension specification, and is incremented when significant changes (bugfixes or added functionality) are made. Note that the revision of an extension defined in vulkan.h and the revision supported by the Vulkan implementation (the *`specVersion`* field of the `VkExtensionProperties` structure corresponding to the extension and returned by one of the Extension Queries) may differ. In such cases, only the functionality and behavior of the lowest-numbered revision can be used.
- VK_EXTNAME_EXTENSION_NAME is a string constant which is the name of the extension.

For example, for the WSI extension VK_KHR_surface, at the time of writing the following definitions were in effect:

```
#define VK_KHR_SURFACE_SPEC_VERSION 24
#define VK_KHR_SURFACE_EXTENSION_NAME "VK_KHR_surface"
```

C.9.1 Reserving Bitfield Values

Enumerants which define bitfield values are a special case, since there are only a small number of unused bits available for extensions. For core Vulkan API and KHR extension bitfield types, reservations must be approved by a vote of the Vulkan Working Group. For EXT and vendor extension bitfield types, reservations must be approved by the listed contact of the extension. Bits are not reserved, and must not be used in a published implementation or specification until the reservation is published as part of the API registry (`vk.xml`) on the Khronos website.

Note
In reality the approving authority for EXT and vendor extension bitfield additions will probably be the owner of the github branch containing the specification of that extension; however, until the github process is fully defined and locked down, it's safest to refer to the listed contact.

C.10 Extension Interactions

Extensions modifying the behavior of existing commands should provide additional parameters by using the param:pNext field of an existing structure, pointing to a new structure defined by the extension, as described in the Valid Usage section. Extension structures defined by multiple extensions affecting the same structure can be chained together in this fashion.

It is in principle possible for extensions to provide additional parameters through alternate means, such as passing a handle parameter to a structure with a *`sType`* defined by the extension, but this approach is discouraged and should not be used.

When chaining multiple extensions to a structure, the implementation will process the chain starting with the base parameter and proceeding through each successive chained structure in turn. Extensions should be defined to accept any order of chaining, and must define their interactions with other extensions such that the results are deterministic. If an extension needs a specific ordering of its extension structure with respect to other extensions in a chain to provide deterministic results, it must define the required ordering and expected behavior as part of its specification.

Appendix D

Invariance

The Vulkan specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different Vulkan implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

D.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of Vulkan commands. For any given Vulkan and framebuffer state vector, and for any Vulkan command, the resulting Vulkan and framebuffer state must be identical whenever the command is executed on that initial Vulkan and framebuffer state. This repeatability requirement doesn't apply when using shaders containing side effects (image and buffer variable stores and atomic operations), because these memory operations are not guaranteed to be processed in a defined order.

One purpose of repeatability is avoidance of visual artifacts when a doublebuffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

D.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different Vulkan mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- "Erasing" a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

D.3 Invariance Rules

For a given instantiation of an Vulkan rendering context:

Rule 1 *For any given Vulkan and framebuffer state vector, and for any given Vulkan command, the resulting Vulkan and framebuffer state must be identical each time the command is executed on that initial Vulkan and framebuffer state.*

Rule 2 *Changes to the following state values have no side effects (the use of any other state value is not affected by the change):*

Required:

- *Framebuffer contents (all bitplanes)*
- *The color buffers enabled for writing*
- *Scissor parameters (other than enable)*
- *Writemasks (color, depth, stencil)*
- *Clear values (color, depth, stencil)*

Strongly suggested:

- *Stencil Parameters (other than enable)*
- *Depth test parameters (other than enable)*
- *Blend parameters (other than enable)*
- *Logical operation parameters (other than enable)*
- *Pixel storage state*

Corollary 1 *Fragment generation is invariant with respect to the state values marked with * in Rule 2.*

Rule 3 *The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it.*

Corollary 2 *Images rendered into different color buffers sharing the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.*

Rule 4 *The same vertex or fragment shader will produce the same result when run multiple times with the same input. The wording "the same shader" means a program object that is populated with the same SPIR-V binary, which is used to create pipelines, possibly multiple times, and which program object is then executed using the same Vulkan state vector. Invariance is relaxed for shaders with side effects, such as performing stores or atomics.*

Rule 5 *All fragment shaders that either conditionally or unconditionally assign* `FragCoord`.z *to* `FragDepth` *are depth-invariant with respect to each other, for those fragments where the assignment to* `FragDepth` *actually is done.*

If a sequence of Vulkan commands specifies primitives to be rendered with shaders containing side effects (image and buffer variable stores and atomic operations), invariance rules are relaxed. In particular, rule 1, corollary 2, and rule 4 do not apply in the presence of shader side effects.

The following weaker versions of rules 1 and 4 apply to Vulkan commands involving shader side effects:

Rule 6 *For any given Vulkan and framebuffer state vector, and for any given Vulkan command, the contents of any framebuffer state not directly or indirectly affected by results of shader image or buffer variable stores or atomic operations must be identical each time the command is executed on that initial Vulkan and framebuffer state.*

Rule 7 *The same vertex or fragment shader will produce the same result when run multiple times with the same input as long as:*

- *shader invocations do not use image atomic operations;*
- *no framebuffer memory is written to more than once by image stores, unless all such stores write the same value; and*
- *no shader invocation, or other operation performed to process the sequence of commands, reads memory written to by an image store.*

When any sequence of Vulkan commands triggers shader invocations that perform image stores or atomic operations, and subsequent Vulkan commands read the memory written by those shader invocations, these operations must be explicitly synchronized.

D.4 Tessellation Invariance

When using a program containing tessellation evaluation shaders, the fixed-function tessellation primitive generator consumes the input patch specified by an application and emits a new set of primitives. The following invariance rules are intended to provide repeatability guarantees. Additionally, they are intended to allow an application with a carefully crafted tessellation evaluation shader to ensure that the sets of triangles generated for two adjacent patches have identical vertices along shared patch edges, avoiding "cracks" caused by minor differences in the positions of vertices along shared edges.

Rule 1 *When processing two patches with identical outer and inner tessellation levels, the tessellation primitive generator will emit an identical set of point, line, or triangle primitives as long as the active program used to process the patch primitives has tessellation evaluation shaders specifying the same tessellation mode, spacing, vertex order, and point mode decorations. Two sets of primitives are considered identical if and only if they contain the same number and type of primitives and the generated tessellation coordinates for the vertex numbered m of the primitive numbered n are identical for all values of m and n.*

Rule 2 *The set of vertices generated along the outer edge of the subdivided primitive in triangle and quad tessellation, and the tessellation coordinates of each, depends only on the corresponding outer tessellation level and the spacing decorations in the tessellation shaders of the pipeline.*

Rule 3 *The set of vertices generated when subdividing any outer primitive edge is always symmetric. For triangle tessellation, if the subdivision generates a vertex with tessellation coordinates of the form (0, x, 1-x), (x, 0, 1-x), or (x, 1-x, 0), it will also generate a vertex with coordinates of exactly (0, 1-x, x), (1-x, 0, x), or (1-x, x, 0), respectively. For quad tessellation, if the subdivision generates a vertex with coordinates of (x, 0) or (0, x), it will also generate a vertex with coordinates of exactly (1-x, 0) or (0, 1-x), respectively. For isoline tessellation, if it generates vertices at (0, x) and (1, x) where x is not zero, it will also generate vertices at exactly (0, 1-x) and (1, 1-x), respectively.*

Rule 4 *The set of vertices generated when subdividing outer edges in triangular and quad tessellation must be independent of the specific edge subdivided, given identical outer tessellation levels and spacing. For example, if vertices at (x, 1 - x, 0) and (1-x, x, 0) are generated when subdividing the w = 0 edge in triangular tessellation, vertices must be generated at (x, 0, 1-x) and (1-x, 0, x) when subdividing an otherwise identical v = 0 edge. For quad tessellation, if vertices at (x, 0) and (1-x, 0) are generated when subdividing the v = 0 edge, vertices must be generated at (0, x) and (0, 1-x) when subdividing an otherwise identical u = 0 edge.*

Rule 5 *When processing two patches that are identical in all respects enumerated in rule 1 except for vertex order, the set of triangles generated for triangle and quad tessellation must be identical except for vertex and triangle order. For each triangle n1 produced by processing the first patch, there must be a triangle n2 produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in n1.*

Rule 6 *When processing two patches that are identical in all respects enumerated in rule 1 other than matching outer tessellation levels and/or vertex order, the set of interior triangles generated for triangle and quad tessellation must be identical in all respects except for vertex and triangle order. For each interior triangle n1 produced by processing* *the first patch, there must be a triangle n2 produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in n1. A triangle produced by the tessellator is considered an interior triangle if none of its vertices lie on an outer edge of the subdivided primitive.*

Rule 7 *For quad and triangle tessellation, the set of triangles connecting an inner and outer edge depends only on the inner and outer tessellation levels corresponding to that edge and the spacing decorations.*

Rule 8 *The value of all defined components of* **`TessellationCoord`** *will be in the range [0, 1]. Additionally, for any defined component x of* **`TessellationCoord`**, *the results of computing 1.0-x in a tessellation evaluation shader will be exact. If any floating-point values in the range [0, 1] fail to satisfy this property, such values must not be used as tessellation coordinate components.*

Appendix E

Credits

Vulkan 1.0 is the result of contributions from many people and companies participating in the Khronos Vulkan Working Group, as well as input from the Vulkan Advisory Panel.

Members of the Working Group, including the company that they represented at the time of their contributions, are listed below. Some specific contributions made by individuals are listed together with their name.

- Alon Or-bach, Samsung Electronics
- Andrew Garrard, Samsung Electronics
- Andrew Woloszyn, Google
- Antoine Labour, Google
- Aras Pranckevičius, Unity
- Bill Licea-Kane, Qualcomm
- Brent E. Insko, Intel
- Chang-Hyo Yo, Samsung Electronics
- Chia-I Wu, LunarG
- Christophe Riccio, Unity
- Cody Northrop, LunarG
- Courtney Goeltzenleuchter, LunarG
- Dan Ginsburg, Valve
- Daniel Johnston, Intel
- Daniel Koch, NVIDIA
- Daniel Rakos, AMD
- David Neto, Google
- Dominik Witczak, AMD

- Graeme Leese, Broadcom
- Graham Connor, Imagination Technologies
- Graham Sellers, AMD
- Ian Elliott, LunarG
- Ian Romanick, Intel
- James Jones, NVIDIA
- Jan-Harald Fredriksen, ARM
- Jeff Bolz, NVIDIA (extensive reviewing and editing to close out many issues)
- Jeff Vigil, Qualcomm Technologies, Inc.
- Jens Owen, LunarG
- Jeremy Hayes, LunarG
- Jesse Barker, ARM
- Jesse Hall, Google
- John Kessenich, Independent
- John McDonald, Valve
- Jon Ashburn, LunarG
- Jon Leech, Independent (XML toolchain, normative language)
- Jonas Gustavsson, Sony Mobile
- Jungwoo Kim, Samsung Electronics
- Kenneth Benzie, Codeplay Software Ltd.
- Krzysztof Iwanicki, Samsung Electronics
- Mark Callow
- Mark Lobodzinski, LunarG
- Mateusz Przybylski, Intel
- Mathias Schott, NVIDIA
- Maurice Ribble, Qualcomm Technologies, Inc.
- Michael Lentine, Google
- Michael Worcester, Imagination Technologies
- Michal Pietrasiuk, Intel
- Mika Isojarvi, Google
- Mike Stroyan, LunarG
- Neil Henning, Codeplay Software Ltd.

- Neil Trevett, NVIDIA
- Nick Penwarden, Epic Games
- Norbert Nopper, Freescale
- Patrick Doane, Blizzard Entertainment
- Pierre Boudier, NVIDIA
- Pierre-Loup A. Griffais, Valve
- Piers Daniell, NVIDIA
- Prabindh Sundareson, Samsung Electronics
- Pyry Haulos, Google
- Ray Smith, ARM
- Robert J. Simpson, Qualcomm Technologies, Inc.
- Rolando Caloca Olivares, Epic Games
- Slawomir Cygan, Intel
- Slawomir Grajewski, Intel
- Stefanus Du Toit, Google
- Stuart Smith, Imagination Technologies
- Timothy Lottes, AMD
- Tobias Hector, Imagination Technologies (validity language and toolchain)
- Tobin Ehlis, LunarG
- Tom Olson, ARM (Working Group Chair)
- Tomasz Kubale, Intel
- Tony Barbour, LunarG
- Yanjun Zhang, Vivante

In addition to the Working Group, the Vulkan Advisory Panel members provided important real-world usage information and advice that helped guide design decisions.

Administrative support to the Working Group was provided by members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark. Technical support was provided by James Riordon, webmaster of Khronos.org and OpenGL.org.