Vulkan™ Programming Guide

The Official Guide to Learning Vulkan

Graham Sellers

With contributions from John Kessenich

Vulkan™ Programming Guide

for the command from memory. This is the stage that fetches those parameters.

- `VK_PIPELINE_STAGE_VERTEX_INPUT_BIT`: This is the stage where vertex attributes are fetched from their respective buffers. After this, content of vertex buffers can be overwritten, even if the resulting vertex shaders have not yet completed execution.
- `VK_PIPELINE_STAGE_VERTEX_SHADER_BIT`: This stage is passed when all vertex shader work resulting from a drawing command is completed.
- `VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT`: This stage is passed when all tessellation control shader invocations produced as the result of a drawing command have completed execution.
- `VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT`: This stage is passed when all tessellation evaluation shader invocations produced as the result of a drawing command have completed execution.
- `VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT`: This stage is passed when all geometry shader invocations produced as the result of a drawing command have completed execution.
- `VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT`: This stage is passed when all fragment shader invocations produced as the result of a drawing command have completed execution. Note that there is no way to know that a primitive has been completely rasterized while the resulting fragment shaders have not yet completed. However, rasterization does not access memory, so no information is lost here.
- `VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT`: All per-fragment tests that might occur *before* the fragment shader is launched have completed.
- `VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT`: All per-fragment tests that might occur *after* the fragment shader is executed have completed. Note that outputs to the depth and stencil attachments happen as part of the test, so this stage and the early fragment test stage include the depth and stencil outputs.
- `VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT`: Fragments produced by the pipeline have been written to the color attachments.

- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT: Compute shader invocations produced as the result of a dispatch have completed.
- VK_PIPELINE_STAGE_TRANSFER_BIT: Any pending transfers triggered as a result of calls to **vkCmdCopyImage()** or **vkCmdCopyBuffer()**, for example, have completed.
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT: All operations considered to be part of the graphics pipeline have completed.
- VK_PIPELINE_STAGE_HOST_BIT: This pipeline stage corresponds to access from the host.
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT: When used as a destination, this special flag means that any pipeline stage may access memory. As a source, it's effectively equivalent to VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT.
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT: This stage is the big hammer. Whenever you just don't know what's going on, use this; it will synchronize everything with everything. Just use it wisely.

Because the flags specified in srcStageMask and dstStageMask are used to indicate *when* things happen, it's acceptable for Vulkan implementations to move them around or interpret them in various ways. The srcStageMask specifies when the source stage has finished reading or writing a resource. As a result, moving the effective position of that stage later in the pipeline doesn't change the fact that those accesses have completed; it may mean only that the implementation waits longer than it really needs to for them to complete.

Likewise, the dstStageMask specifies the point at which the pipeline will wait before proceeding. If an implementation moves that wait point earlier, that will still work. The event that it waits on will still have completed when the logically later parts of the pipeline begin execution. That implementation just misses the opportunity to perform work when it was instead waiting.

The dependencyFlags parameter specifies a set of flags that describes how the dependency represented by the barrier affects the resources referenced by the barrier. The only defined flag is VK_DEPENDENCY_BY_REGION_BIT, which indicates that the barrier affects only the region modified by the source stages (if it can be determined), which is consumed by the destination stages.

A single call to **vkCmdPipelineBarrier()** can be used to trigger many barrier operations. There are three types of barrier operations: global memory barriers, buffer barriers, and image barriers. Global memory barriers affect things such as synchronized access to mapped memory between the host and the device. Buffer and image barriers primarily affect device access to buffer and image resources, respectively.

Global Memory Barriers

The number of global memory barriers to be triggered by **vkCmdPipelineBarrier()** is specified in memoryBarrierCount. If this is nonzero, then pMemoryBarriers points to an array of memoryBarrierCount VkMemoryBarrier structures, each defining a single memory barrier. The definition of VkMemoryBarrier is

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

The sType field of VkMemoryBarrier should be set to VK_STRUCTURE_TYPE_MEMORY_BARRIER, and pNext should be set to nullptr. The only other fields in the structure are the source and destination access masks specified in srcAccessMask and dstAccessMask, respectively. The access masks are bitfields containing members of the VkAccessFlagBits. The source access mask specifies how the memory was last written, and the destination access mask specifies how the memory will next be read. The available access flags are

- VK_ACCESS_INDIRECT_COMMAND_READ_BIT: The memory referenced will be the source of commands in an indirect drawing or dispatch command such as **vkCmdDrawIndirect()** or **vkCmdDispatchIndirect()**.
- VK_ACCESS_INDEX_READ_BIT: The memory referenced will be the source of index data in an indexed drawing command such as **vkCmdDrawIndexed()** or **vkCmdDrawIndexedIndirect()**.
- VK_ACCESS_VERTEX_ATTRIBUTE_READ_BIT: The memory referenced will be the source of vertex data fetched by Vulkan's fixed-function vertex assembly stage.
- VK_ACCESS_UNIFORM_READ_BIT: The memory referenced is the source of data for a uniform block accessed by a shader.

- VK_ACCESS_INPUT_ATTACHMENT_READ_BIT: The memory referenced is used to back an image used as an input attachment.
- VK_ACCESS_SHADER_READ_BIT: The memory referenced is used to back an image object that is read from using image loads or texture reads in a shader.
- VK_ACCESS_SHADER_WRITE_BIT: The memory referenced is used to back an image object that is written to using image stores in a shader.
- VK_ACCESS_COLOR_ATTACHMENT_READ_BIT: The memory referenced is used to back an image used as a color attachment where reads are performed, perhaps because blending is enabled. Note that this is not the same as an input attachment, where data is read explicitly by the fragment shader.
- VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT: The memory referenced is used to back an image used as a color attachment that will be written to.
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT: The memory referenced is used to back an image used as a depth or stencil attachment that will be read from because the relevant test is enabled.
- VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT: The memory referenced is used to back an image used as a depth or stencil attachment that will be written to because the relevant write mask is enabled.
- VK_ACCESS_TRANSFER_READ_BIT: The memory referenced is used as the source of data in a transfer operation such as **vkCmdCopyImage()**, **vkCmdCopyBuffer()**, or **vkCmdCopyBufferToImage()**.
- VK_ACCESS_TRANSFER_WRITE_BIT: The memory referenced is used as the destination of a transfer operation.
- VK_ACCESS_HOST_READ_BIT: The memory referenced is mapped and will be read from by the host.
- VK_ACCESS_HOST_WRITE_BIT: The memory referenced is mapped and will be written to by the host.
- VK_ACCESS_MEMORY_READ_BIT: All other memory reads not explicitly covered by the preceding cases should specify this bit.
- VK_ACCESS_MEMORY_WRITE_BIT: All other memory writes not explicitly covered by the preceding cases should specify this bit.

Memory barriers provide two important pieces of functionality. First, they help avoid hazards, and second, they help ensure data consistency.

A *hazard* occurs when read and write operations are reordered relative to the order in which the programmer expects them to execute. They can be very hard to diagnose because they are often platform- or timing-dependent. There are three types of hazards:

- A *read-after-write*, or RaW, hazard occurs when the programmer expects to read from a piece of memory that has recently been written to and that those reads will *see* the results of the writes. If the read is rescheduled and ends up executing before the write is complete, the read will see old data.
- A *write-after-read*, or WaR, hazard occurs when a programmer expects to overwrite a piece of memory that had previously been read by another part of the program. If the write operation ends up being scheduled before the read operation, then the read operation will see the new data, not the older data it was expecting.
- A *write-after-write*, or WaW, hazard occurs when a programmer expects to overwrite the same location in memory multiple times and that only the results of the last write will be visible to subsequent readers. If the writes are rescheduled with respect to one another, then only the result of the write that happened to execute last will be visible to readers.

There is no such thing as a read-after-read hazard because no data is modified.

In the memory barrier, the source isn't necessarily a producer of data but the first operation that is protected by that barrier. For avoiding RaW hazards, the source is actually a read operation.

For example, to ensure that all texture fetches are complete before overwriting an image with a copy operation, we need to specify `VK_ACCESS_SHADER_READ_BIT` in the `srcAccessMask` field and `VK_ACCESS_TRANSFER_WRITE_BIT` in the `dstAccessMask` field. This tells Vulkan that the first stage is reading from an image in a shader and that the second stage may overwrite that image, so we should not reorder the copy into the image before any shaders that may have read from it.

Note that there is some overlap between the bits in VkAccessFlagBits and those in VkPipelineStageFlagBits. The VkAccessFlagBits flags specify *what* operation is being performed, and the VkPipelineStageFlagBits describe *where* in the pipeline the action is performed.

The second piece of functionality provided by the memory barrier is to ensure consistency of the views of data from different parts of the pipeline. For example, if an application contains a shader that writes to a buffer from a shader and then needs to read that data back from the buffer by mapping the underlying memory object, it should specify VK_ACCESS_SHADER_WRITE_BIT in srcAccessMask and VK_ACCESS_HOST_READ_BIT in dstAccessMask. If there are caches in the device that may buffer writes performed by shaders, those caches may need to be flushed in order for the host to see the results of the write operations.

Buffer Memory Barriers

Buffer memory barriers provide finer-grained control of the memory used to back buffer objects. The number of buffer memory barriers executed by a call to **vkCmdPipelineBarrier()** is specified in the bufferMemoryBarrierCount parameter, and the pBufferMemoryBarriers field is a pointer to an array of this many VkBufferMemoryBarrier structures, each defining a buffer memory barrier. The definition of VkBufferMemoryBarrier is

```
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 sType field of each VkBufferMemoryBarrier structure should be set to VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER, and pNext should be set to nullptr. The srcAccessMask and dstAccessMask fields have the same meanings as they do in the VkMemoryBarrier structure. Obviously, some of the flags that refer specifically to images, such as color or depth attachments, have little meaning when dealing with buffer memory.