A series of vkCmdxxx() calls are meant to run “flat-out”, that is, as fast as the Vulkan runtime can get them executing. But, many times, that is not desirable because the output of one command might be needed as the input to a subsequent command.

Pipeline Barriers solve this problem by declaring which stages of the hardware pipeline in subsequent vkCmdyyy() calls need to wait until which stages in previous vkCmdxxx() calls are completed.

Why Do We Need Pipeline Barriers?

Potential Memory Race Conditions that Pipeline Barriers can Prevent

1. Read-after-Write (R-a-W) – the memory write in one operation starts overwriting the memory that another operation’s read needs to use.
2. Write-after-Read (W-a-R) – the memory read in one operation hasn’t yet finished before another operation starts overwriting that memory.
3. Write-after-Write (W-a-W) – two operations start overwriting the same memory and the end result is non-deterministic.

Note: there is no problem with Read-after-Read (R-a-R) as no data gets changed.

These are the Commands that could be entered into a Command Buffer, I

vkCmdBeginConditionalRendering
vkCmdBeginDebugUtilsLabel
vkCmdBeginQuery
vkCmdBeginQueryIndexed
vkCmdBeginRendering
vkCmdBeginRenderPass
vkCmdBeginRenderPass2
vkCmdBindDescriptorSets
vkCmdBindIndexBuffer
vkCmdBindInvocationMask
vkCmdBindPipeline
vkCmdBindPipelineShaderGroup
vkCmdBindShadingRateImage
vkCmdBindTransformFeedbackBuffers
vkCmdBindVertexBuffers
vkCmdBindVertexBuffers2
vkCmdBlitImage
These are the Commands that could be entered into a Command Buffer, II

- vkCmdCopyQueryPoolResults
- vkCmdDispatch
- vkCmdDispatchBase
- vkCmdDispatchIndirect
- vkCmdDraw
- vkCmdDrawIndexed
- vkCmdDrawIndexedIndirect
- vkCmdDrawIndexedIndirectCount
- vkCmdDrawIndirect
- vkCmdDrawIndirectByteCount
- vkCmdNextSubpass
- vkCmdPipelineBarrier

These are the Commands that could be entered into a Command Buffer, III

- vkCmdPreprocessGeneratedCommands
- vkCmdPushConstants
- vkCmdPushDescriptorSet
- vkCmdPushDescriptorSetWithTemplate
- vkCmdResolveImage
- vkCmdResolveImage2
- vkCmdSetBlendConstants
- vkCmdSetSampleLocations
- vkCmdSetStencilReference
- vkCmdSetViewPort
- vkCmdSetViewport

These are the Commands that could be entered into a Command Buffer, IV

- vkCmdSetSampleLocations
- vkCmdSetScissor
- vkCmdSetViewportWithCount
- vkCmdSetViewportShadingRatePalette
- vkCmdDisableDepthBias
- vkCmdDisableDepthBiasEnable
- vkCmdDisableDepthBounds
- vkCmdDisableDepthCompareOp

vkCmdPipelineBarrier( ) Function Call

A Pipeline Barrier is a way to establish a dependency between commands that were submitted before the barrier and commands that are submitted after the barrier.

vkCmdPipelineBarrier( commandBuffer, 
srcStageMask, dstStageMask, 
MemoryBarrierCount, pBufferMemoryBarriers, 
imageViewBarrierCount, pImageMemoryBarriers, 

vkCmdPipelineBarrier( commandBuffer, 
srcStageMask, dstStageMask, 
VK_DEPENDENCY_BY_REGION_BIT, 
memoryBarrierCount, pBufferMemoryBarriers, 
imageViewMemoryBarrierCount, pImageMemoryBarriers, 

The hope is to maximize the number of unblocked stages: produce data early and consume data late.
The Scenario

1. The cross-streets are named after pipeline stages
2. All traffic lights start out green
3. There are special sensors at all intersections that will know when any car in the src group is in that intersection
4. There are connections from those sensors to the traffic lights so that when any car in the src group is in the intersection, the proper dst traffic lights will be turned red
5. When the last car in the src group completely makes it through its intersection, the proper dst traffic lights are turned back to green
6. The Vulkan command pipeline ordering is this: (1) the src cars get released by the previous vkCmdxxx, (2) the pipeline barrier is invoked (which turns some lights red), (3) the dst cars get released by the next vkCmdyyy, (4) the dst cars stop at the red light, (5) the src cars clear the intersection, (6) the dst lights turn green, (6) the dst cars continue.

Pipeline Stage Masks – Where in the Pipeline is this Memory Data being Generated or Consumed?

- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_VERTEX_INPUT_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_HOST_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
Access Masks –
What are you interested in generating or consuming this memory for?

- VK_ACCESS INDIRECT_COMMAND_READ_BIT
- VK_ACCESS INDEX_READ_BIT
- VK_ACCESS VERTEX_ATTRIBUTE_READ_BIT
- VK_ACCESS UNIFORM_READ_BIT
- VK_ACCESS INPUT_ATTACHMENT_READ_BIT
- VK_ACCESS_SHADER_READ_BIT
- VK_ACCESS_SHADER_WRITE_BIT
- VK_ACCESS COLOR_ATTACHMENT_READ_BIT
- VK_ACCESS COLOR_ATTACHMENT_WRITE_BIT
- VK_ACCESS DEPTH_STENCIL_ATTACHMENT_READ_BIT
- VK_ACCESS DEPTH_STENCIL_ATTACHMENT_WRITE_BIT
- VK_ACCESS TRANSFER_READ_BIT
- VK_ACCESS TRANSFER_WRITE_BIT
- VK_ACCESS HOST_READ_BIT
- VK_ACCESS MEMORY_READ_BIT
- VK_ACCESS MEMORY_WRITE_BIT

Pipeline Stages and what Access Operations are Allowed

- VK_PIPELINE_STAGE_ALL_COMMANDS_BIT
- VK_PIPELINE_STAGE_ALL_GRAPHICS_BIT
- VK_PIPELINE_STAGE_HOST_BIT
- VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT
- VK_PIPELINE_STAGE_TRANSFER_BIT
- VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT
- VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT
- VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT
- VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT
- VK_PIPELINE_STAGE_GEOMETRY_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_EVALUATION_SHADER_BIT
- VK_PIPELINE_STAGE_TESSELLATION_CONTROL_SHADER_BIT
- VK_PIPELINE_STAGE_VERTEX_SHADER_BIT
- VK_PIPELINE_STAGE_DRAW_INDIRECT_BIT
- VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT

Example #1: Be sure we are done writing an output image before using it as a Fragment Shader Texture

Access types

- src
- dst
Example #1: The Scenario

src cars are generating the image

dst cars are waiting to use that image as a texture

Example #2: Setting a Pipeline Barrier so the Drawing Waits for the Compute Shader to Finish

```
VkBufferMemoryBarrier
vbmb;
vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
vbmb.pNext = nullptr;
vbmb.srcAccessFlags = VK_ACCESS_SHADER_WRITE_BIT;
vbmb.dstAccessFlags = VK_ACCESS_SHADER_READ_BIT;
vbmb.srcQueueFamilyIndex = 0;
vbmb.dstQueueFamilyIndex = 0;
vbmb.buffer = vbmb.offset = 0;
vbmb.size = NUM_PARTICLES * sizeof(glm::vec4);
const uint32 bufferMemoryBarrierCount = 1;
vkCmdPipelineBarrier(
    commandBuffer,
    VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
    VK_DEPENDENCY_BY_REGION_BIT,
    0, nullptr, bufferMemoryBarrierCount, IN &vbmb, 0, nullptr);
```

Example #2: Setting a Pipeline Barrier so the Compute Shader Waits for the Drawing to Finish

```
VkBufferMemoryBarrier
vbmb;
vbmb.sType = VK_STRUCTURE_TYPE_BUFFER_MEMORY_BARRIER;
vbmb.pNext = nullptr;
vbmb.srcAccessFlags = VK_ACCESS_SHADER_WRITE_BIT;
vbmb.dstAccessFlags = VK_ACCESS_SHADER_READ_BIT;
vbmb.srcQueueFamilyIndex = 0;
vbmb.dstQueueFamilyIndex = 0;
vbmb.buffer = vbmb.offset = 0;
vbmb.size = NUM_PARTICLES * sizeof(glm::vec4);
const uint32 bufferMemoryBarrierCount = 1;
vkCmdPipelineBarrier(
    commandBuffer,
    VK_PIPELINE_STAGE_VERTEX_SHADER_BIT,
    VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
    VK_DEPENDENCY_BY_REGION_BIT,
    0, nullptr, bufferMemoryBarrierCount, IN &vbmb, 0, nullptr);
```