Vulkan.

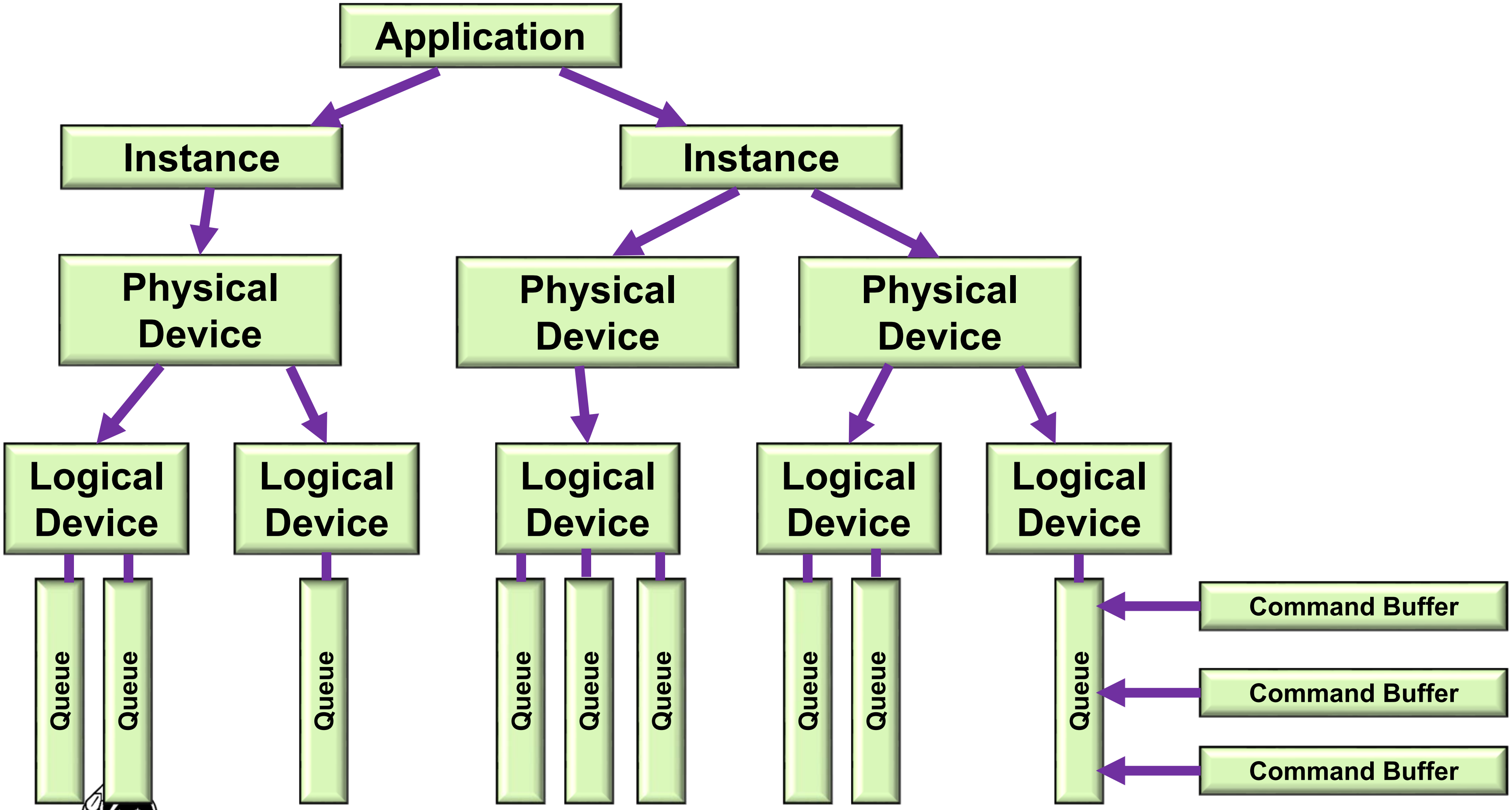
Queues and Command Buffers

Oregon State
University

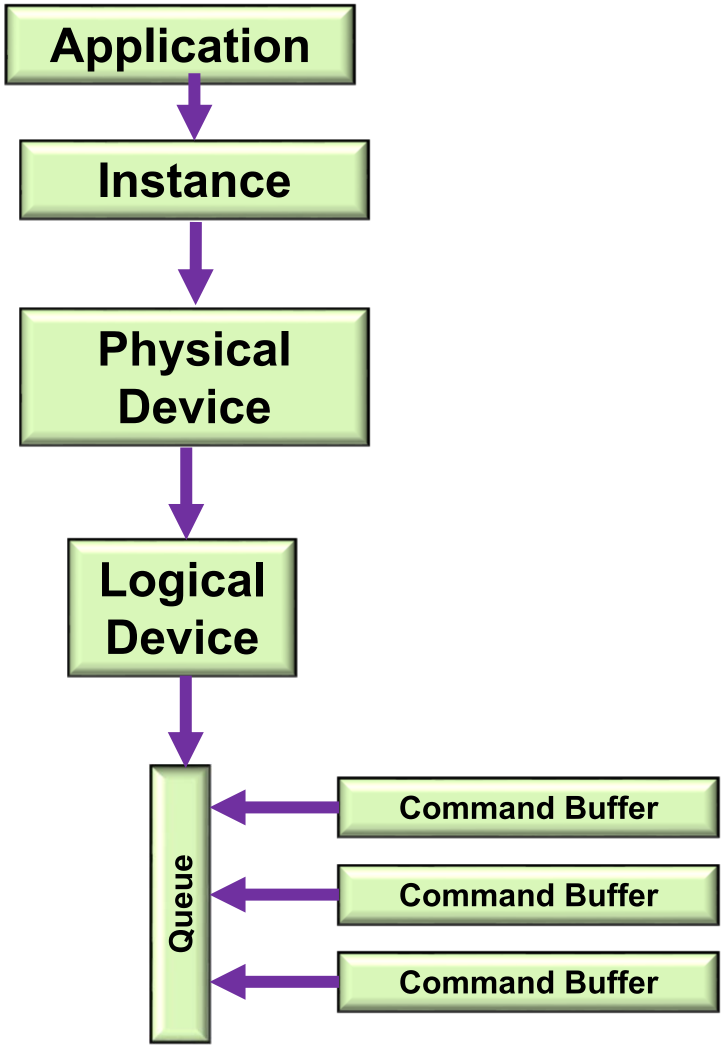
Mike Bailey

mjb@cs.oregonstate.edu

Vulkan: Overall Block Diagram



Vulkan: a More Typical (and Simplified) Block Diagram



Application

Instance

Physical Device

Logical Device

Queue

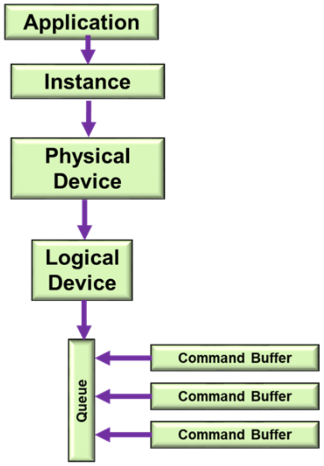
Command Buffer

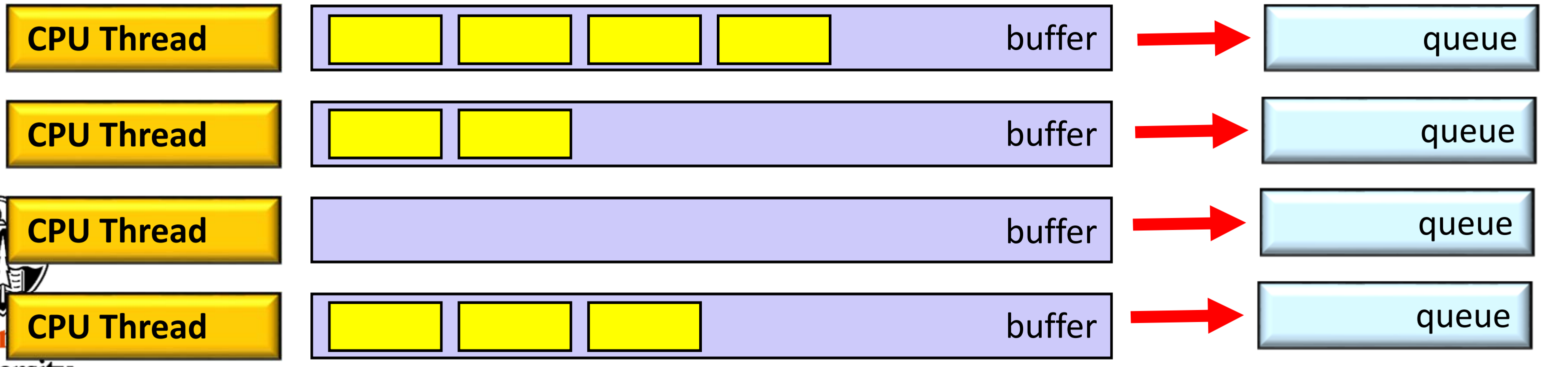
Command Buffer

Command Buffer

Vulkan Queues and Command Buffers

- Graphics commands are recorded in command buffers, e.g., *vkCmdDoSomething(cmdBuffer, …);*
- You can have as many simultaneous Command Buffers as you want
- Each command buffer can be filled from a different thread
- Command Buffers record our commands, but no work takes place until a Command Buffer is submitted to a Queue
- We don't create Queues – the Logical Device has them already
- Each Queue belongs to a Queue Family
- We don't create Queue Families – the Physical Device already has them


Application
Instance
Physical Device
Logical Device
Queue
Command Buffer
Command Buffer
Command Buffer


CPU Thread
buffer
queue
CPU Thread
buffer
queue
CPU Thread
buffer
queue
CPU Thread
buffer
queue

Querying what Queue Families are Available

```
uint32_t  count;
vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *) nullptr );

VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
vkGetPhysicalDeviceFamilyProperties(  PhysicalDevice, &count, OUT &vqfp, );

for( unsigned int i = 0; i < count; i++ )
{
	fprintf( FpDebug, "\t%d: Queue Family Count = %2d  ;   ", i, vqfp[i].queueCount );
	if( ( vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )       fprintf( FpDebug, " Graphics" );
	if( ( vqfp[i].queueFlags & VK_QUEUE_COMPUTE_BIT  ) != 0 )       fprintf( FpDebug, " Compute " );
	if( ( vqfp[i].queueFlags & VK_QUEUE_TRANSFER_BIT ) != 0 )       fprintf( FpDebug, " Transfer" );
	fprintf(FpDebug, "\n");
}
```

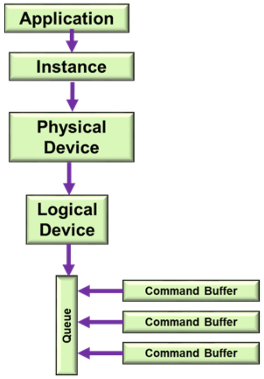
```
Found 3 Queue Families:
	0: Queue Family Count = 16  ;    Graphics Compute Transfer
	1: Queue Family Count =   1  ;    Transfer
	2: Queue Family Count =   8  ;    Compute
```

Similarly, we Can Write a Function that Finds the Proper Queue Family

```
int
FindQueueFamilyThatDoesGraphics( )
{
        uint32_t count = -1;
        vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr )
;
        VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[ count ];
        vkGetPhysicalDeviceQueueFamilyProperties( IN PhysicalDevice, &count, OUT vqfp );

        for( unsigned int i = 0; i < count; i++ )
        {
                if( ( vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT ) != 0 )
                        return i;
        }
        return -1;
}
```

“These are not the Queue Families you’re looking for.”

Creating a Logical Device Queue Needs to Know Queue Family Information



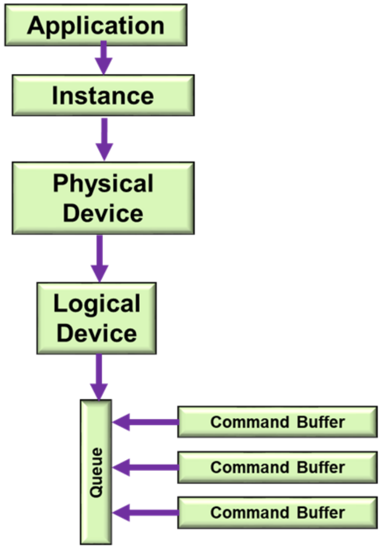
```
 float   queuePriorities[ ] =
{
                1.                    // one entry per queueCount
};

VkDeviceQueueCreateInfo  vdqci[1];
        vdqci.sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO;
        vdqci.pNext = nullptr;
        vdqci.flags = 0;
        vdqci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );
        vdqci.queueCount = 1;
        vdqci.queuePriorities = (float *) queuePriorities;

 VkDeviceCreateInfo   vdci;
            vdci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
            vdci.pNext = nullptr;
            vdci.flags = 0;
            vdci.queueCreateInfoCount = 1;                   // # of device queues wanted
            vdci.pQueueCreateInfos = IN &vdqci[0];           // array of VkDeviceQueueCreateInfo's
            vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
            vdci.ppEnabledLayerNames = myDeviceLayers;
            vdci.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
            vdci.ppEnabledExtensionNames = myDeviceExtensions;
            vdci.pEnabledFeatures = IN &PhysicalDeviceFeatures;     // already created

result = vkCreateLogicalDevice( PhysicalDevice, IN &vdci, PALLOCATOR, OUT &LogicalDevice );

VkQueue Queue;
uint32_t  queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );
uint32_t  queueIndex = 0;

result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex,  OUT &Queue );
```

Creating the Command Pool as part of the Logical Device

```
VkResult
Init06CommandPool( )
{
	VkResult result;

	VkCommandPoolCreateInfo			vcpci;
		vcpci.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
		vcpci.pNext = nullptr;
		vcpci.flags =	VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
				| VK_COMMAND_POOL_CREATE_TRANSIENT_BIT;
#ifdef CHOICES
VK_COMMAND_POOL_CREATE_TRANSIENT_BIT
VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
#endif
		vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics( );

	result = vkCreateCommandPool( LogicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool );

	return result;
}
```

Creating the Command Buffers

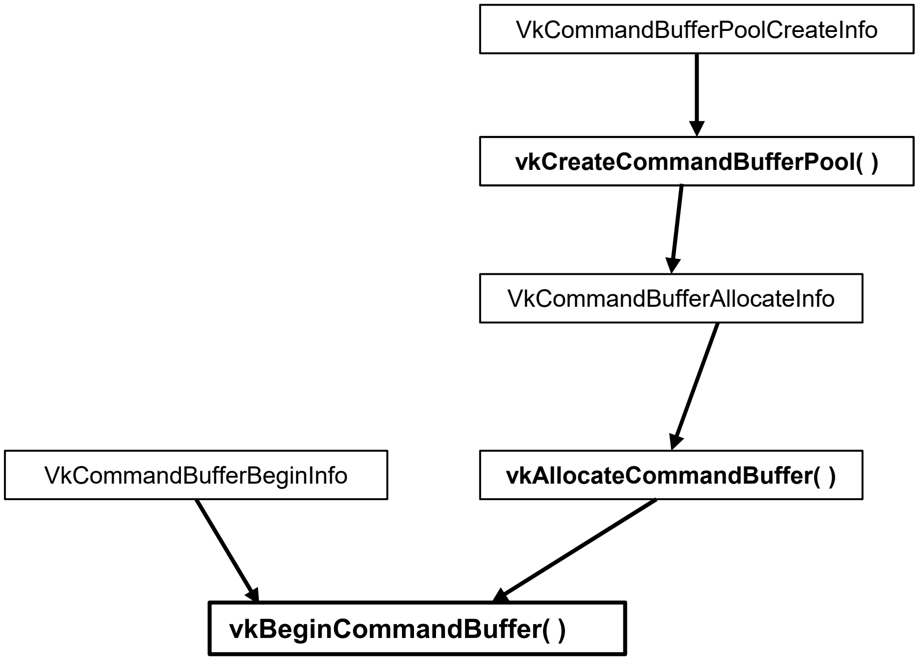
```
VkResult
Init06CommandBuffers( )
{
	VkResult result;

	// allocate 2 command buffers for the double-buffered rendering:

	{
		VkCommandBufferAllocateInfo			vcbai;
			vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
			vcbai.pNext = nullptr;
			vcbai.commandPool = CommandPool;
			vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
			vcbai.commandBufferCount = 2;		// 2, because of double-buffering

		result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &CommandBuffers[0] );
	}


	// allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:

	{
		VkCommandBufferAllocateInfo			vcbai;
			vcbai.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
			vcbai.pNext = nullptr;
			vcbai.commandPool = CommandPool;
			vcbai.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
			vcbai.commandBufferCount = 1;

		result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer );
	}

	return result;
}
```



Beginning a Command Buffer

```
VkSemaphoreCreateInfo                vsci;
        vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
        vsci.pNext = nullptr;
        vsci.flags = 0;

VkSemaphore  imageReadySemaphore;
result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );

uint32_t  nextImageIndex;
vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN UINT64_MAX,
                                        IN imageReadySemaphore, IN VK_NULL_HANDLE, OUT &nextImageIndex );


VkCommandBufferBeginInfo             vcbbi;
            vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
            vcbbi.pNext = nullptr;
            vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
            vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;

result = vkBeginCommandBuffer( CommandBuffers[nextImageIndex], IN &vcbbi );

        . . .

vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
```

Beginning a Command Buffer



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

```
vkCmdBeginQuery( commandBuffer, flags );
vkCmdBeginRenderPass( commandBuffer, const contents );
vkCmdBindDescriptorSets( commandBuffer, pDynamicOffsets );
vkCmdBindIndexBuffer( commandBuffer, indexType );
vkCmdBindPipeline( commandBuffer, pipeline );
vkCmdBindVertexBuffers( commandBuffer, firstBinding, bindingCount, const pOffsets );
vkCmdBlitImage( commandBuffer, filter );
vkCmdClearAttachments( commandBuffer, attachmentCount, const pRects );
vkCmdClearColorImage( commandBuffer, pRanges );
vkCmdClearDepthStencilImage( commandBuffer, pRanges );
vkCmdCopyBuffer( commandBuffer, pRegions );
vkCmdCopyBufferToImage( commandBuffer, pRegions );
vkCmdCopyImage( commandBuffer, pRegions );
vkCmdCopyImageToBuffer( commandBuffer, pRegions );
vkCmdCopyQueryPoolResults( commandBuffer, flags );
vkCmdDebugMarkerBeginEXT( commandBuffer, pMarkerInfo );
vkCmdDebugMarkerEndEXT( commandBuffer );
vkCmdDebugMarkerInsertEXT( commandBuffer, pMarkerInfo );
vvkCmdDispatch( commandBuffer, groupCountX, groupCountY, groupCountZ );
vkCmdDispatchIndirect( commandBuffer, offset );
vkCmdDraw( commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance );
vkCmdDrawIndexed( commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance );
vkCmdDrawIndexedIndirect( commandBuffer, stride );
vkCmdDrawIndexedIndirectCountAMD( commandBuffer, stride );
vkCmdDrawIndirect( commandBuffer, stride );
vkCmdDrawIndirectCountAMD( commandBuffer, stride );
vkCmdEndQuery( commandBuffer, query );
vkCmdEndRenderPass( commandBuffer );
vkCmdExecuteCommands( commandBuffer, commandBufferCount, const pCommandBuffers );
```

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

```
vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );
vkCmdNextSubpass( commandBuffer, contents );
vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers,
bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );
vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );
vkCmdPushDescriptorSetKHR( commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites );
vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );
vkCmdReserveSpaceForCommandsNVX( commandBuffer, pReserveSpaceInfo );
vkCmdResetEvent( commandBuffer, event, stageMask );
vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );
vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );
vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );
vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );
vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );
vkCmdSetEvent( commandBuffer, event, stageMask );
vkCmdSetLineWidth( commandBuffer, lineWidth );
vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );
vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask );
vkCmdSetStencilReference( commandBuffer, faceMask, reference );
vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );
vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );
vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );
vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );
vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers,
                    bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );
```

```
VkResult
RenderScene( )
{
        VkResult result;
        VkSemaphoreCreateInfo                   vsci;
            vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
            vsci.pNext = nullptr;
            vsci.flags = 0;

        VkSemaphore  imageReadySemaphore;
        result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );

        uint32_t nextImageIndex;
        vkAcquireNextImageKHR( LogicalDevice,    IN SwapChain, IN UINT64_MAX, IN VK_NULL_HANDLE,
                                                IN VK_NULL_HANDLE, OUT &nextImageIndex );

        VkCommandBufferBeginInfo              vcbbi;
                vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
                vcbbi.pNext = nullptr;
                vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
                vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;

        result = vkBeginCommandBuffer( CommandBuffers[nextImageIndex], IN &vcbbi );
```

```
VkClearColorValue                       vccv;
        vccv.float32[0] = 0.0;
        vccv.float32[1] = 0.0;
        vccv.float32[2] = 0.0;
        vccv.float32[3] = 1.0;

VkClearDepthStencilValue                vcdsv;
        vcdsv.depth = 1.f;
        vcdsv.stencil = 0;

VkClearValue                            vcv[2];
        vcv[0].color = vccv;
        vcv[1].depthStencil = vcdsv;

VkOffset2D o2d = { 0, 0 };
VkExtent2D e2d = { Width, Height };
VkRect2D r2d = { o2d, e2d };

VkRenderPassBeginInfo                   vrpbi;
        vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
        vrpbi.pNext = nullptr;
        vrpbi.renderPass = RenderPass;
        vrpbi.framebuffer = Framebuffers[ nextImageIndex ];
        vrpbi.renderArea = r2d;
        vrpbi.clearValueCount = 2;
        vrpbi.pClearValues = vcv;            // used for VK_ATTACHMENT_LOAD_OP_CLEAR

vkCmdBeginRenderPass( CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE );
```

```
VkViewport viewport =
{
	0.,			// x
	0.,			// y
	(float)Width,
	(float)Height,
	0.,			// minDepth
	1.			// maxDepth
};

vkCmdSetViewport( CommandBuffers[nextImageIndex], 0, 1, IN &viewport );		// 0=firstViewport, 1=viewportCount

VkRect2D scissor =
{
	0,
	0,
	Width,
	Height
};

vkCmdSetScissor( CommandBuffers[nextImageIndex], 0, 1, IN &scissor );

vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS,
				GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr );
								// dynamic offset count, dynamic offsets
vkCmdBindPushConstants( CommandBuffers[nextImageIndex], PipelineLayout, VK_SHADER_STAGE_ALL, offset, size, void *values );

VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };

VkDeviceSize offsets[1] = { 0 };

vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );		// 0, 1 = firstBinding, bindingCount

const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;
vkCmdDraw( CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance );

vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );

vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
```

Submitting a Command Buffer to a Queue for Execution

```
VkSubmitInfo                              vsi;
            vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
            vsi.pNext = nullptr;
            vsi.commandBufferCount = 1;
            vsi.pCommandBuffers = &CommandBuffer;
            vsi.waitSemaphoreCount = 1;
            vsi.pWaitSemaphores = imageReadySemaphore;
            vsi.signalSemaphoreCount = 0;
            vsi.pSignalSemaphores = (VkSemaphore *)nullptr;
            vsi.pWaitDstStageMask = (VkPipelineStageFlags *)nullptr;
```

The Entire Submission / Wait / Display Process

```
VkFenceCreateInfo                    vfci;
        vfci.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
        vfci.pNext = nullptr;
        vfci.flags = 0;

VkFence  renderFence;
vkCreateFence( LogicalDevice, &vfci, PALLOCATOR, OUT &renderFence );
result = VK_SUCCESS;

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
VkQueue presentQueue;
vkGetDeviceQueue( LogicalDevice, FindQueueFamilyThatDoesGraphics( ), 0, OUT &presentQueue );\
                                                     // 0 =, queueIndex

VkSubmitInfo                         vsi;
        vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
        vsi.pNext = nullptr;
        vsi.waitSemaphoreCount = 1;
        vsi.pWaitSemaphores = &imageReadySemaphore;
        vsi.pWaitDstStageMask = &waitAtBottom;
        vsi.commandBufferCount = 1;
        vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
        vsi.signalSemaphoreCount = 0;
        vsi.pSignalSemaphores = &SemaphoreRenderFinished;

result = vkQueueSubmit( presentQueue,   1, IN &vsi, IN renderFence );    // 1 = submitCount
result = vkWaitForFences( LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX );    // waitAll, timeout

vkDestroyFence( LogicalDevice, renderFence, PALLOCATOR );

VkPresentInfoKHR                         vpi;
        vpi.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
        vpi.pNext = nullptr;
        vpi.waitSemaphoreCount = 0;
        vpi.pWaitSemaphores = (VkSemaphore *)nullptr;
        vpi.swapchainCount = 1;
        vpi.pSwapchains = &SwapChain;
        vpi.pImageIndices = &nextImageIndex;
        vpi.pResults = (VkResult *)nullptr;

result = vkQueuePresentKHR( presentQueue, IN &vpi );
```

Create fence

Get the queue

Fill in the queue information

Submit the queue

Wait for the fence