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, but doesn’t have to be
- Command Buffers record commands, but no work takes place until a Command Buffer is submitted to a Queue
- We don’t create Queues – the Logical Device already has them
- Each Queue belongs to a Queue Family
- We don’t create Queue Families – the Physical Device already has them
### Querying what Queue Families are Available

```c
uint32_t count;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr);
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceFamilyProperties(IN 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");
}
```

For the Nvidia A6000 cards:

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

### Finding the Proper Queue Family

Similarly, we can write a function that finds the proper queue family:

```c
int FindQueueFamilyThatDoesGraphics()
{
    uint32_t count = -1;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, OUT &count, OUT (VkQueueFamilyProperties *)nullptr);
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, IN &count, OUT vqfp);
    for(unsigned int i = 0; i < count; i++)
    {
        if((vqfp[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0)
            return i;
    }
    return -1;
}
```

### Creating a Logical Device Needs to Know Queue Family Information

```c
float queuePriorities[] = {
    1. // one entry per queueCount
};
VkDeviceQueueCreateInfo vdqci[1];
vdqci[0].sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO;
vdqci[0].pNext = nullptr;
vdqci[0].flags = 0;
vdqci[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vdqci[0].queueCount = 1;
vdqci[0].queuePriorities = (float *) queuePriorities;
```

### Creating the Command Pool as part of the Logical Device

```c
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;
    vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
    result = vkCreateCommandPool(LogicalDevice, IN &vcpci, PALLOCATOR, OUT &CommandPool);
    return result;
}
```
Creating the Command Buffers

```c
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[nextImageIndex] );

    // allocate 1 command buffer for the transferring pixels from a staging buffer to a texture buffer:
    vcbai.commandBufferCount = 1;
    result = vkAllocateCommandBuffers( LogicalDevice, IN &vcbai, OUT &TextureCommandBuffer );

    return result;
}
```

Beginning a Command Buffer

```c
// Beginning a Command Buffer – One per Image

vkSemaphoreCreateInfo vsci;

vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

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;

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

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

- vkCmdBeginConditionalRendering
- vkCmdBeginDebugUtilsLabel
- vkCmdBeginQuery
- vkCmdBeginQueryIndexed
- vkCmdBeginRenderPass
- vkCmdBeginRenderPass2
- vkCmdBeginTransformFeedback
- vkCmdBindDescriptorSets
- vkCmdBindIndexBuffer
- vkCmdBindInvocationMask
- vkCmdBindPipeline
- vkCmdBindPipelineShaderGroup
- vkCmdBindShadingRateImage
- vkCmdBindTransformFeedbackBuffers
- vkCmdBindVertexBuffers
- vkCmdBindVertexBuffers2
- vkCmdBlitImage
- vkCmdBlitImage2
- vkCmdBuildAccelerationStructure
- vkCmdBuildAccelerationStructureIndirect
- vkCmdClearAttachments
- vkCmdClearColorImage
- vkCmdClearDepthStencilImage
- vkCmdCopyAccelerationStructure
- vkCmdCopyAccelerationStructureToMemory
- vkCmdCopyBuffer
- vkCmdCopyBuffer2
- vkCmdCopyBufferToImage
- vkCmdCopyBufferToImage2
- vkCmdCopyMemory
- vkCmdCopyMemoryToBuffer
- vkCmdCopyMemoryToBuffer2
- vkCmdCopyMemoryToAccelerationStructure
These are the Commands that could be entered into a Command Buffer, II

- vkCmdCopyQueryPoolResults
- vkCmdCuLaunchKernelX
- vkCmdDebugMarkerBegin
- vkCmdDebugMarkerEnd
- vkCmdDebugMarkerInsert
- vkCmdDispatch
- vkCmdDispatchBase
- vkCmdDispatchIndirect
- vkCmdDraw
- vkCmdDrawIndexed
- vkCmdDrawIndirect
- vkCmdDrawIndexed
- vkCmdDrawIndexedIndirect
- vkCmdDrawIndexedIndirectCount
- vkCmdDrawIndirect
- vkCmdDrawIndirectByteCount
- vkCmdDrawIndirectCount
- vkCmdDrawMeshTasksIndirectCount
- vkCmdDrawMeshTasksIndirect
- vkCmdDrawMeshTasks

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

- vkCmdPreprocessGeneratedCommands
- vkCmdPushConstants
- vkCmdPushDescriptorSet
- vkCmdPushDescriptorSetWithTemplate
- vkCmdResetEvent
- vkCmdResetEvent2
- vkCmdResetQueryPool
- vkCmdResolveImage
- vkCmdResolveImage2
- vkCmdSetBlendConstants
- vkCmdSetCheckpoint
- vkCmdSetCoarseSampleOrder
- vkCmdSetCullMode
- vkCmdSetDepthBias
- vkCmdSetDepthBiasEnable
- vkCmdSetDepthBounds
- vkCmdSetDepthBoundsTestEnable
- vkCmdSetDepthCompareOp
- vkCmdSetDepthTestEnable
- vkCmdSetDepthWriteEnable
- vkCmdSetDeviceMask
- vkCmdSetDiscardRectangle
- vkCmdSetExclusiveScissor
- vkCmdSetFragmentShaderRateEnum
- vkCmdSetFragmentShadingRate
- vkCmdSetFrontFace
- vkCmdSetLineStipple
- vkCmdSetLineWidth
- vkCmdSetLogicOp
- vkCmdSetPatchControlPoints
- vkCmdSetPrimitiveRestartEnable
- vkCmdSetPrimitiveTopology
- vkCmdSetSampleLocations
- vkCmdSetScissor
- vkCmdSetScissorWithCount
- vkCmdSetStencilCompareMask
- vkCmdSetStencilOp
- vkCmdSetStencilReference
- vkCmdSetStencilTestEnable
- vkCmdSetStencilWriteMask
- vkCmdSetVertexInput
- vkCmdSetViewport
- vkCmdSetViewportShadingRatePalette
- vkCmdSetViewportWithCount
- vkCmdSetViewportWScaling
- vkCmdSubpassShading
- vkCmdTraceRaysIndirect
- vkCmdTraceRaysIndirect
- vkCmdTraceRays
- vkCmdUpdateBuffer
- vkCmdWaitEvents
- vkCmdWaitEvents2
- vkCmdWriteAccelerationStructuresProperties
- vkCmdWriteBufferMarker
- vkCmdWriteTimestamp
- vkCmdWriteTimestamp2

How the RenderScene() Function Works

```cpp
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 );
    // Render scene
    // ...
    result = vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
    vkCmdExecuteCommandBuffers( PhysicalDevice, IN CommandBuffers, IN UINT32_COUNT );
    return result;
}
```
```
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);

vkCmdBindPushConstants(CommandBuffers[nextImageIndex], PipelineLayout, VK_SHADER_STAGE_ALL, offset, size, void *values);

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

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

const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);

vkCmdDraw(CommandBuffers[nextImageIndex], vertexCount, instanceCount, firstVertex, firstInstance);

vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);

vkEndCommandBuffer(CommandBuffers[nextImageIndex]);
```

```
提交命令缓冲到队列执行

VSUBMITInfo
vs structure = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vs pNext = nullptr;
vs commandBufferCount = 1;
vsi commandBuffers = CommandBuffer;
vsi waitSemaphoreCount = 1;
vsi pWaitSemaphores = imageReadySemaphore;
vsi signalSemaphoreCount = 0;
vsi pSignalSemaphores = (VkSemaphore *)nullptr;

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);
```
What Happens After a Queue has Been Submitted?

As the Vulkan Specification says:

"Command buffer submissions to a single queue respect submission order and other implicit ordering guarantees, but otherwise may overlap or execute out of order. Other types of batches and queue submissions against a single queue (e.g. sparse memory binding) have no implicit ordering constraints with any other queue submission or batch. Additional explicit ordering constraints between queue submissions and individual batches can be expressed with semaphores and fences."

In other words, the Vulkan driver on your system will execute the commands in a single buffer in the order in which they were put there.

But, between different command buffers submitted to different queues, the driver is allowed to execute commands between buffers in-order or out-of-order or overlapped-order, depending on what it thinks it can get away with.

The message here is, I think, always consider using some sort of Vulkan synchronization when one command depends on a previous command reaching a certain state first.