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 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
Querying what Queue Families are Available

```c
uint32_t count;
vkGetPhysicalDeviceQueueFamilyProperties(IN PhysicalDevice, &count, OUT (VkQueueFamilyProperties *)nullptr);
VkQueueFamilyProperties *vqfp = new VkQueueFamilyProperties[count];
vkGetPhysicalDeviceQueueFamilyProperties(PhysicalDevice, &count, OUT &vqfp);
for(unsigned int i = 0; i < count; i++)
{
    fprintf(FpDebug, "%d: Queue Family Count = %2d ; 
        
    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, 

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

```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;
VkDeviceCreateInfo vdci;
vdc[i].sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdc[i].pNext = nullptr;
vdc[i].flags = 0;
vdc[i].queueCreateInfoCount = 1; // # of device queues wanted
vdci.pQueueCreateInfos = IN &vdqci[0]; // array of VkDeviceQueueCreateInfo's
```

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;
    #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

```cpp
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:
    {
        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

```cpp
void vkBeginCommandBuffer( )
{
    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 );
}
```

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

```c
VkResult
vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );

RenderScene( )

VkResult result;

bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);

vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );

vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;

vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );

vkCmdPushDescriptorSetKHR( commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites );

vkCmdPushDescriptorSetWithTemplateKHR( commandBuffer, descriptorUpdateTemplate, layout, set, pData );

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

vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );

vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );

vkCmdSetDepthBounds( commandBuffer, minDepthBounds, maxDepthBounds );

vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );

vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );

vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask) ;

vkCmdSetStencilReference( commandBuffer, faceMask, reference );

vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );

vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );

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

vkCmdResetEvent( commandBuffer, event, stageMask );

vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );

vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );

uint32_t nextImageIndex;

vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );

vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );

vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask) ;

vkCmdSetStencilReference( commandBuffer, faceMask, reference );

vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );

vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );

vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );

vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );

vsci.pNext = nullptr;vsci.flags = 0;

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

vkCmdBindDescriptorSets( CommandBuffers[nextImageIndex], VK_PIPELINE_BIND_POINT_GRAPHICS, GraphicsPipelineLayout, 0, 4, DescriptorSets, 0, (uint32_t *)nullptr );

VkRect2D r2d = { o2d, e2d };

vkCmdBindPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );

const uint32_t vertexCount = (int)VertexData.size();

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

vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );

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

---

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

```c
VkClearColorValue vccv;

vccv.float32[0] = 0.0;

vccv.float32[1] = 0.0;

vccv.float32[2] = 0.0;

vccv.float32[3] = 1.0;

vkCmdSetViewport( CommandBuffers[nextImageIndex], 0, 1, IN &viewport );

vkCmdClearDepthStencilValue vcdsv;

vcdsv.depth = 1.f;

vcdsv.stencil = 0;

vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets );

const uint32_t vertexCount = (int)VertexData.size();

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

---

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

```c
const uint32_t vertexCount = (int)VertexData.size();

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

vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );
```

---

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

```c
vkCmdEndCommandBuffer( CommandBuffers[nextImageIndex] );
```
Submitting a Command Buffer to a Queue for Execution

```c
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;
```

Submitting a Command Buffer to a Queue for Execution

```c
result = vkQueueSubmit(presentQueue, 1, IN &vsi, IN renderFence);
result = vkWaitForFences(LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX);
vkDestroyFence(LogicalDevice, renderFence, PALLOCATOR);
```

The Entire Submission / Wait / Display Process

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

What Happens After a Queue has Been Submitted?

As the Vulkan 1.1 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.