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
Querying what Queue Families are Available

```c
uint32_t count;
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++) {
    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

```c
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 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[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

```cpp
vkBeginCommandBuffer( )
VkCommandBufferBeginInfo
vkAllocateCommandBuffer( )
VkCommandBufferAllocateInfo
vkCreateCommandBufferPool( )
VkCommandBufferPoolCreateInfo
```

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

```cpp
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 );
vkCmdDispatch( 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

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

RenderScene( )
```

```c
vkCmdNextSubpass( commandBuffer, contents );
```

```c
vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );
```

```c
VkSemaphoreCreateInfo vsci;
```

```c
vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );
```

```c
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
```

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

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

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

```c
uint32_t nextImageIndex;
```

```c
vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );
```

```c
vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );
```

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

```c
vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );
```

```c
vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );
```

```c
vkCmdSetEvent( commandBuffer, event, stageMask );
```

```c
vkCmdSetLineWidth( commandBuffer, lineWidth );
```

```c
vkCmdSetScissor( commandBuffer, firstScissor, scissorCount, pScissors );
```

```c
vkCmdSetStencilCompareMask( commandBuffer, faceMask, compareMask);
```

```c
vkCmdSetStencilReference( commandBuffer, faceMask, reference );
```

```c
vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );
```

```c
vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );
```

```c
vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );
```

```c
vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );
```

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

```c
VkClearColorValue vccv;
```

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

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

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

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

```c
VkClearValue vcv[2];
```

```c
vcv[0].color = vccv;
vcv[1].depthStencil = vcdsv;
```

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

```c
VkRect2D r2d = { o2d, e2d };
```

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

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

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

```c
vkCmdEndRenderPass( CommandBuffers[nextImageIndex] );
```

```c
vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
```

```
```
```
```
The Entire Submission / Wait / Display Process

1. **Create fence**
   - `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;`

2. **Get the queue**
   - `VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;`
   - `VkQueue presentQueue;`
   - `vkGetDeviceQueue(LogicalDevice, FindQueueFamilyThatDoesGraphics(), 0, OUT &presentQueue);`

3. **Fill in the queue information**
   - `VkSubmitInfo vsi;`
   - `vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;`
   - `vsi.pNext = nullptr;`
   - `vsi.waitSemaphoreCount = 1;`
   - `vsi.pWaitSemaphores = &imageReadySemaphore;`
   - `vsi.waitDstStageMask = &waitAtBottom;`
   - `vsi.commandBufferCount = 1;`
   - `vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];`
   - `vsi.signalSemaphoreCount = 0;`
   - `vsi.pSignalSemaphores = &SemaphoreRenderFinished;`

4. **Submit the queue**
   - `result = vkQueueSubmit(presentQueue, 1, IN &vsi, IN renderFence);` - `// 1 = submitCount`
   - `result = vkWaitForFences(LogicalDevice, 1, IN &renderFence, VK_TRUE, UINT64_MAX);` - `// waitAll, timeout`

5. **Wait for the fence**
   - `vkDestroyFence(LogicalDevice, renderFence, PALLOCATOR);`

6. **Present to window**
   - `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;`

7. **Present to window**
   - `result = vkQueuePresentKHR(presentQueue, IN &vpi);` - `// IN &vpi`