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;
```

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;
vqci.pNext = nullptr;
vqci.flags = 0;
vqci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vqci.queueCount = 1;
vqci.queuePriorities = (float*)queuePriorities;
```

```
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;
vqci.pNext = nullptr;
vqci.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT
    #ifdef CHOICES
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT
    #endif
    vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
    result = vkCreateCommandPool(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:
    {
        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

```c
vkCreateSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;
    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore( LogicalDevice, IN &vsci, allocator, 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] );
```

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 );`
- `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

1. `vkResult vkCmdFillBuffer( commandBuffer, dstBuffer, dstOffset, size, data );`
2. `vkCmdNextSubpass( commandBuffer, contents );`
3. `vkCmdPipelineBarrier( commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, VkMemoryBarrier* pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );`
4. `vkCmdProcessCommandsNVX( commandBuffer, pProcessCommandsInfo );`
5. `vkCmdPushConstants( commandBuffer, layout, stageFlags, offset, size, pValues );`
6. `vkCmdPushDescriptorSetKHR( commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites );`
7. `vkAcquireNextImageKHR( LogicalDevice, IN SwapChain, IN UINT64_MAX, IN VK_NULL_HANDLE, &imageReadySemaphore );`
8. `vkCmdReserveSpaceForCommandsNVX( commandBuffer, pReserveSpaceInfo );`
9. `vkCmdResetEvent( commandBuffer, event, stageMask );`
10. `vkCmdResetQueryPool( commandBuffer, queryPool, firstQuery, queryCount );`
11. `vkCmdResolveImage( commandBuffer, srcImage, srcImageLayout, dstImage, dstImageLayout, regionCount, pRegions );`
12. `vkCmdSetBlendConstants( commandBuffer, blendConstants[4] );`
13. `vkCmdSetDepthBias( commandBuffer, depthBiasConstantFactor, depthBiasClamp, depthBiasSlopeFactor );`
14. `vkCmdSetDeviceMaskKHX( commandBuffer, deviceMask );`
15. `vkCmdSetDiscardRectangleEXT( commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles );`
16. `vkCmdSetStenciling( commandBuffer, faceMask, compareMask );`
17. `vkCmdSetStencilReference( commandBuffer, faceMask, reference );`
18. `vkCmdSetStencilWriteMask( commandBuffer, faceMask, writeMask );`
19. `vkCmdSetViewport( commandBuffer, firstViewport, viewportCount, pViewports );`
20. `vkCmdSetViewportWScalingNV( commandBuffer, firstViewport, viewportCount, pViewportWScalings );`
21. `vkCmdUpdateBuffer( commandBuffer, dstBuffer, dstOffset, dataSize, pData );`
22. `vkCmdWaitEvents( commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers );`
23. `vkCmdWriteTimestamp( commandBuffer, pipelineStage, queryPool, query );`
Submitting a Command Buffer to a Queue for Execution

```cpp
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

```cpp
VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &imageReadySemaphore;
vsi.signalSemaphoreCount = 0;
vsi.pSignalSemaphores = (VkSemaphore *)nullptr;
```

The Entire Submission / Wait / Display Process

```cpp
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);

VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &imageReadySemaphore;
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);
```