Queues and Command Buffers

Vulkan: Overall Block Diagram

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

Vulkan: a More Typical (and Simplified) Block Diagram

- Application
- Instance
- Physical Device
- Logical Device
- Queue
- 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
Querying what Queue Families are Available

```cpp
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

```cpp
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

```cpp
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.ppQueueCreateInfos = IN &vdqci[0]; // array of VkDeviceQueueCreateInfo's
vdci.ppEnabledLayerNames = myDeviceLayers;
vdci.ppEnabledExtensionNames = myDeviceExtensions;
vdci.pEnabledFeatures = IN &PhysicalDeviceFeatures; // already created

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

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

```cpp
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_RESET_COMMAND_BUFFER_BIT
    VK_COMMAND_POOL_CREATE_TRANSIENT_BIT
    #endif
    vcpci.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
    result = vkCreateCommandPool(IN 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[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
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);
```

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

1. `vkResult result;`
2. `uint32_t nextImageIndex;`
3. `vkAcquireNextImageKHR(LogicalDevice, SwapChain, UINT64_MAX, VK_NULL_HANDLE, VK_NULL_HANDLE, &nextImageIndex);`
4. `vkCmdFillBuffer(commandBuffer, dstBuffer, dstOffset, size, data);`
5. `VkCommandBufferBeginInfo vcbbi;`
6. `vcbbi.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;`
7. `vcbbi.pNext = nullptr;`
8. `vcbbi.flags = VK_COMMAND_BUFFER_USAGE_ONE TIME_SUBMIT_BIT;`
9. `vcbbi.pInheritanceInfo = (VkCommandBufferInheritanceInfo *)nullptr;`
10. `vkCmdProcessCommandsNVX(commandBuffer, pProcessCommandsInfo);`
11. `result = vkBeginCommandBuffer(commandBuffer);`
12. `vkCmdNextSubpass(commandBuffer, contents);`
13. `vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);`
14. `vkCmdBindDescriptorSets(commandBuffer, pipelineBindPoint, layout, set, descriptorWriteCount, pDescriptorWrites);`
15. `vkCmdBindBuffer(commandBuffer, buffer, offset, size, data);`
16. `vkCmdBindBuffer(commandBuffer, buffer, offset, size, data);`
The Entire Submission / Wait / Display Process

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

vkDestroyFence(LogicalDevice, renderFence, PALLOCATOR);
```