Queues and Command Buffers

Vulkan: Overall Block Diagram

Application

Instance

Physical Device

Logical Device

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 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

CPU Thread

CPU Thread

CPU Thread

CPU Thread

Command Buffer

Command Buffer

Command Buffer

Command Buffer

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

```
VkDeviceQueueCreateInfo vdqci[1] = {
    .queueCount = 1,
    .queueFamilyIndex = FindQueueFamilyThatDoesGraphics(),
    .queuePriorities = (float *)&queuePriorities[0],
    .flags = 0,
    .pNext = nullptr,
};
```

Creating the Command Pool as part of the Logical Device

```
VkCommandPoolCreateInfo vcpci = {
    .queueFamilyIndex = FindQueueFamilyThatDoesGraphics(),
    .flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT | VKCOMMAND_POOL_CREATE_TRANSIENT_BIT;
    .pNext = nullptr;
};
```

Beginning a Command Buffer

```
VkCommandBufferBeginInfo vcbbi = {
    .pNext = nullptr;
    .flags = 0;
};
```

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

```
- vkCmdBeginQuery(commandBuffer, flags);
- vkCmdEndQuery(commandBuffer, query);
- vkCmdEndRenderPass(commandBuffer);
- vkCmdDispatchIndirect(commandBuffer, offset);
- vkCmdCopyBufferToImage(commandBuffer, pRegions);
- vkCmdCopyImage(commandBuffer, pRegions);
- vkCmdCopyImageToBuffer(commandBuffer, pRegions);
- vkCmdCopyQueryPoolResults(commandBuffer, flags);
- vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);
- vkCmdDebugMarkerEndEXT(commandBuffer);
- vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);
- vkCmdClearColorImage(commandBuffer, pRanges);
- vkCmdClearDepthStencilImage(commandBuffer, pRanges);
- vkCmdClearAttachments(commandBuffer, attachmentCount, const pRects);
- vkCmdBlitImage(commandBuffer, filter);
- vkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);
- vkCmdBindPipeline(commandBuffer, pipeline);
- vkCmdBeginQuery(commandBuffer, flags);
- vkCmdDrawIndexedIndirectCountAMD(commandBuffer, stride);
- vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);
- vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);
- vkCmdCopyBufferToImage(commandBuffer, pRegions);
```
These are the Commands that could be entered into the Command Buffer, II

- vkCmdSetStencilCompareMask
- vkCmdSetViewport
- vkCmdSetStencilWriteMask
- vkCmdSetScissor
- vkCmdSetLineWidth
- vkCmdSetDiscardRectangleEXT
- vkCmdSetDeviceMaskKHX
- vkCmdSetViewportWScalingNV
- vkCmdSetDepthBounds
- vkCmdReserveSpaceForCommandsNVX
- vkCmdPushDescriptorSetWithTemplateKHR
- vkCmdPushDescriptorSetKHR
- vkCmdProcessCommandsNVX
- vkCmdFillBuffer
- vkCmdSetBlendConstants
- vkCmdResolveImage
- vkCmdResetQueryPool
- vkCmdResetEvent
- vkCmdPipelineBarrier

```
VkRenderPassBeginInfo vrpbi;
VkExtent2D e2d = { Width, Height };
VkOffset2D o2d = { 0, 0 };
VkClearValue vcv[2];
VkClearDepthStencilValue vcdsv;
vcv[1].depthStencil = vcdsv;
vcv[0].color = vccv;
vcdsv.stencil = 0;
vcdsv.depth = 1.f;
vccv.float32[3] = 1.0;
vccv.float32[2] = 0.0;
vccv.float32[1] = 0.0;
vrpbi.pClearValues = vcv;               // used for VK_ATTACHMENT_LOAD_OP_CLEAR
vrpbi.clearValueCount = 2;
vrpbi.renderArea = r2d;

VkSubmitInfo vsi;

mjb – September 10, 2019
```
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.