Queues and Command Buffers

Vulkan: Overall Block Diagram

Application

Instance

Instance

Physical Device

Physical Device

Physical Device

Logical Device

Logical Device

Logical Device

Logical Device

Logical Device

Queue

Queue

Queue

Queue

Queue

Queue

Command Buffer

Command Buffer

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.

Vulkan: a More Typical (and Simplified) Block Diagram
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, "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(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 a Logical Device Queue Needs to Know Queue Family Information

```c
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;
vdc.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdc.pNext = nullptr;
vdc.flags = 0;
vdc.queueCreateInfoCount = 1; // # of device queues wanted
vdci.pQueueCreateInfos = &vdqci[0]; // array of VkDeviceQueueCreateInfo's
vdci.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char*);
vdci.ppEnabledLayerNames = myDeviceLayers;
vdc.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char*);
vdc.ppEnabledExtensionNames = myDeviceExtensions;
vdc.pEnabledFeatures = &PhysicalDeviceFeatures; // already created
result = vkCreateLogicalDevice(PhysicalDevice, &vdci, PALLOCATOR, OUT &LogicalDevice);
VkQueue Queue;
uint32_t queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
uint32_t queueIndex = 0;
result = vkGetDeviceQueue(LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue);
```

Creating the Command Pool as part of the Logical Device

```c
VkResult Init06CommandPool() {
    VkResult result;
    VkCommandPoolCreateInfo vcppi;
    vcppi.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    vcppi.pNext = nullptr;
    vcppi.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
    vcppi.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
    result = vkCreateCommandPool(LogicalDevice, &vcppi, 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
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);
```
Beginning a Command Buffer

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

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

- `vkCmdBeginQuery(commandBuffer, flags);`
- `vkCmdBeginRenderPass(commandBuffer, const contents);`
- `vkCmdBindDescriptorSets(commandBuffer, pDynamicOffsets);`
- `vkCmdBindDescriptorSets(commandBuffer, pDynamicOffsets);`
- `vkCmdBindPipeline(commandBuffer, pipeline);`
- `vkCmdBindPipeline(commandBuffer, pipeline);`
- `vkCmdBindPipeline(commandBuffer, pipeline);`
- `vkCmdBindIndexBuffer(commandBuffer, indexType);`
- `vkCmdBindIndexBuffer(commandBuffer, indexType);`
- `vkCmdBindIndexBuffer(commandBuffer, indexType);`
- `vkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);`
- `vkCmdBindVertexBuffers(commandBuffer, firstBinding, bindingCount, const pOffsets);`
- `vkCmdBlitImage(commandBuffer, filter);`
- `vkCmdBlitImage(commandBuffer, filter);`
- `vkCmdClearAttachments(commandBuffer, attachmentCount, const pRects);`
- `vkCmdClearAttachments(commandBuffer, attachmentCount, const pRects);`
- `vkCmdClearColorImage(commandBuffer, pRanges);`
- `vkCmdClearColorImage(commandBuffer, pRanges);`
- `vkCmdClearDepthStencilImage(commandBuffer, pRanges);`
- `vkCmdClearDepthStencilImage(commandBuffer, pRanges);`
- `vkCmdCopyBuffer(commandBuffer, pRegions);`
- `vkCmdCopyBuffer(commandBuffer, pRegions);`
- `vkCmdCopyBufferToImage(commandBuffer, pRegions);`
- `vkCmdCopyBufferToImage(commandBuffer, pRegions);`
- `vkCmdCopyQueryPoolResults(commandBuffer, flags);`
- `vkCmdCopyQueryPoolResults(commandBuffer, flags);`
- `vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);`
- `vkCmdDebugMarkerBeginEXT(commandBuffer, pMarkerInfo);`
- `vkCmdDebugMarkerEndEXT(commandBuffer);`
- `vkCmdDebugMarkerEndEXT(commandBuffer);`
- `vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);`
- `vkCmdDebugMarkerInsertEXT(commandBuffer, pMarkerInfo);`
- `vkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);`
- `vkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);`
- `vkCmdDispatch(commandBuffer, groupCountX, groupCountY, groupCountZ);`
- `vkCmdDispatchIndirect(commandBuffer, offset);`
- `vkCmdDispatchIndirect(commandBuffer, offset);`
- `vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);`
- `vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);`
- `vkCmdDraw(commandBuffer, vertexCount, instanceCount, firstVertex, firstInstance);`
- `vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);`
- `vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);`
- `vkCmdDrawIndexed(commandBuffer, indexCount, instanceCount, firstIndex, int32_t vertexOffset, firstInstance);`
- `vkCmdDrawIndexedIndirect(commandBuffer, stride);`
- `vkCmdDrawIndexedIndirect(commandBuffer, stride);`
- `vkCmdDrawIndexedIndirect(commandBuffer, stride);`
- `vkCmdDrawIndexedIndirectCountAMD(commandBuffer, stride);`
- `vkCmdDrawIndexedIndirectCountAMD(commandBuffer, stride);`
- `vkCmdDrawIndexedIndirectCountAMD(commandBuffer, stride);`
- `vkCmdDrawIndirect(commandBuffer, stride);`
- `vkCmdDrawIndirect(commandBuffer, stride);`
- `vkCmdDrawIndirect(commandBuffer, stride);`
- `vkCmdDrawIndirectCount(commandBuffer, stride);`
- `vkCmdDrawIndirectCount(commandBuffer, stride);`
- `vkCmdDrawIndirectCount(commandBuffer, stride);`
- `vkCmdEndQuery(commandBuffer, query);`
- `vkCmdEndQuery(commandBuffer, query);`
- `vkCmdEndQuery(commandBuffer, query);`
- `vkCmdEndRenderPass(commandBuffer);`
- `vkCmdEndRenderPass(commandBuffer);`
- `vkCmdEndRenderPass(commandBuffer);`
- `vkCmdExecuteCommands(commandBuffer, commandBufferCount, const pCommandBuffers);`
- `vkCmdExecuteCommands(commandBuffer, commandBufferCount, const pCommandBuffers);`
vkCmdFillBuffer(commandBuffer, dstBuffer, dstOffset, size, data);
vkCmdNextSubpass(commandBuffer, contents);
vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);
vkCmdPushConstants(commandBuffer, pSet, stageFlags, offset, size, pValues);
vkCmdPushDescriptorSetWithTemplateKHR(commandBuffer, pTemplate, pValues);
vkCmdPipelineBarrier(commandBuffer, srcStageMask, dstStageMask, dependencyFlags, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);
vkCmdResetEvent(commandBuffer, event, stageMask);
vkCmdResetQueryPool(commandBuffer, queryPool, firstQuery, queryCount);
vkCmdSetBlendConstants(commandBuffer, blendConstants);
vkCmdSetDepthBounds(commandBuffer, minDepthBounds, maxDepthBounds);
vkCmdSetDeviceMaskKHR(commandBuffer, deviceMask);
vkCmdSetDiscardRectangleEXT(commandBuffer, firstDiscardRectangle, discardRectangleCount, pDiscardRectangles);
vkCmdSetEvent(commandBuffer, event, stageMask);
vkCmdSetLineWidth(commandBuffer, lineWidth);
vkCmdSetScissor(commandBuffer, firstScissor, scissorCount, pScissors);
vkCmdSetStencilCompareMask(commandBuffer, faceMask, compareMask);
vkCmdSetStencilReference(commandBuffer, faceMask, reference);
vkCmdSetStencilWriteMask(commandBuffer, faceMask, writeMask);
vkCmdSetViewport(commandBuffer, firstViewport, viewportCount, pViewports);
vkCmdSetViewportWScalingNV(commandBuffer, firstViewport, viewportCount, pViewportWScalings);
vkCmdUpdateBuffer(commandBuffer, dstBuffer, dstOffset, dataSize, pData);
vkCmdWaitEvents(commandBuffer, eventCount, pEvents, srcStageMask, dstStageMask, memoryBarrierCount, pMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers, bufferMemoryBarrierCount, pBufferMemoryBarriers, imageMemoryBarrierCount, pImageMemoryBarriers);
vkCmdWriteTimestamp(commandBuffer, pipelineStage, queryPool, query);

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

VkResult RenderScene( )
{
VkResult result;
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;
vkCmdBeginRenderPass ( CommandBuffers[nextImageIndex], IN &vcbbi, IN VK_SUBPASS_CONTENTS_INLINE )
VkViewport viewport =
{
0., 0., (float)Width, (float)Height, 0., 1.};

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

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

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

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

vkCmdBindPushConstants(CommandBuffers[nextImageIndex], PipelineLayout, VK_SHADER_STAGE_ALL, offset, size, void *values);

VkBuffer buffers[1] = { MyVertexDataBuffer.buffer };

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

const uint32_t vertexCount = sizeof(VertexData) / sizeof(VertexData[0]);
const uint32_t instanceCount = 1;
const uint32_t firstVertex = 0;
const uint32_t firstInstance = 0;

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

vkCmdEndRenderPass(CommandBuffers[nextImageIndex]);

vkEndCommandBuffer(CommandBuffers[nextImageIndex]);
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;
vsii.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.waitSemaphoreCount = 0;
vsi.pWaitSemaphores = &SemaphoreImageAvailable;
vsi.pWaitDstStageMask = &waitAtBottom;
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
vsi.commandBufferCount = 1;
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;

result = vkQueuePresentKHR(LogicalDevice, renderFence, PALLOCATOR);

The Entire Submission / Wait / Display Process