





1


Queues and Command Buffers

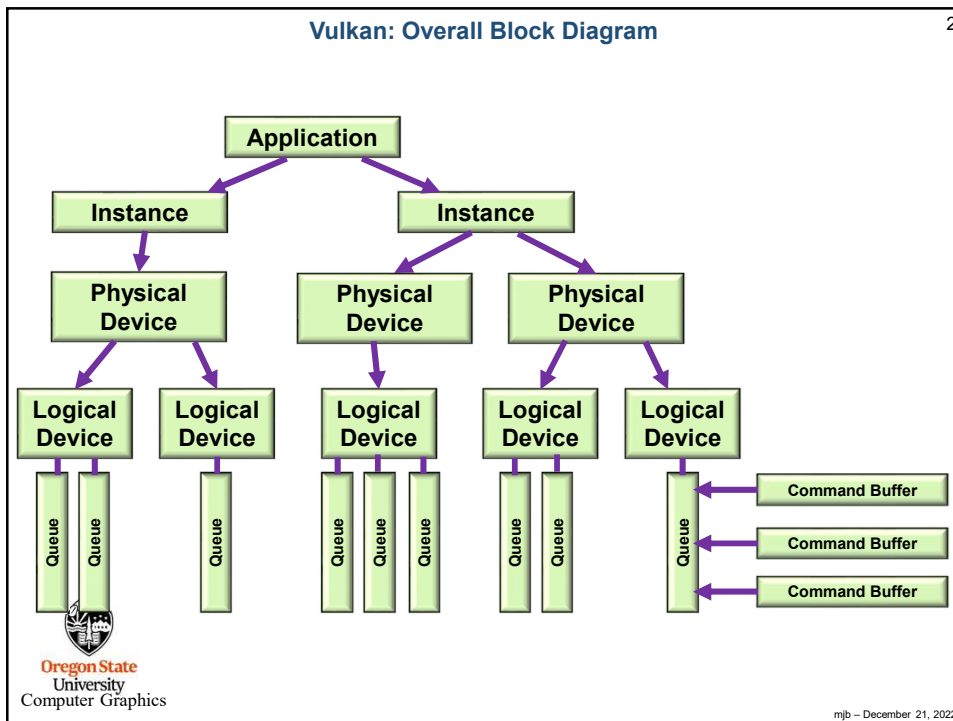

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

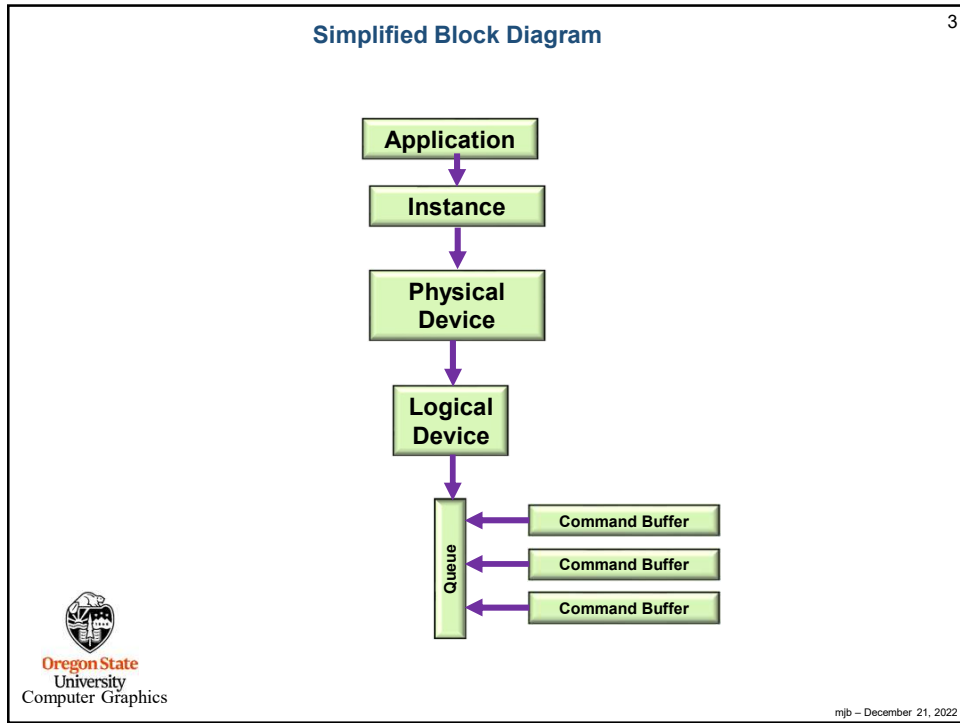


This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/)



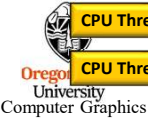
QueuesAndCommandBuffers.pptx mjb - December 21, 2022



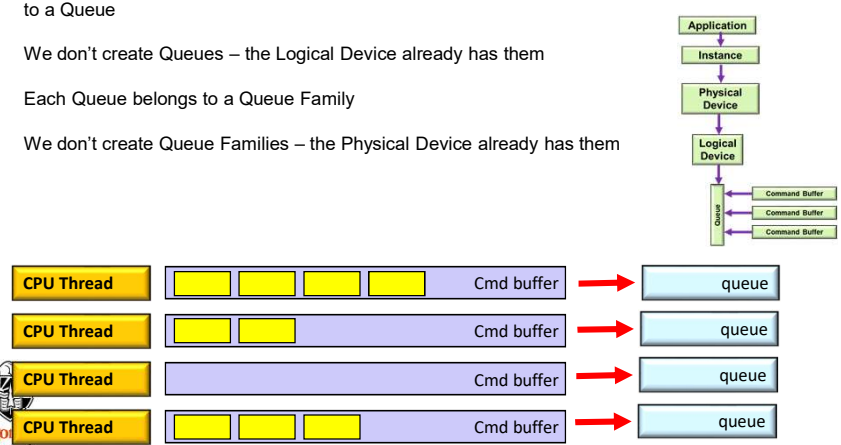


Vulkan Queues and Command Buffers

- 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, but doesn't have to be
- 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 already has them
- Each Queue belongs to a Queue Family
- We don't create Queue Families – the Physical Device already has them



mjb – December 21, 2022



Querying what Queue Families are Available

5

```

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, "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" );
}

```

For the Nvidia A6000 cards:

Found 3 Queue Families:

```

0: Queue Family Count = 16 ; Graphics Compute Transfer
1: Queue Family Count = 2 ; Transfer
2: Queue Family Count = 8 ; Compute Transfer

```



mjb - December 21, 2022

Similarly, we Can Write a Function that Finds the Proper Queue Family

6

```

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;
}

```



mjb - December 21, 2022

Creating a Logical Device Needs to Know Queue Family Information 7

```

float queuePriorities[] =
{
    1. // one entry per queueCount
};

VkDeviceQueueCreateInfo vdcqi[1];
vdcqi[0].sType = VK_STRUCTURE_TYPE_QUEUE_CREATE_INFO;
vdcqi[0].pNext = nullptr;
vdcqi[0].flags = 0;
vdcqi[0].queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
vdcqi[0].queueCount = 1;
vdcqi[0].queuePriorities = (float*) queuePriorities;

VkDeviceCreateInfo vdc:
vdc.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
vdc.pNext = nullptr;
vdc.flags = 0;
vdc.queueCreateInfoCount = 1; // # of device queues wanted
vdc.pQueueCreateInfos = IN &vdcqi[0]; // array of VkDeviceQueueCreateInfo
vdc.enabledLayerCount = sizeof(myDeviceLayers) / sizeof(char *);
vdc.ppEnabledLayerNames = myDeviceLayers;
vdc.enabledExtensionCount = sizeof(myDeviceExtensions) / sizeof(char *);
vdc.ppEnabledExtensionNames = myDeviceExtensions;
vdc.pEnabledFeatures = IN &PhysicalDeviceFeatures; // already created

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

VkQueue Queue;
uint32_t queueFamilyIndex = FindQueueFamilyThatDoesGraphics();
uint32_t queueIndex = 0;

result = vkGetDeviceQueue ( LogicalDevice, queueFamilyIndex, queueIndex, OUT &Queue );
                
```

Oregon State
University
Computer Graphics

mjb - December 21, 2022

Creating the Command Pool as part of the Logical Device 8

```

VkResult
Init06CommandPool( )
{
    VkResult result;

    VkCommandPoolCreateInfo vcp:
    vcp.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
    vcp.pNext = nullptr;
    vcp.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

    vcp.queueFamilyIndex = FindQueueFamilyThatDoesGraphics();

    result = vkCreateCommandPool( LogicalDevice, IN &vcp, PALLOCATOR, OUT &CommandPool );

    return result;
}
                
```

Oregon State
University
Computer Graphics

mjb - December 21, 2022

Creating the Command Buffers

```

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;
}
                
```

mjb - December 21, 2022

Beginning a Command Buffer – One per Image

```

VkSemaphoreCreateInfo vsci;
vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
vsci.pNext = nullptr;
vsci.flags = 0;

VkSemaphore imageReadySemaphore;
result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, 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 );

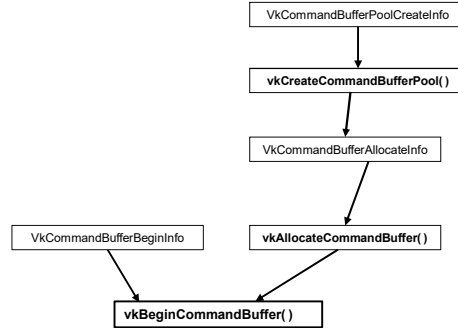
...

vkEndCommandBuffer( CommandBuffers[nextImageIndex] );
                
```

mjb - December 21, 2022

Beginning a Command Buffer

11



mjb - December 21, 2022

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

12

- vkCmdBeginConditionalRendering
- vkCmdBeginDebugUtilsLabel
- vkCmdBeginQuery
- vkCmdBeginQueryIndexed
- vkCmdBeginRendering
- vkCmdBeginRenderPass
- vkCmdBeginRenderPass2
- vkCmdBeginTransformFeedback
- vkCmdBindDescriptorSets
- vkCmdBindIndexBuffer
- vkCmdBindInvocationMask
- vkCmdBindPipeline
- vkCmdBindPipelineShaderGroup
- vkCmdBindShadingRateImage
- vkCmdBindTransformFeedbackBuffers
- vkCmdBindVertexBuffers
- vkCmdBindVertexBuffers2
- vkCmdBlitImage

- vkCmdBlitImage2
- vkCmdBuildAccelerationStructure
- vkCmdBuildAccelerationStructuresIndirect
- vkCmdBuildAccelerationStructures
- vkCmdClearAttachments
- vkCmdClearColorImage
- vkCmdClearDepthStencilImage
- vkCmdCopyAccelerationStructure
- vkCmdCopyAccelerationStructureToMemory
- vkCmdCopyBuffer
- vkCmdCopyBuffer2
- vkCmdCopyBufferToImage
- vkCmdCopyBufferToImage2
- vkCmdCopyImage
- vkCmdCopyImage2
- vkCmdCopyImageToBuffer
- vkCmdCopyImageToBuffer2
- vkCmdCopyMemoryToAccelerationStructure



mjb - December 21, 2022

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

13

```

vkCmdCopyQueryPoolResults
vkCmdCuLaunchKernelX
vkCmdDebugMarkerBegin
vkCmdDebugMarkerEnd
vkCmdDebugMarkerInsert
vkCmdDispatch
vkCmdDispatchBase
vkCmdDispatchIndirect
vkCmdDraw
vkCmdDrawIndexed
vkCmdDrawIndexedIndirect
vkCmdDrawIndexedIndirectCount
vkCmdDrawIndirect
vkCmdDrawIndirectByteCount
vkCmdDrawIndirectCount
vkCmdDrawMeshTasksIndirectCount
vkCmdDrawMeshTasksIndirect
vkCmdDrawMeshTasks

```

```

vkCmdDrawMulti
vkCmdDrawMultiIndexed
vkCmdEndConditionalRendering
vkCmdEndDebugUtilsLabel
vkCmdEndQuery
vkCmdEndQueryIndexed
vkCmdEndRendering
vkCmdEndRenderPass
vkCmdEndRenderPass2
vkCmdEndTransformFeedback
vkCmdExecuteCommands
vkCmdExecuteGeneratedCommands
vkCmdFillBuffer
vkCmdInsertDebugUtilsLabel
vkCmdNextSubpass
vkCmdNextSubpass2
vkCmdPipelineBarrier
vkCmdPipelineBarrier2

```



mjb - December 21, 2022

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

14

```

vkCmdPreprocessGeneratedCommands
vkCmdPushConstants
vkCmdPushDescriptorSet
vkCmdPushDescriptorSetWithTemplate
vkCmdResetEvent
vkCmdResetEvent2
vkCmdResetQueryPool
vkCmdResolveImage
vkCmdResolveImage2
vkCmdSetBlendConstants
vkCmdSetCheckpoint
vkCmdSetCoarseSampleOrder
vkCmdSetCullMode
vkCmdSetDepthBias
vkCmdSetDepthBiasEnable
vkCmdSetDepthBounds
vkCmdSetDepthBoundsTestEnable
vkCmdSetDepthCompareOp

```

```

vkCmdSetDepthTestEnable
vkCmdSetDepthWriteEnable
vkCmdSetDeviceMask
vkCmdSetDiscardRectangle
vkCmdSetEvent
vkCmdSetEvent2
vkCmdSetExclusiveScissor
vkCmdSetFragmentShadingRateEnum
vkCmdSetFragmentShadingRate
vkCmdSetFrontFace
vkCmdSetLineStipple
vkCmdSetLineWidth
vkCmdSetLogicOp
vkCmdSetPatchControlPoints
vkCmdSetPrimitiveRestartEnable
vkCmdSetPrimitiveTopology
vkCmdSetRasterizerDiscardEnable
vkCmdSetRayTracingPipelineStackSize

```



mjb - December 21, 2022

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

15

```

vkCmdSetSampleLocations
vkCmdSetScissor
vkCmdSetScissorWithCount
vkCmdSetStencilCompareMask
vkCmdSetStencilOp
vkCmdSetStencilReference
vkCmdSetStencilTestEnable
vkCmdSetStencilWriteMask
vkCmdSetVertexInput
vkCmdSetViewport
vkCmdSetViewportShadingRatePalette
vkCmdSetViewportWithCount
vkCmdSetViewportWScaling

```

```

vkCmdSubpassShading
vkCmdTraceRaysIndirect2
vkCmdTraceRaysIndirect
vkCmdTraceRays
vkCmdUpdateBuffer
vkCmdWaitEvents
vkCmdWaitEvents2
vkCmdWriteAccelerationStructuresProperties
vkCmdWriteBufferMarker2
vkCmdWriteBufferMarker
vkCmdWriteTimestamp
vkCmdWriteTimestamp2

```



How the *RenderScene()* Function Works

16

```

VkResult
RenderScene()
{
    VkResult result;
    VkSemaphoreCreateInfo vsci;
    vsci.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
    vsci.pNext = nullptr;
    vsci.flags = 0;

    VkSemaphore imageReadySemaphore;
    result = vkCreateSemaphore( LogicalDevice, IN &vsci, PALLOCATOR, OUT &imageReadySemaphore );

    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 );
}

```




```

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

VkClearDepthStencilValue
vcdsv.depth = 1.f;
vcdsv.stencil = 0;

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

VkOffset2D o2d = { 0, 0 };
VkExtent2D e2d = { Width, Height };
VkRect2D r2d = { o2d, e2d };

VkRenderPassBeginInfo
vrpbi.sType = VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
vrpbi.pNext = nullptr;
vrpbi.renderPass = RenderPass;
vrpbi.framebuffer = Framebuffers[ nextImageIndex ];
vrpbi.renderArea = r2d;
vrpbi.clearValueCount = 2;
vrpbi.pClearValues = vcv; // used for VK_ATTACHMENT_LOAD_OP_CLEAR

vkCmdBeginRenderPass( CommandBuffers[nextImageIndex], IN &vrpbi, IN VK_SUBPASS_CONTENTS_INLINE );

```



```

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

vkCmdSetViewport( CommandBuffers[nextImageIndex], 0, 1, IN &viewport ); // 0=firstViewport, 1=viewPortCount

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 );
// dynamic offset count, dynamic offsets

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

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

VkDeviceSize offsets[1] = { 0 };

vkCmdBindVertexBuffers( CommandBuffers[nextImageIndex], 0, 1, buffers, offsets ); // 0, 1 = firstBinding, bindingCount

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] );


```

Submitting a Command Buffer to a Queue for Execution

```

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;

```


mjb - December 21, 2022

The Entire Submission / Wait / Display Process

```

VkFenceCreateInfo vfc;
vfc.sType = VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;
vfc.pNext = nullptr;
vfc.flags = 0;

VkFence renderFence;
vkCreateFence( LogicalDevice, IN &vfc, PALLOCATOR, OUT &renderFence );
result = VK_SUCCESS;

VkPipelineStageFlags waitAtBottom = VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT;
VkQueue presentQueue;
vkGetDeviceQueue( LogicalDevice, FindQueueFamilyThatDoesGraphics(), 0, OUT &presentQueue );
// 0 =, queueIndex

VkSubmitInfo vsi;
vsi.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
vsi.pNext = nullptr;
vsi.waitSemaphoreCount = 1;
vsi.pWaitSemaphores = &imageReadySemaphore;
vsi.pWaitDstStageMask = &waitAtBottom;
vsi.commandBufferCount = 1;
vsi.pCommandBuffers = &CommandBuffers[nextImageIndex];
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 );

```


per 21, 2022

What Happens After a Queue has Been Submitted?

21

As the Vulkan 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.



Oregon State
University
Computer Graphics



mjb - December 21, 2022