Vulkan: A Hand Waving Guided Tour

Hai Nguyen / Google
Dec 2018
About Me

- My name Hai Nguyen (pronounced Hi Win)
- Been at Google for 4+ years
- Currently Working on DXC/SPIR-V
- Screenshot of my first Vulkan program
  - Circa Christmas 2015
We’re Hiring!!!
Please Contact
Kazuki Hara
kazhara@gmail.com
Agenda

• Getting Started
• HLSL in Vulkan
• Vulkan Best Practices
Getting Started With Vulkan
Getting Started With Vulkan

• Taking a slightly different approach with this talk
  - Instead of walking through code, this talk will go through fundamental Vulkan objects and topics
  - Will end with some pseudo code that outlines a simple Vulkan program

• What’s covered in this talk
  - Basic outline of Vulkan graphics program
  - Conceptual details about Vulkan objects in a small example

• What’s not covered in this talk
  - Code walk through of a Vulkan program
    - Many online tutorials and blog posts available on line with code examples
  - Painful details about every Vulkan object type
  - Create info structs
    - Vulkan spec is best source for understanding these

• There will be a bit of handwaving
  - Some Vulkan objects, like shader module, are pretty straight forward
  - Going into full detail would require at least a few hours!
Vulkan: What Is It?

- Vulkan is a cross platform computing API
  - ‘Computing’ because...
    - ...Vulkan can be used on both GPUs and CPUs
    - ...Vulkan can be used for graphics applications
    - ...Vulkan can be used for compute applications
    - ...Vulkan can be used for graphics + compute applications
  - Cross platform exception
    - Surfaces for graphics application

- Vulkan Platforms
  - Android, iOS, Linux, macOS, Switch, Windows

- Vulkan Ecosystem
  - Almost all Vulkan tools are open source
    - IHV specific tools being the exception
  - Vulkan Validation Layers are open source and available on Github
  - Vulkan Loader is open source and available on Github
  - Open source Vulkan drivers available for AMD and Intel GPUs
Getting Started With Vulkan

• Writing Vulkan code has gotten easier over the past year!
  - Thank you Vulkan community for providing feedback to push Vulkan forward!
  - Thank you Khronos Vulkan WG for working tirelessly to improve the Vulkan experience!

• How has it gotten easier?
  - Vulkan 1.1 was released bringing with it new features
    - Many useful extensions promoted into the Core API
  - Helper libraries helped simplify complex topics in Vulkan
    - AMD’s Vulkan Memory Allocator (VMA)
    - Tobias Hector’s Simple Vulkan Synchronization
  - Features and quality of Vulkan tools continuously improving
    - RenderDoc’s feature richness keeps growing
    - Improved shader compiler support for GLSL and HLSL
    - Binaries available for DXC so you don’t have to build it yourself
  - Continuously refined spec language
    - Vulkan specification continues to improve clarity and provides greater detail
  - More Vulkan literature and tutorials available
    - Numerous resources are now available online to help get started with Vulkan
Getting Started With Vulkan

• Essentials
  - PC running Linux or Windows equipped with a GPU that supports Vulkan
  - Vulkan SDK from LunarG
  - C/C++ Compiler
    - Visual Studio on Windows
    - clang or gcc on Linux
      - Visual Studio Code is also available on Linux
      - Your favorite Linux IDE
    - VIM or EMacs

• Supplementary (all available on Github)
  - GLFW
    - Simple cross platform windowing library
  - Vulkan Memory Allocator
    - Easy to use memory allocation library for Vulkan
  - Simple Vulkan Synchronization
    - Reduces the headache of trying to understand Vulkan synchronization
Vulkan Application Overview

- Create Vulkan instance
- Create Vulkan device
- Create surface and swapchain
- Create some or all of the following
  - Buffers, images, descriptor sets, shaders, pipelines, command buffers, semaphores, fences

- Render loop
  - Update resources
  - Update descriptors with references to resources
  - Build command buffers
  - Submit command buffers
  - Present

- When exiting application...
  - ...destroy all the objects created for application
Fundamental Vulkan Objects

- Any Vulkan application will require creation / retrieval of these objects
  - VkInstance
  - VkPhysicalDevice
  - VkDevice
  - VkQueue

- Creation Sequence
  - VkInstance object must be the first Vulkan object created
  - VkPhysicalDevice object(s) are enumerated from the VkInstance object
  - VkDevice object is created from a VkPhysicalDevice object
  - VkQueue object(s) are retrieved from VkDevice object
**VkInstance**

- Stores per application states for Vulkan
- Created with `vkCreateInstance`
- Destroyed with `vkDestroyInstance`
- **Instance layers and extensions**
  - Instance level layers must be specified in create info
  - Instance level extensions must be specified in create info
- **Single instance can support multiple Vulkan implementations**
  - TL;DR; Vulkan instance knows about all the registered GPUs that support Vulkan on the system
- **Does the Vulkan instance need to remain alive throughout an application’s lifetime?**
  - Yes!
  - Destroy the instance object will lead to undefined behavior or most likely a crash!
VkPhysicalDevice

- Represents a single complete implementation of Vulkan available to the application
  - Conceptually you can think of it as the GPU

- Enumerated using vkEnumeratePhysicalDevices

- Once enumerated physical devices can be queried for device properties
  - vkGetPhysicalDeviceProperties
  - Driver version, vendor, device type, device name, etc.

- Things applications should pay attention to because they will vary
  - Device features - vkGetPhysicalDeviceFeatures
  - Lets the application know what the Vulkan implementation is capable of
  - Device limits - vkGetPhysicalDeviceProperties
  - Lets the application know the limits of the Vulkan implementation are
    - Max buffer sizes, max image dimensions, max memory allocations, etc.
  - Queue family properties - vkGetPhysicalDeviceQueueFamilyProperties
    - Lets the application know what queues are available
  - Helps determine if the Vulkan implementation can run your application
VkDevice

• Logical Device
• Provides connection between application and physical device
• Multiple logical devices an be created from a single physical device
  - In practice you probably won’t do this, special case
• Created with vkCreateDevice
• Destroyed with vkDestroyDevice
• Device layers and extensions
  - **WARNING**: Device level layers are deprecated and will be ignored
  - Device level extensions must be specified in create info
• Queues
  - Must tell the [logical] device creation what queues application plans to use
  - Queue types and counts can be queried from physical device
• Make your life easier!
  - When starting with Vulkan just use a single graphics queue
VkQueue

- Retrieved using `vkGetDeviceQueue`
- Queues belong to queue families, which determines the type of commands the queue can execute
  - Queue families can support a combination of the following command types
    - Graphics, Compute, and Transfer
  - Graphics queues also support Compute and Transfer commands
  - Compute queues also support Transfer commands

- Queue configurations will vary from GPU to GPU

- **Over Simplification:**
  - AMD - Single graphics, multiple compute queues, multiple transfer queues
  - Intel - One queue to rule them all
  - NVIDIA - Multiple graphics queues, multiple compute queues, multiple transfer queues

- Applications should be aware of different queue configurations
  - Rigid queue requirements may prevent applications running correctly or at all

- **Make your life easier!**
  - When starting with Vulkan just use a single graphics queue
Fundamental Vulkan Objects (RECAP)

- Any Vulkan application will require creation / retrieval of these objects
  - VkInstance
  - VkPhysicalDevice
  - VkDevice
  - VkQueue

- Creation Sequence
  - VkInstance object must be the first Vulkan object created
  - VkPhysicalDevice object(s) are enumerated from the VkInstance object
  - VkDevice object is created from a VkPhysicalDevice object
  - VkQueue object(s) are retrieved from VkDevice object
Fundamental Vulkan Graphics Objects

- For Vulkan graphics applications, two more fundamental objects are needed
  - VkSurfaceKHR
  - VkSwapchainKHR

- Both are WSI objects
  - Window System Integration
  - WSI is what connects the application to the platforms windowing system
    - This what allows the applications to present what it renders (aka page swap or page flip)

- Creation sequence
  - VkSurfaceKHR object is created from VkInstance object
    - NOTE: VkSurfaceKHR is an instance level object
  - VkSwapchainKHR is created from vkDevice object using VkSurfaceKHR object
    - NOTE: VkSwapchainKHR is a device level object
VkSurfaceKHR

- Creation of VkSurfaceKHR is platform specific!
  - Windows
    - vkCreateWin32SurfaceKHR
  - Linux
    - vkCreateXcbSurfaceKHR / vkCreateXlibSurfaceKHR
  - Android
    - vkCreateAndroidSurfaceKHR

- Each vkCreate*SurfaceKHR function takes specific platform details
  - See Vk*SurfaceCreateInfo structs for platform details
  - GLFW has functions to access Windows and Linux details
  - GLFW can also take care of creating the surface for you

- Vulkan surfaces capabilities are very straight forward
  - Surface capabilities tells you ever you need to know about the surface
    - Min/max image dimensions, min/max image count, pixel format, color spaces, etc.
  - Use vkGetPhysicalDeviceSurfaceCapabilitiesKHR to get capabilities
  - No crazy guessing games or creating dummy windows for pixel formats
  - Easier than OpenGL!
VkSwapchainKHR

- Back to platform independence!
- Created with vkCreateSwapchainKHR
- Destroyed with vkDestroySwapchainKHR
- Created from VkDevice object

- Use surface capabilities to help decide values for swapchain parameters
  - Parameters will vary from implementation to implementation
  - Example parameters:
    - Minimum / maximum image count
    - Minimum / maximum image size (aka extents)
    - Image format (pixel format)
    - Color spaces
    - Presentation mode

- Once the swapchain is created, VkImage objects can be retrieved from the swapchain and used for rendering
  - vkGetSwapchainImagesKHR returns a list of available images
Fundamental Vulkan Graphics Objects (RECAP)

- Any Vulkan application will require creation / retrieval of these objects
  - VkInstance
  - VkPhysicalDevice
  - VkDevice
  - VkQueue
  - VkSurfaceKHR
  - VkSwapchainKHR

- Creation Sequence
  - VkInstance object must be the first Vulkan object created
  - VkPhysicalDevice object(s) are enumerated from the VkInstance object
  - VkDevice object is created from a VkPhysicalDevice object
  - VkQueue object(s) are retrieved from VkDevice object
  - VkSurfaceKHR object is created from VkInstance object
  - VkSwapchainKHR object is created from VkDevice object using VkSurfaceKHR object
Writing a Vulkan Graphics Application

- We just need a few more objects for a Vulkan graphics app
  - VkDescriptorsetLayout
  - VkDescriptorPool
  - VkDescriptorSet
  - VkPipelineLayout
  - VkPipeline
  - VkShaderModule
  - VkBuffer
  - VkImage
  - VkImageView
  - VkSampler
  - VkDeviceMemory
  - VkRenderPass
  - VkFramebuffer
  - VkCommandPool
  - VkCommandBuffer

- Don’t worry! We’re not going to go through all these!
Writing a Vulkan Graphics Application

• Writing a simple Vulkan application that draws a spinning textured triangle
  - VkBuffer object + VkDeviceMemory object for vertex and texture coordinates
    - We’ll skip using an index buffer to keep things simple
  - VkBuffer object + VkDeviceMemory object for uniform (aka constant) data
  - VkImage object + VkDeviceMemory object for texture data
  - VkImageView object to make VkImage object visible to shader via a descriptor
  - VkSampler object to describe texture sampling
  - VkDescriptorSet object to for descriptor data
    - Will also need VkDescriptorSetLayout and VkDescriptorPool objects as well
  - VkRenderPass object + VkFramebuffer object for image attachments
  - VkShaderModule objects for vertex and fragment shaders
    - aka VS and FS (or PS in HLSL parlance)
  - VkPipeline object for render state, shader linkage, pipeline layout
    - VkPipelineLayout specifies the descriptor types that the pipeline will use
  - VkCommandBuffer
    - Will also need VkCommandPool

• Lets shorten this!
Writing a Vulkan Graphics Application

- Writing a simple Vulkan application that draws a spinning textured triangle
  - VkBuffer object + VkDeviceMemory = Vertex Buffer
  - VkBuffer object + VkDeviceMemory = Uniform Buffer / Constant Buffer
  - VkImage object + VkDeviceMemory = Texture
  - VkImageView object = Image View
  - VkSampler object = Sampler
  - VkDescriptorSet object = Descriptor Set
  - VkRenderPass object + VkFramebuffer object = Render Pass
  - VkShaderModule object(s) = Shader Modules
  - VkPipeline object = Pipeline (aka Graphics Pipeline)
  - VkCommandBuffer object = Command Buffer

- Lets categorize these objects into functional topics!
Writing a Vulkan Graphics Application

• Writing a simple Vulkan application that draws a spinning textured triangle
  - Objects that require *Memory Allocations*
    - VkBuffer object + VkDeviceMemory = Vertex Buffer
    - VkBuffer object + VkDeviceMemory = Uniform Buffer / Constant Buffer
    - VkImage object + VkDeviceMemory = Texture
  - Objects related to *Descriptor Sets*
    - VkImageView object = Image View
    - VkSampler object = Sampler
    - VkDescriptorSet object = Descriptor Set
  - Objects related to *Render Passes*
    - VkRenderPass object + VkFramebuffer object = Render Pass
  - Objects related to *Graphics Pipelines*
    - VkShaderModule object(s) = Shader Modules
    - VkPipeline object = Pipeline (aka Graphics Pipeline)
  - **Building the Command Buffer**
    - VkCommandBuffer object = Command Buffer

• Lets take a brief look at these topics!
Memory Allocations

- Images and buffers require memory allocation
  - These are the only two Vulkan objects that have explicit memory allocation

- Allocated memory must be freed when no longer used

- Over Simplification:
  - Device Local means GPU memory (fast)
  - Host Visible means CPU memory (slow)
  - On UMA systems these are the same thing

- Vulkan has different memory types that come from different heaps
  - Memory type and heaps will vary from implementation to implementation
  - It takes some time to fully understand memory in Vulkan

- Make your life easier! Use Vulkan Memory Allocator (aka VMA)!
  - [https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator](https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator)
    - Apart of AMD’s GPUOpen initiative
    - Author is Adam Sawicki
    - Google “vulkan memory allocator”

- Ash’s talk will go in depth about Vulkan memory usage!
Descriptor Sets

• **Over Simplification:**
  - Descriptor sets are containers for descriptor bindings that are described using `VkDescriptorSetLayout` objects
  - Descriptor bindings connect resources to shaders
    - Descriptor bindings are sometimes referred to as descriptors
    - Buffers, images, and samplers are all resources
  - A descriptor set is referred to by its ‘set number’ from a shader

• **Descriptor sets must be allocated from descriptor pools**
  - Descriptor pools maintains a pool available descriptors
  - Descriptor pools are created with fixed sizes for each descriptor type (`VkDescriptorType`)
    - Descriptor pool sizes example:
      - 20 descriptors for Sampled Images (Textures)
      - 10 descriptors for Uniform Buffers (Constant Buffers)
      - 15 descriptors for Storage Buffers (Structured Buffers)
      - 5 descriptors for Samplers
  - Allocation will fail when a pool runs out of available descriptors
Descriptor Sets

• Before a descriptor set can be used it must be updated
  - References to resources are written to the descriptor set

• Example descriptor set
  - 1 Uniform Buffer
  - 1 Sampled Image
  - 1 Sampler

• Updating descriptor set
  - Update descriptor for Uniform Buffer to refer to a VkBuffer object
  - Update descriptor for Sampled Image to refer to a VkImage object
    - Actually a VkImageView
  - Update descriptor for Sampler to refer to a VkSampler object

• Once updated, descriptor sets can be bound for use
Render Passes

• **Over Simplification:**
  - Stores a list of attachments (images) an application renders to
  - Each attachment has pixel format and sample count associated with it

• **In Vulkan everything that gets drawn must happen within a render pass**
  - Conceptually looks like this:
    - Begin a render pass
    - Draw geometry
    - End render pass
  - Render passes cannot be nested
  - A render pass that has begun must be ‘ended’ before another can begin

• **Why is there a framebuffer associated with a render pass? (TL;DR;)**
  - Framebuffers stores the actual references to the attachments (images) you’ll render to
  - Framebuffers stores dimensions (Width x Height) used for rendering

• **What about subpasses?**
  - If you’re just getting started, just use one subpass
  - If you’re not planning to target mobile, just use one subpass
Graphics Pipeline

**Over Simplification:**
- Graphics pipeline stores vertex data description, shaders, render states, pipeline layout, etc.
  - Biggest create info struct in Vulkan, please see spec
- Describes what type of geometry (primitive topology) is rendered and how it’s rendered
- Similar, in concept, to graphics pipelines of every other graphics API

**Shaders in a graphics pipeline are per shader stages**
- Shaders for each shader stage is referenced using `VkShaderModule` objects
- Valid combinations
  - VS
  - VS / PS
  - VS / GS / PS
  - VS / HS / DS / PS
  - VS / HS / DS / GS / PS
  - HS (Hull Shader aka Tessellation Control Shader), DS (Domain Shader aka Tessellation Evaluation Shader)

**Pipeline Layout**
- Conceptually a program interface between the Vulkan API and shaders in a pipeline
- Takes in *descriptor set layouts* as apart of its creation
  - Remember that a *descriptor set layout* is what a descriptor set looks like not the actual references to resources
- Tells Vulkan what types of resources the shaders will use
  - This means that there must be agreement between what the shader expects and what’s in the pipeline layout
Building the Command Buffer

- Writing a simple Vulkan application that draws a spinning textured triangle
  - `VkBuffer` object + `VkDeviceMemory` = Vertex Buffer
  - `VkBuffer` object + `VkDeviceMemory` = Uniform Buffer / Constant Buffer
  - `VkImage` object + `VkDeviceMemory` = Texture
  - `VkImageView` object = Image View
  - `VkSampler` object = Sampler
  - `VkDescriptorSet` object = Descriptor Set
  - `VkRenderPass` object + `VkFramebuffer` object = Render Pass
  - `VkShaderModule` object(s) = Shader Modules
  - `VkPipeline` object = Pipeline (aka Graphics Pipeline)
  - `VkCommandBuffer` object = Command Buffer

- Lets rearrange it a bit!
Building the Command Buffer

- External to building the command buffer
  - VkBuffer object + VkDeviceMemory = Uniform Buffer / Constant Buffer
  - VkImage object + VkDeviceMemory = Texture
  - VkImageView object = Image View
  - VkSampler object = Sampler
  - VkDescriptorSet object = Descriptor Set

- Building the command buffer
  - VkBuffer object + VkDeviceMemory = Vertex Buffer
  - VkDescriptorSet object = Descriptor Set
  - VkRenderPass object + VkFramebuffer object = Render Pass
  - VkPipeline object = Pipeline (aka Graphics Pipeline)

- Lets clean it up!
Building the Command Buffer

• External to building the command buffer
  - Uniform Buffer / Constant Buffer
  - Image View (references Texture)
  - Sampler
  - Descriptor Set (needs updating to reference Uniform Buffer, Image View, and Sampler)

• Building the command buffer
  - Vertex Buffer
  - Descriptor Set
  - Render Pass
  - Graphics Pipeline

• Lets map this to Vulkan calls!
Building the Command Buffer

- **External to command buffer building**
  - Uniform Buffer / Constant Buffer
  - Image View (references Texture)
  - Sampler
  - Descriptor Set (call \texttt{vkUpdateDescriptorSets} to write uniform buffer, image view, and sampler)

- **Command buffer building**
  - Call \texttt{vkBeginCommandBuffer}
  - Bind Vertex Buffer (call \texttt{vkCmdBindVertexBuffers})
  - Bind Descriptor Set (call \texttt{vkCmdBindDescriptorSets})
  - Begin Render Pass (call \texttt{vkCmdBeginRenderPass})
  - Bind Graphics Pipeline (call \texttt{vkCmdBindPipeline})
  - \textit{Draw Triangle} (call \texttt{vkCmdDraw})
  - End Render Pass (call \texttt{vkCmdEndRenderPass})
  - Call \texttt{vkEndCommandBuffer}

- **Lets see the pseudo code!**
Simple Render Loop

// Update transforms, etc.
// Update uniform buffers

vkUpdateDescriptorSets(...); // if necessary

vkBeginCommandBuffer(cmd);
vkCmdBindVertexBuffers(...vertexBuffer...);
vkCmdBindDescriptorSets(...descriptorSet...);
vkCmdBeginRenderPass(...renderPass...);
vkCmdBindPipeline(...graphicsPipeline...);
vkCmdDraw(...);
vkEndCommandBuffer(cmd);

vkQueueSubmit(...cmd...);
vkQueuePresent(...);
Descriptor Sets and Pipeline Layout

- typedef struct VkPipelineLayoutCreateInfo {
  VkStructureType sType;
  const void* pNext;
  VkPipelineLayoutCreateFlags flags;
  uint32_t setLayoutCount;
  const VkDescriptorSetLayout* pSetLayouts;
  uint32_t pushConstantRangeCount;
  const VkPushConstantRange* pPushConstantRanges;
} VkPipelineLayoutCreateInfo;

void vkCmdBindDescriptorSets(
  VkCommandBuffer commandBuffer,
  VkPipelineBindPoint pipelineBindPoint,
  VkPipelineLayout layout,
  uint32_t firstSet,
  uint32_t descriptorSetCount,
  const VkDescriptorSet* pDescriptorSets,
  uint32_t dynamicOffsetCount,
  const uint32_t* pDynamicOffsets);

- Parameter firstSet has two meanings
  - Is the set number of the first descriptor set to be bound
    - For example firstSet=3 means that pDescriptorSets[0] is at set number 3, pDescriptorSets[1] is at set number 4, pDescriptorSets[2] is at set number 5, and so on
  - It also serves as an index that maps into VkPipelineLayoutCreateInfo::pSetLayouts array
    - So for firstSet=3 the descriptor set layout of pDescriptorSets[0...] must match what was specified to VkPipelineLayoutCreateInfo::pSetLayouts[3...] when the Layout (VkPipelineLayout object) was created
  - Validation layers will catch this! Otherwise, undefined behavior!
Getting Started With Vulkan (RECAP)

• Writing Vulkan code has gotten easier over the past year!

• Essentials
  - PC running Linux or Windows equipped with a GPU that supports Vulkan
  - Vulkan SDK from LunarG
  - C/C++ Compiler
    - Visual Studio on Windows
    - clang or gcc on Linux
      - Visual Studio Code is also available on Linux

• Supplementary (all available on Github)
  - GLFW
    - Simple cross platform windowing library
  - Vulkan Memory Allocator
    - Easy to use memory allocation library for Vulkan
  - Simple Vulkan Synchronization
    - Reduces the headache of trying to understand Vulkan synchronization
HLSL in Vulkan
HLSL in Vulkan

- Vulkan Shader Languages
- HLSL Compilers
- HLSL Example
- Registers and Bindings
- Sets and Spaces
- Resource Mappings
- HLSL Vulkanizations
Vulkan Shader Languages

- Vulkan does not have a spec for a high level language for shaders
- Vulkan implementations must accept shaders in SPIR-V format
- Vulkan ecosystem tools support two high level languages
  - GLSL
  - HLSL
- Most tutorials and code samples available online use GLSL
  - But we wanted to let you know that HLSL can be used as well :)
HLSL Compilers and Language

- **HLSL Compilers for Vulkan**
  - DirectX Shader Compiler (aka DXC) - Microsoft’s open source HLSL compiler
    - Google contributed and maintains the SPIR-V backend
  - glslang - Khronos reference compiler

- **HLSL Language in Vulkan**
  - DXC supports SM6.0
    - Work on 6.1+ has started
  - glslang supports SM5.1+
    - Some HLSL 5.1 features not supported
    - Some HLSL 6.0 features present thanks to community contributions
  - Source compatible with HLSL in DirectX
    - Caveat features not available in Vulkan
    - DXC has stronger source compatibility support

- **Which compiler should I use for HLSL in Vulkan?**
  - DXC is recommended if you care about SM6.0 and up
  - DXC is the future of HLSL in Vulkan going forward
  - DXC or glslang is fine if you’re only targeting SM5.1 and below
HLSL Example

```hlsl
// dxc -spirv -Tvs_5_0 -E VSMain -Fo vs.spv shader.hlsl
// dxc -spirv -Tps_5_0 -E PSMain -Fo ps.spv shader.hlsl
cbuffer UniformBlock0 : register(b0, space0) {
    float4x4 mvp;
};

struct VSOutput {
    float4 Position : SV_POSITION;
    float2 TexCoord : TEXCOORD;
};

VSOutput VSMain(float4 Position : POSITION, float2 TexCoord : TEXCOORD0) {
    VSOutput result;
    result.Position = mul(mvp, Position);
    result.TexCoord = TexCoord;
    return result;
}

Texture2D uTex0 : register(t1, space0);
SamplerState uSampler0 : register(s2, space0);

float4 PSMain(VSOutput input) : SV_TARGET {
    return uTex0.Sample(uSampler0, input.TexCoord);
}
```

HLSL in Vulkan
 Registers and Bindings

- All Vulkan resource types map to a binding number
  - HLSL’s `register(bW)` maps to binding number W
  - HLSL’s `register(sX)` maps to binding number X
  - HLSL’s `register(tY)` maps to binding number Y
  - HLSL’s `register(uZ)` maps to binding number Z

- Unlike DirectX, Vulkan does not have namespaces for resource types
  - `register(bN)`, `register(sN)`, `register(tN)`, and `register(uN)` all map to binding N
  - `register(b0)`, `register(s0)`, `register(t0)`, and `register(u0)` all map to binding 0

- Shift flags in DXC and glslang
  - Helps developers work around the lack of namespace for resource types in Vulkan
  - Shift flags lets you shift the binding number based on the resource type
  - For example, to shift all `b#` resource types in DXC by 16
    - Use the command line option: `-fvk-b-shift 16 0`
Sets and Spaces

- HLSL’s `spaceN` keyword is analogous to set N in Vulkan
  - HLSL’s `register(b0, space0)` is analogous to binding number 0, set number 0
  - HLSL’s `register(t2, space1)` is analogous to binding number 2, set number 1
  - HLSL’s `register(u4, space2)` is analogous to binding number 4, set number 2
  - HLSL’s `register(s8, space6)` is analogous to binding number 8, set number 6

- If a `spaceN` is not provided, the default is `space0 / set 0`
  - All these will be in set 0
    - `ConstantBuffer<T>` MyConstants : register(b0);
    - `Texture2D` MySampledImage : register(t2);
    - `RWTexture2D<float4>` MyOutputImage : register(u4);
    - `SamplerState` MySampler : register(s8);
Resource Mappings

- DirectX’s HLSL resource to Vulkan descriptor type mapping
  - `cbuffer / ConstantBuffer<T>` : CBV (Constant Buffer View)
    - `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER`
    - `VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC`
  - `Texture[N]D, Buffer<T>, StructuredBuffer<T>` : SRV (Shader Resource View)
    - `VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE`
    - `VK_DESCRIPTOR_TYPE_UNIFORM_TEXEL_BUFFER`
    - `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`
  - `RWTexture[N]D, RWBuffer<T>, RWStructuredBuffer<T>` : UAV (Unordered Access View)
    - `VK_DESCRIPTOR_TYPE_STORAGE_IMAGE`
    - `VK_DESCRIPTOR_TYPE_STORAGE_TEXEL_BUFFER`
    - `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`
    - `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER_DYNAMIC`
  - `SamplerState`: Sampler
    - `VK_DESCRIPTOR_TYPE_SAMPLER`
  - No HLSL mapping for these types
    - `VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER`
SV_TARGET Outputs

- **SV_TARGET** is the same as **SV_TARGET0**
  - Same as it is in DirectX

- **SV_TARGET[N]** refers to the Nth color attachment in a VkRenderPass

- **Dual source blending**
  - Vulkan does not automatically detect that dual source blending is present
  - Must tell SPIR-V targets intended for dual source blending using attributes
  - Example:
    ```
    struct PSOut {
        [[vk::location(0), vk::index(0)]] float4 a: SV_TARGET0;
        [[vk::location(0), vk::index(1)]] float4 b: SV_TARGET1;
    };
    PSOut main() { ... }
    ```
## HLSL Vulkanizations

```cpp
// Vulkan binding=1, set=1
[[vk::binding(1, 1)]] cbuffer UniformBlock0 : register(b0, space0) {
    float4x4 mvp;
};

struct VSOutput {
    float4 Position : SV_POSITION;
    float2 TexCoord : TEXCOORD;
};

VSOutput VSMain(float4 Position : POSITION, float2 TexCoord : TEXCOORD0) {
    VSOutput result;
    result.Position = mul(mvp, Position);
    result.TexCoord = TexCoord;
    return result;
}

// Vulkan binding=0, set=2 and binding=1, set=2
[[vk::binding(0, 2)]] Texture2D uTex0 : register(t1, space0);
[[vk::binding(1, 2)]] SamplerState uSampler0 : register(s2, space0);

float4 PSMain(VSOutput input) : SV_Target {
    return uTex0.Sample(uSampler0, input.TexCoord);
}
```
HLSL in Vulkan (RECAP)

• HLSL Compilers
  - DXC - https://github.com/Microsoft/DirectXShaderCompiler
    - Google “dxc shader compiler”
  - glslang - https://github.com/KhronosGroup/glslang
    - Google “glslang”

• HLSL Language in Vulkan
  - DXC supports SM6.0
    - Work on 6.1+ has started
  - glslang supports SM5.1+
    - Some HLSL 5.1 features not supported
    - Some HLSL 6.0 features present thanks to community contributions
  - Source compatible with HLSL in DirectX
    - Caveat features not available in Vulkan
    - DXC has stronger source compatibility support

• DXC/SPIR-V HLSL Feature Mapping Manual
  - https://github.com/Microsoft/DirectXShaderCompiler/blob/master/docs/SPIR-V.rst
  - Google “dxc hlsl spirv manual”
Vulkan Best Practices
Vulkan Best Practices

- Features, Limits, and Portability
- Multiple Frames In Flight
- Vulkan Memory Allocator (VMA)
- Pipelines and Render Passes
- Pipeline Barriers
- Performance
- Debugging Tips
- Tools
Features, Limits, and Portability

• Features
  - Only enable the GPU features that are needed by the application
    - Applications must handle cases where features and capabilities aren’t available
  - Turning all available features returned from vkGetPhysicalDeviceFeatures is a bad idea
    - Might turn on unwanted behavior
  - Features will vary from GPU to GPU (or implementation to implementation)
    - Even different GPUs from the same IHV

• Limits
  - Don’t make any assumptions about the consistency of limits
  - Limits will vary from GPU to GPU (or implementation to implementation)
    - Even different GPUs from the same IHV

• Portability
  - Certain features and capabilities are unique to an IHV
    - Applications must handle cases where features and capabilities aren’t available
  - AMD’s 256MB Device Local | Host Visible heap
    - Using this heap may require an additional path for updating Buffers and Images
Multiple Frames In Flight

- Allows work to begin on frame N+1, N+2, etc. while frame N is rendering
- For every possible frame in flight there will be some amount of dependencies
  - For example:
    - Uniform buffers
    - Descriptor sets
    - Command buffers
    - Semaphores
    - Fences
    - Images
    - Image Views
    - Render passes for swapchain images
  - Application may X copies of dependencies for X number frames in flight
- How does the application know when a frame’s resources can be used?
  - Use a fence, on a frame’s last work submission, to signal when all work is complete
  - Every frame in flight should have a fence associated with it
**Vulkan Memory Allocator (VMA)**

- **Make your life easier! Use Vulkan Memory Allocator!**
  - Production ready!
  - [https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator](https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator)
    - Apart of AMD’s GPUOpen initiative
    - Author is Adam Sawicki
    - Google “vulkan memory allocator”

- **Why VMA?**
  - Helps to choose correct and optimal memory type based on usage
  - Usage flags are based on idioms and are descriptive
    - GPU_ONLY
    - CPU_ONLY
    - CPU_TO_GPU
    - GPU_TO_CPU
  - Cross platform and cross vendor support
    - Android, Linux, macOS, Windows
    - AMD, ARM, Intel, NVIDIA, Qualcomm, etc.
  - Pool memory allocation with support for custom memory pools

- **Ash’s talk will go in depth about Vulkan memory usage!**
Pipelines and Render Passes

- Do I need to create all the render passes that are used by pipelines before creating the pipelines?
  - NO!

- Pipelines can be created using a dummy render pass object!
  - Pipelines can be used with render passes that are compatible with the render pass specified during pipeline creation!
  - Two render passes are compatible if their corresponding attachments have the same pixel format and sample count!
    - The number of attachments can even be different...
    - ...as long as all the existing attachments match!
  - Guaranteed by the spec, see “7.2 Render Pass Compatibility”

- TL;DR;
  - Create dummy render passes to get through all your pipeline creation!
    - REMEMBER: Attachments need to match in pixel format and sample count! That’s it!
  - Create the actual render passes that you’ll use later!
Pipeline Barriers

- Pipeline barriers is Vulkan can be daunting
  - Lots of parameters and struct fields that are confusing

- Make your life easier! Use Simple Vulkan Synchronization!
  - [https://github.com/Tobski/simple_vulkan_synchronization](https://github.com/Tobski/simple_vulkan_synchronization)
    - Written by Tobias Hector
    - Google “simple vulkan synchronization”

- Simple Vulkan Synchronization advantages
  - Great starting point for pipeline barriers in your application
  - Great reference to understand pipeline barriers
  - Usage flags are based on idioms and are descriptive
    - THSVS_ACCESS_VERTEX_BUFFER
    - THSVS_ACCESS_VERTEX_SHADER_READ_UNIFORM_BUFFER
    - THSVS_ACCESS_FRAGMENT_SHADER_READ_UNIFORM_BUFFER
    - THSVS_ACCESS_FRAGMENT_SHADER_READ_SAMPLED_IMAGE_OR_UNIFORM_TEXEL_BUFFER
    - etc..

Getting Started With Vulkan
Performance

- **Memory Performance**
  - On non-UMA GPUs Device Local memory is *always* faster than Host Visible
    - Buffers and images that accessed frequently during draws or dispatches should reside in Device Local
  - Ash’s talk will go in depth about memory usage in Vulkan

- **Updating Descriptors**
  - General advice is to organize descriptor sets based on update frequency
    - Still holds true, but isn’t always easy to do in practice!
  - Consider VK_KHR_descriptor_update_template (in Core API as of Vulkan 1.1)
    - Designed to allow quick and sensible updates of descriptors

- **Clearing Attachments**
  - In general, clearing on render pass loads are the fastest path
    - May vary from GPU to GPU - measure the time to make sure!

- **Other Things to Consider**
  - On certain GPUs things can run slower in Secondary Command Buffers
  - Consider using VkEvents where possible for synchronization
    - Remember that VkEvent objects must operate in the same queue!
  - Fences are very heavy synchronization objects, try to limit use as much as possible!
Debugging Tips

- **Use Vulkan Validation Layers!**
  - Helps ensure that your app is using Vulkan correctly!
  - Consider using `VK_EXT_debug_utils` if you’re not using it!
  - Use labels and name objects!

- **If using multiple queues...**
  - Have a single queue path available!
  - Will save you a lot of headache!

- **vkDeviceWaitIdle, vkQueueWaitIdle**
  - Definitely should not be used when performance is necessary!
  - Can be helpful at times to debug certain types of workloads

- **Track execution progress of command buffers using markers and checkpoints!**
  - IHV extensions to help accomplish this
    - `VK_AMD_buffer_marker`
    - `VK_NV_device_diagnostic_checkpoints`
  - Helps you track progress of command buffer execution on the GPU!
  - See “Mark My DWORDS!” talk from Vancouver SIGGRAPH 2018 BOF earlier this year
Tools

- **Debugging Tools**
  - RenderDoc - most popular Vulkan tool!
    - [https://renderdoc.org](https://renderdoc.org)
  - NVIDIA NSight

- **Profiling Tools**
  - AMD RGP
    - [https://github.com/GPUOpen-Tools/Radeon-GPUProfiler/releases](https://github.com/GPUOpen-Tools/Radeon-GPUProfiler/releases)
    - Google “github rgp”
  - NVIDIA NSight

- **SPIR-V reflection**
  - SPIRV-Reflect
    - [https://github.com/chaoticbob/SPIRV-Reflect](https://github.com/chaoticbob/SPIRV-Reflect)
    - Google “spirv reflect”
  - SPIRV-Cross
    - [https://github.com/KhronosGroup/SPIRV-Cross](https://github.com/KhronosGroup/SPIRV-Cross)
    - Google “spirv cross”
We’re Hiring!!!

Please Contact

Kazuki Hara

kazhara@google.com
We’re Hiring!

• **Living Room Games Technical Solutions Consultant**
  - Locations: **Tokyo**, London, Mountain View
  - Drive both technical execution of strategic partner relationships within Google, and end-to-end launches of partner apps/services
  - Perform implementation reviews, evangelize new product features, and ensure the prompt and proper resolution of technical challenges

• **Living Room Games Field Engineer**
  - Locations: **Tokyo**, London, Mountain View
  - Flexibility for travel (average 25%, occasionally more) and short-term co-location with partners
  - Work with partners in order to resolve any technical issues that might arise from the use of Google gaming technologies within partners’ games
  - Heroically overcome the hardest technical issues ensuring an exceptional level of content ecosystem quality

• **Contact:** Kazuki Harakaz | kazhara@google.com