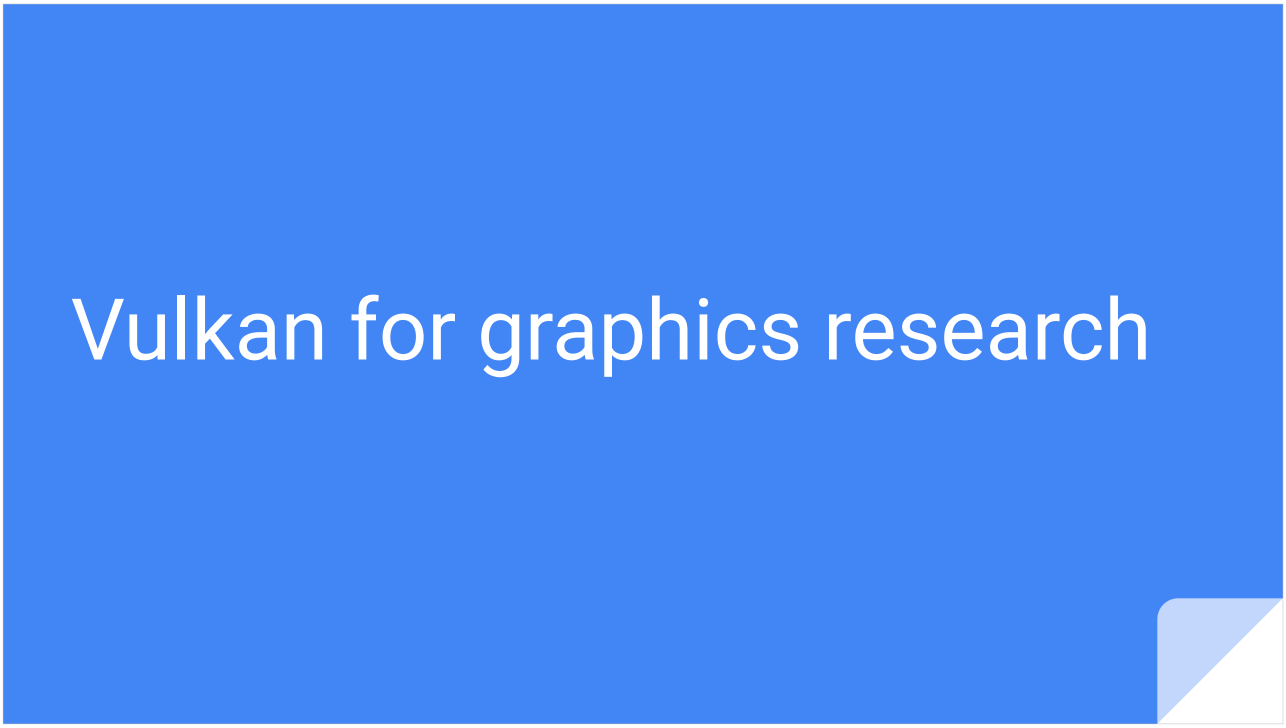


Vulkan for graphics research

Tutorial Resources

https://github.com/PacktPublishing/Mastering-Graphics-Programming-with-Vulkan

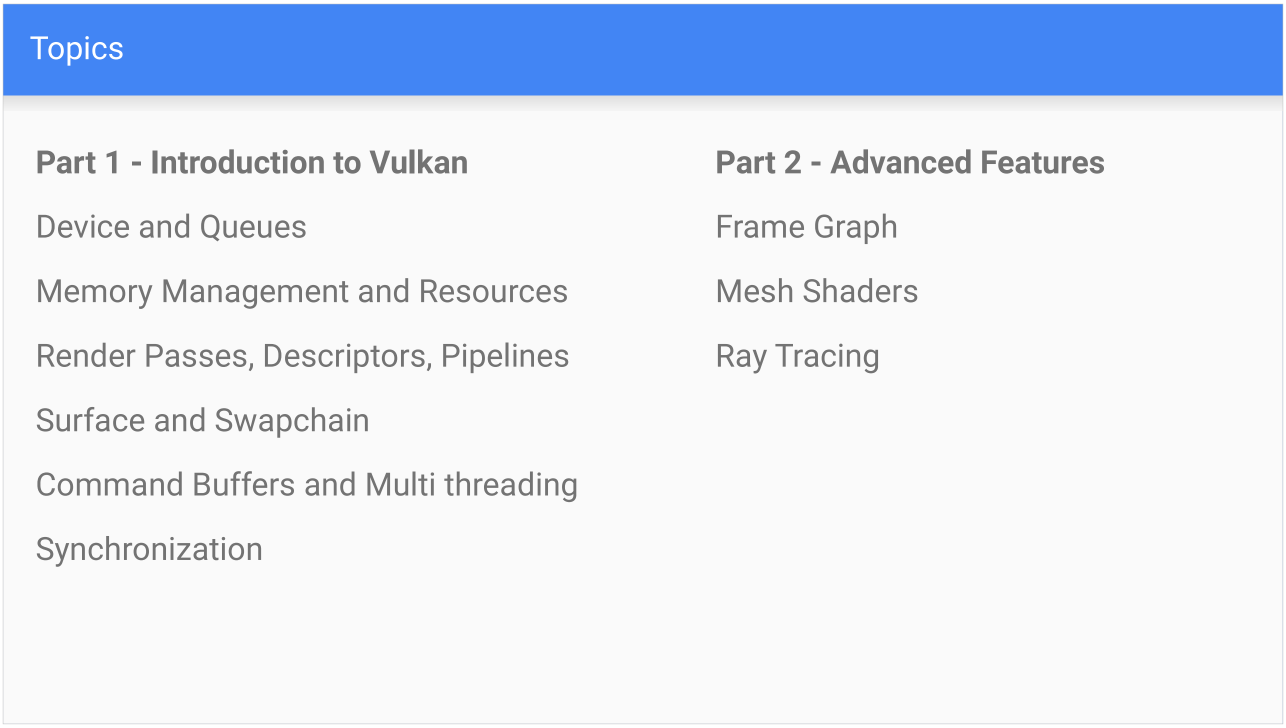
Presenters

Marco Castorina first got familiar with Vulkan while working as a driver developer at Samsung. Later he developed a 2D and 3D renderer in Vulkan from scratch for a leading media-server company. He recently joined the games graphics performance team at AMD. In his spare time, he keeps up to date with the latest techniques in real-time graphics.

Gabriel Sassone is a rendering enthusiast currently working as a Principal Rendering Engineer at Multiplayer Group. Previously working for Avalanche Studios, where his first contact with Vulkan happened, where they developed the Vulkan layer for the proprietary Apex Engine and its Google Stadia Port. He previously worked at ReadyAtDawn, Codemasters, FrameStudios, and some non-gaming tech companies. His spare time is filled with music and rendering, gaming, and outdoor activities.

Co-authors of “Mastering Graphics Programming with Vulkan”

‹packt›
1ST EDITION
Mastering
Graphics Programming
with Vulkan
Develop a modern rendering engine from first principles to state-of-the-art techniques
MARCO CASTORINA | GABRIEL SASSONE



Topics

Part 1 - Introduction to Vulkan

Device and Queues

Memory Management and Resources

Render Passes, Descriptors, Pipelines

Surface and Swapchain

Command Buffers and Multi threading

Synchronization

Part 2 - Advanced Features

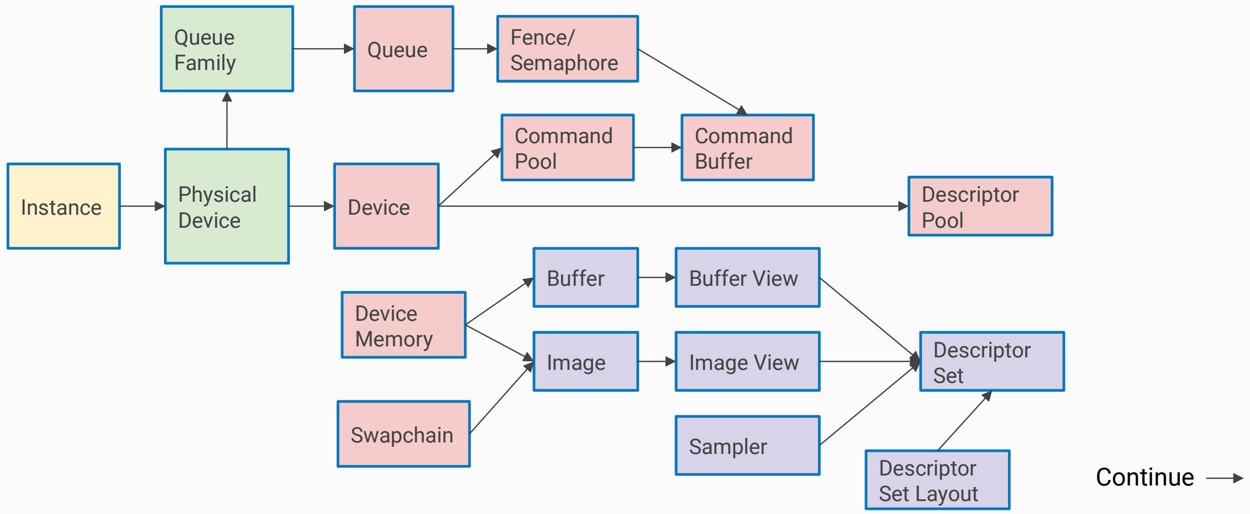
Frame Graph

Mesh Shaders

Ray Tracing

We break down each part into two 40’ sessions, so that we have time for questions in between. Before moving to the next section, could we get a show of hands if you are already familiar with Vulkan or DX12? How many are still using OpenGL? Anyone on MacOS?

Vulkan objects overview



Queue Family

Queue

Fence/ Semaphore

Command Pool

Command Buffer

Instance

Physical Device

Device

Descriptor Pool

Buffer

Buffer View

Device Memory

Image

Image View

Descriptor Set

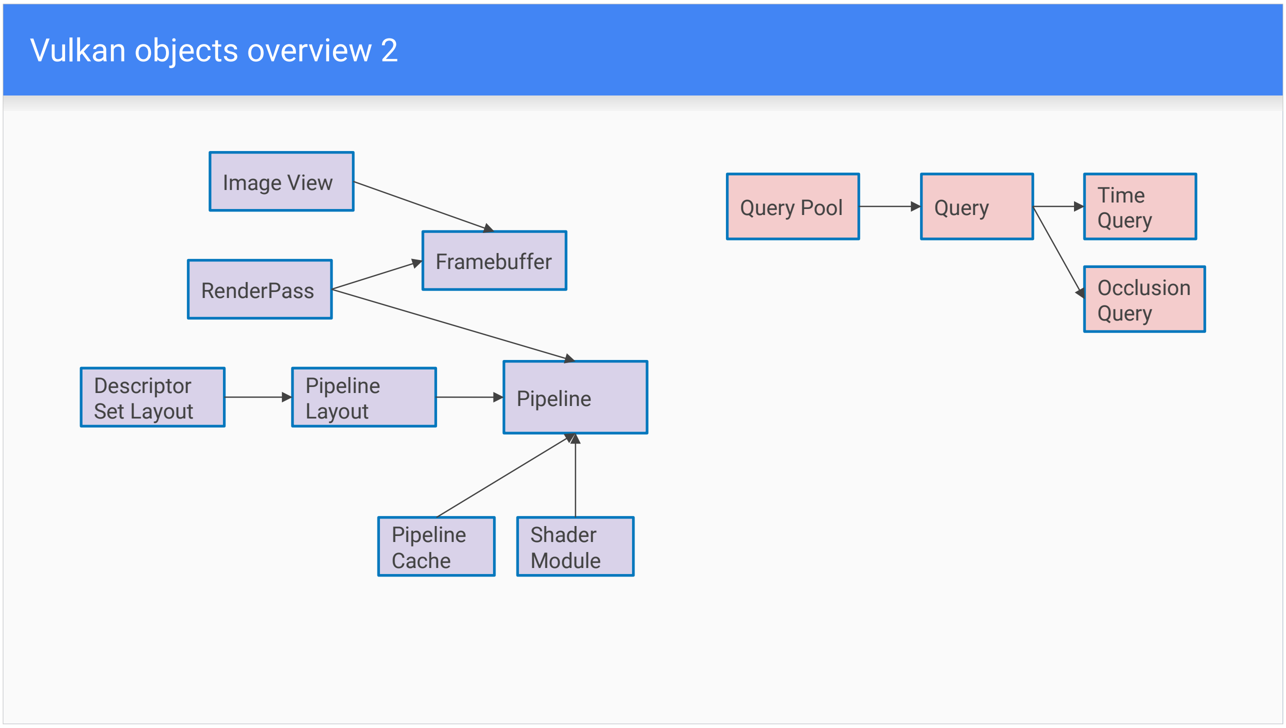
Swapchain

Sampler

Descriptor Set Layout

Continue →

It is useful to have an overview of the different objects that are used in Vulkan. There is a much finer granularity compared to older rendering APIs (both OpenGL and DirectX11 and prior versions), and we will gradually explain each of them. First there is the Instance, that is normally associated with the application: there can be only one. Then we have the physical devices, the GPUs, and Vulkan gives the flexibility to choose which GPU and how to use it. We can use multiple GPUs for different purposes. For each GPU, we create a ‘logical’ device, that will handle all the remaining Vulkan objects: command buffers, resources, swapchain and such. We are using colors to denote different areas of Vulkan objects: yellow is the Instance, green is Physical Device, red is for the logical Device, violet is for resources. In the next slide we will see the remaining objects.



Vulkan objects overview 2

Image View

Framebuffer

RenderPass

Descriptor Set Layout

Pipeline Layout

Pipeline

Pipeline Cache

Shader Module

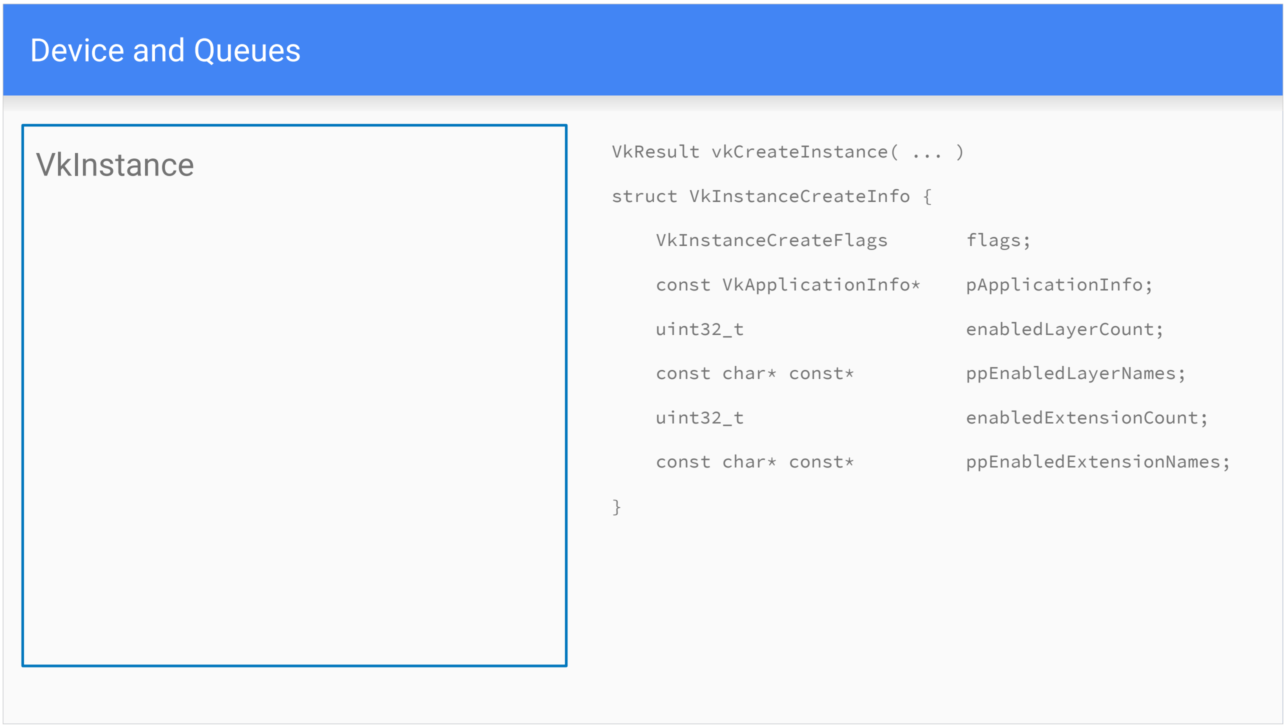
Query Pool

Query

Time Query

Occlusion Query

The rest of the objects are here: the Pipeline is the most important one, as it describes almost totally the data used by the GPU to draw/execute shaders. Pipeline includes the ShaderModule, used to describe which shaders to be used, a PipelineLayout, used to specify which resources are used and a RenderPass, used to describe which textures are used to render to when performing graphics work. Parallel to these objects there are the query objects, used to read informations from the GPU back to the CPU, like execution times (for profiling purposes), occlusion queries and statistics queries. Let’s start going deeper into each object.

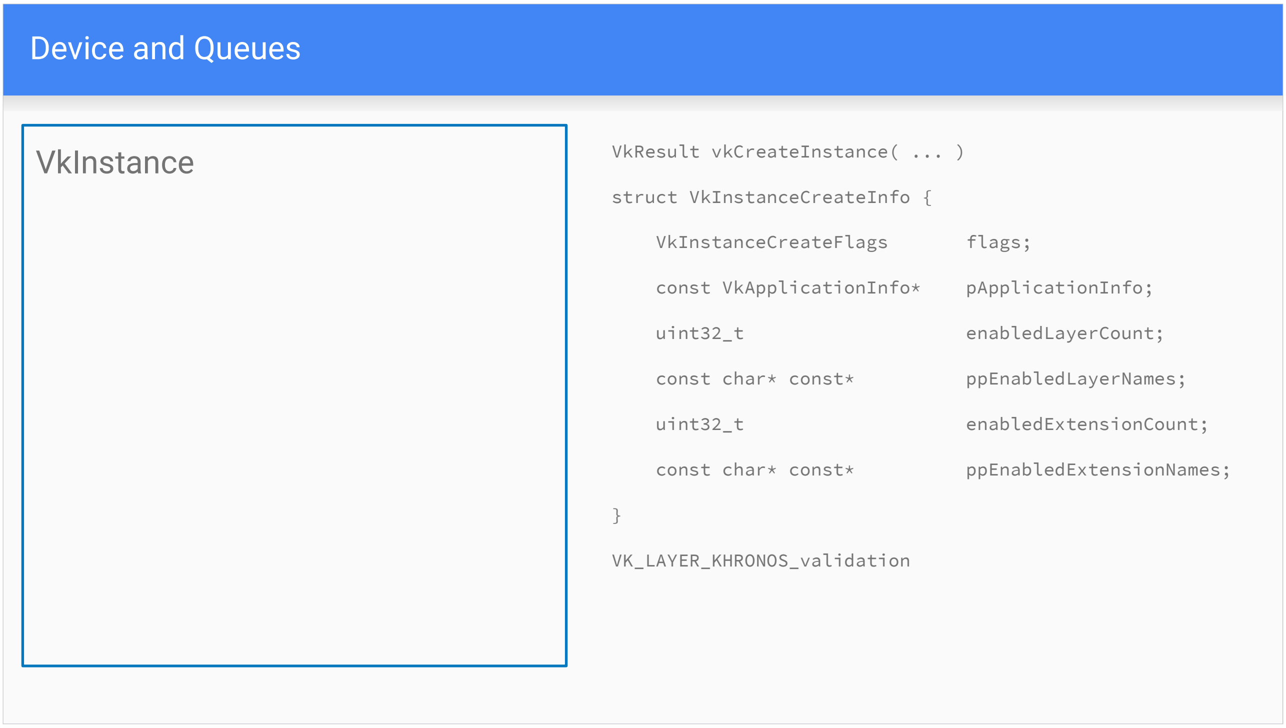


Device and Queues

VkInstance

```
VkResult vkCreateInstance( ... )

struct VkInstanceCreateInfo {

    VkInstanceCreateFlags       flags;

    const VkApplicationInfo*    pApplicationInfo;

    uint32_t                    enabledLayerCount;

    const char* const*          ppEnabledLayerNames;

    uint32_t                    enabledExtensionCount;

    const char* const*          ppEnabledExtensionNames;

}
```

In OpenGL we always had to create a context before performing any operations. Similarly in Vulkan we have to create a VkInstance: here we tell the driver which Vulkan version we want to use. This is also where we specify which instance layers and extensions we want to enable. If you want to render to screen, for instance, we need to add the surface and swapchain extensions at this point.



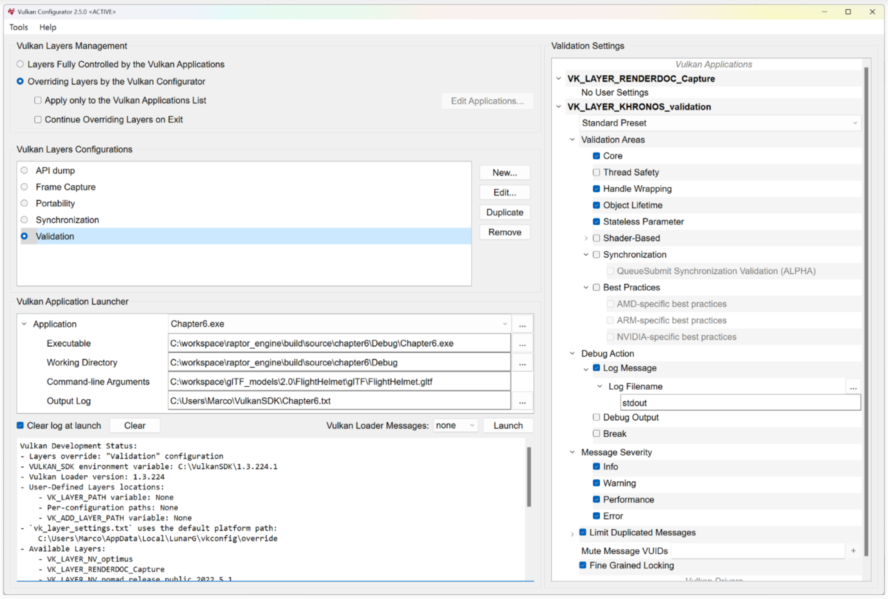
Device and Queues

VkInstance

```
VkResult vkCreateInstance( ... )

struct VkInstanceCreateInfo {

    VkInstanceCreateFlags       flags;

    const VkApplicationInfo*    pApplicationInfo;

    uint32_t                    enabledLayerCount;

    const char* const*          ppEnabledLayerNames;

    uint32_t                    enabledExtensionCount;

    const char* const*          ppEnabledExtensionNames;

}

VK_LAYER_KHRONOS_validation
```

This is also where we request to enable validation layers. With OpenGL you had to query for an error after each operation, which was tedious and error prone - and at times error messages were not very useful and only made sense after consulting the spec. They later added the option to register a callback that would trigger when an error was encountered. We can do the same in Vulkan, but if an error occurs the validation layer will always print an error to the console, so we are always notified even if we don’t manually register a callback. The error messages are also a lot more descriptive. The validation layer is essential during development and we strongly recommend you always enable it.

Device and Queues



Vulkan Layers Management

○ Layers Fully Controlled by the Vulkan Applications
● Overriding Layers by the Vulkan Configurator
□ Apply only to the Vulkan Applications List
□ Continue Overriding Layers on Exit

Vulkan Layers Configurations

○ API dump
○ Frame Capture
○ Portability
○ Synchronization
● Validation

Vulkan Application Launcher

Application	Chapter6.exe
Executable	C:\workspace\raptor_engine\build\source\chapter6\Debug\Chapter6.exe
Working Directory	C:\workspace\raptor_engine\build\source\chapter6\Debug
Command-line Arguments	C:\workspace\glTF_models\2.0\FlightHelmet\glTF\FlightHelmet.gltf
Output Log	C:\Users\Marco\VulkanSDK\Chapter6.txt

☑ Clear log at launch

Vulkan Loader Messages: none

```
Vulkan Development Status:
- Layers override: "Validation" configuration
- VULKAN_SDK environment variable: C:\VulkanSDK\1.3.224.1
- Vulkan Loader version: 1.3.224
- User-Defined Layers locations:
    - VK_LAYER_PATH variable: None
    - Per-configuration paths: None
    - VK_ADD_LAYER_PATH variable: None
- `vk_layer_settings.txt` uses the default platform path:
    C:\Users\Marco\AppData\Local\LunarG\vkconfig\override
- Available Layers:
    - VK_LAYER_NV_optimus
    - VK_LAYER_RENDERDOC_Capture
    - VK_LAYER_NV_nomad_release_public_2022_5_1
```

Validation Settings

Vulkan Applications

VK_LAYER_RENDERDOC_Capture
No User Settings

VK_LAYER_KHRONOS_validation
Standard Preset

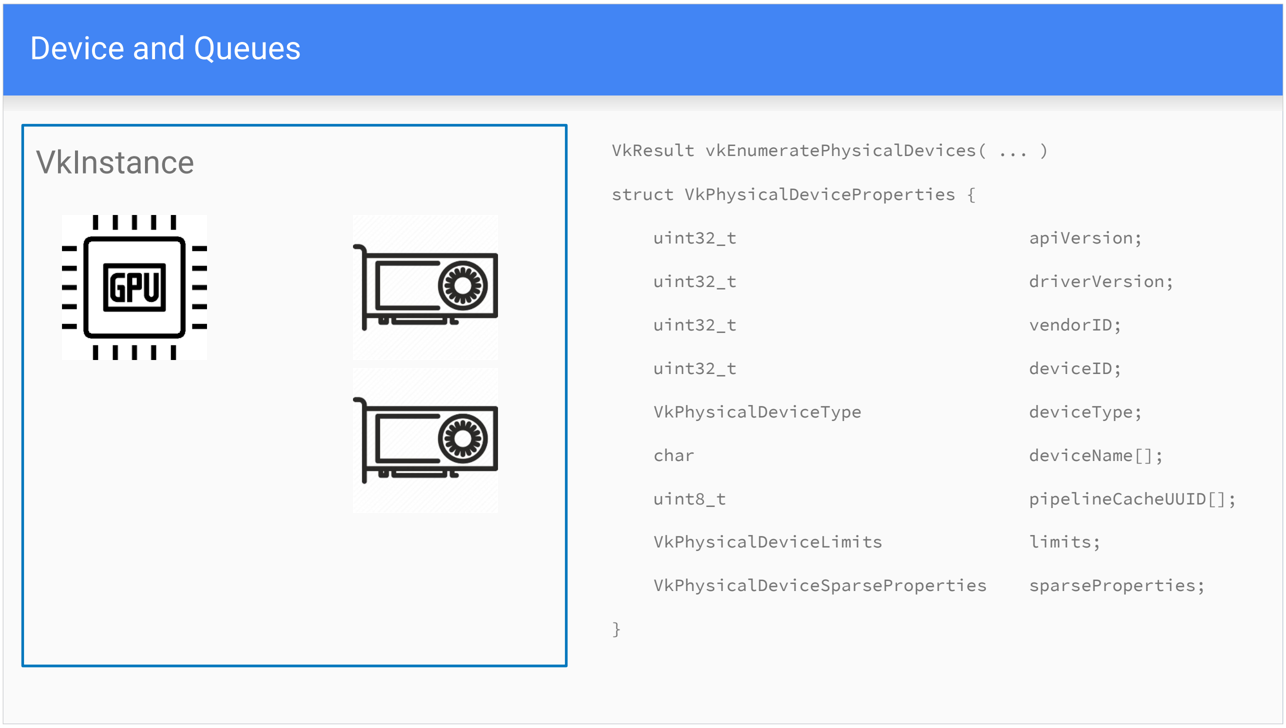
Validation Areas
☑ Core
□ Thread Safety
☑ Handle Wrapping
☑ Object Lifetime
☑ Stateless Parameter
□ Shader-Based
□ Synchronization
□ QueueSubmit Synchronization Validation (ALPHA)
□ Best Practices
□ AMD-specific best practices
□ ARM-specific best practices
□ NVIDIA-specific best practices

Debug Action
☑ Log Message
Log Filename: stdout
□ Debug Output
□ Break

Message Severity
☑ Info
☑ Warning
☑ Performance
☑ Error

☑ Limit Duplicated Messages
Mute Message VUIDs
☑ Fine Grained Locking

There is also another option to control which layers are enabled at runtime. The Vulkan SDK ships with a Vulkan configurator that makes the process easier and allows you to add or remove layers without modifying your code. We'll get into a few the other validation layers later.

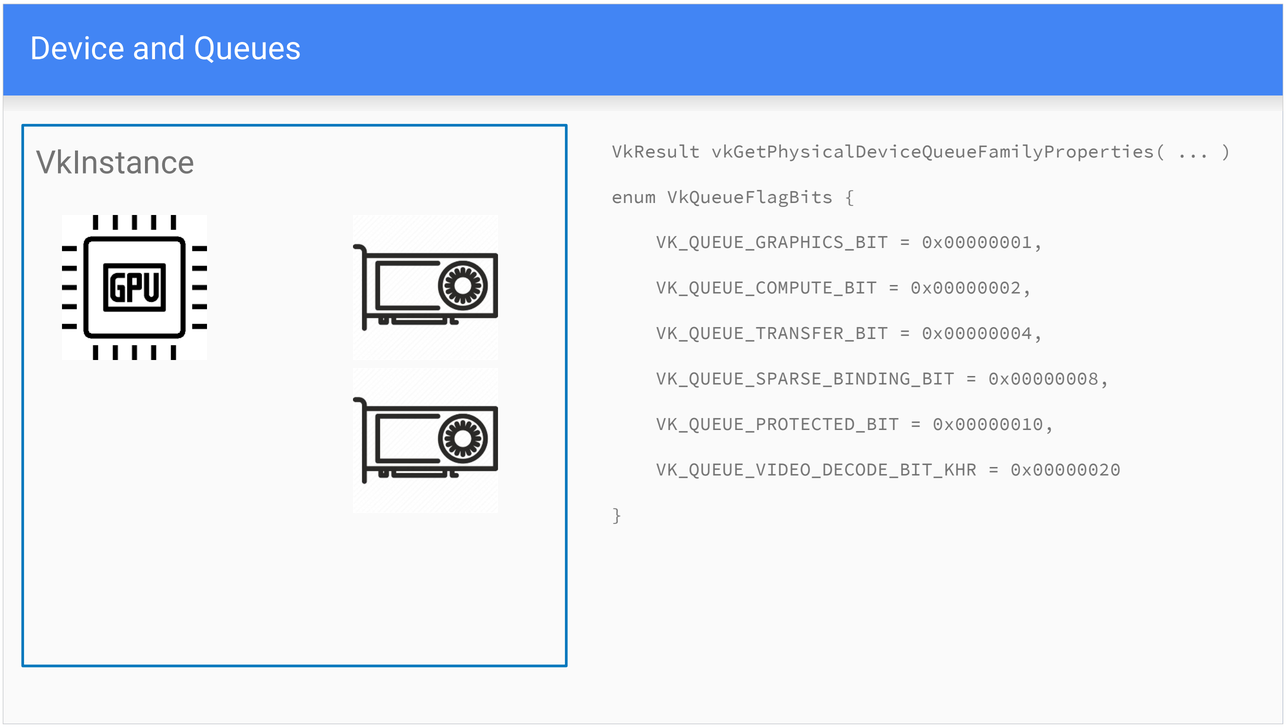


Device and Queues

VkInstance

```
VkResult vkEnumeratePhysicalDevices( ... )

struct VkPhysicalDeviceProperties {

    uint32_t                            apiVersion;

    uint32_t                            driverVersion;

    uint32_t                            vendorID;

    uint32_t                            deviceID;

    VkPhysicalDeviceType                deviceType;

    char                                deviceName[];

    uint8_t                             pipelineCacheUUID[];

    VkPhysicalDeviceLimits              limits;

    VkPhysicalDeviceSparseProperties    sparseProperties;

}
```

Your system might expose more than one device, i.e. one integrated GPU and one discrete GPU or you could have multiple discrete GPUs installed. We need to query our system to discover which devices are available so that we can choose which one we want to use. This gives you a lot of flexibility and allows to pick the preferred device (i.e. your discrete GPU). There are APIs to query which extensions and functionality each device exposes.

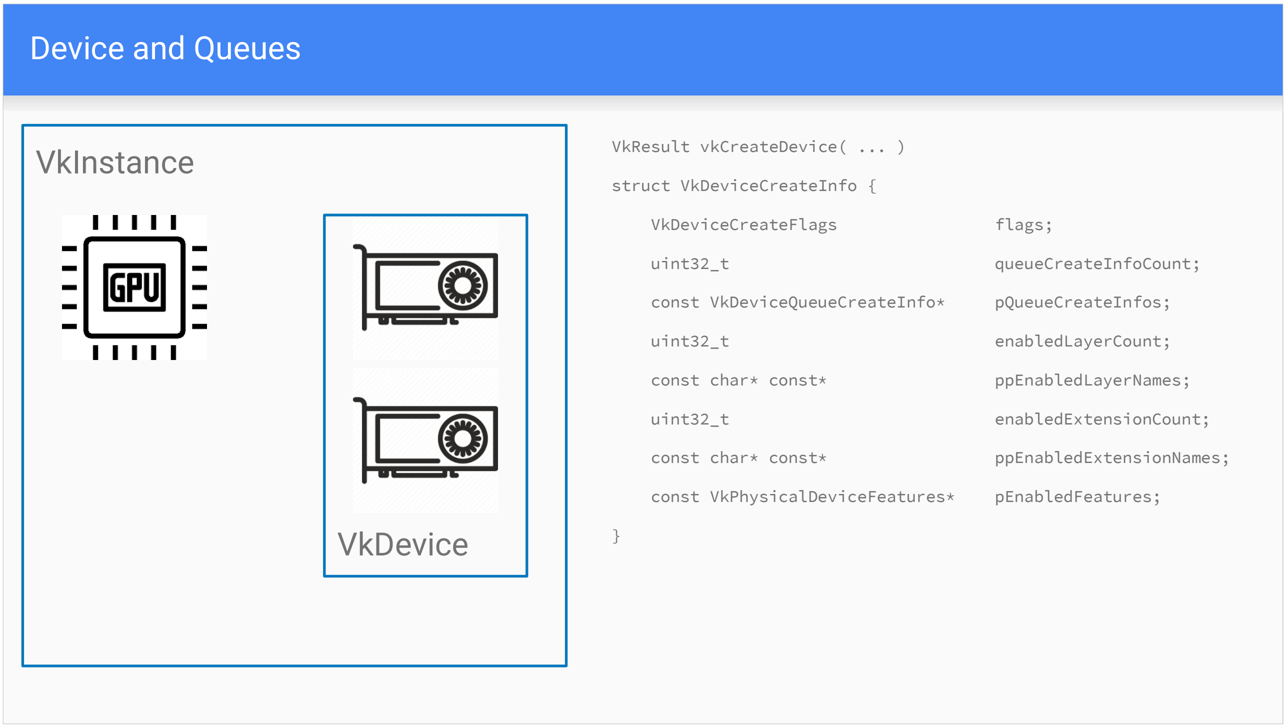


Device and Queues

VkInstance

```
VkResult vkGetPhysicalDeviceQueueFamilyProperties( ... )

enum VkQueueFlagBits {

    VK_QUEUE_GRAPHICS_BIT = 0x00000001,

    VK_QUEUE_COMPUTE_BIT = 0x00000002,

    VK_QUEUE_TRANSFER_BIT = 0x00000004,

    VK_QUEUE_SPARSE_BINDING_BIT = 0x00000008,

    VK_QUEUE_PROTECTED_BIT = 0x00000010,

    VK_QUEUE_VIDEO_DECODE_BIT_KHR = 0x00000020

}
```

Next, we have to determine which queues a physical device exposes. These are the objects that we are going to use to submit work to the GPU. Devices expose different queue types and for each type a device might expose multiple queues. Unfortunately it’s not clear what are the advantages of using more than queue per type, other than maybe not having to synchronize access to the queue. We’ll talk more about this in the section on synchronization.



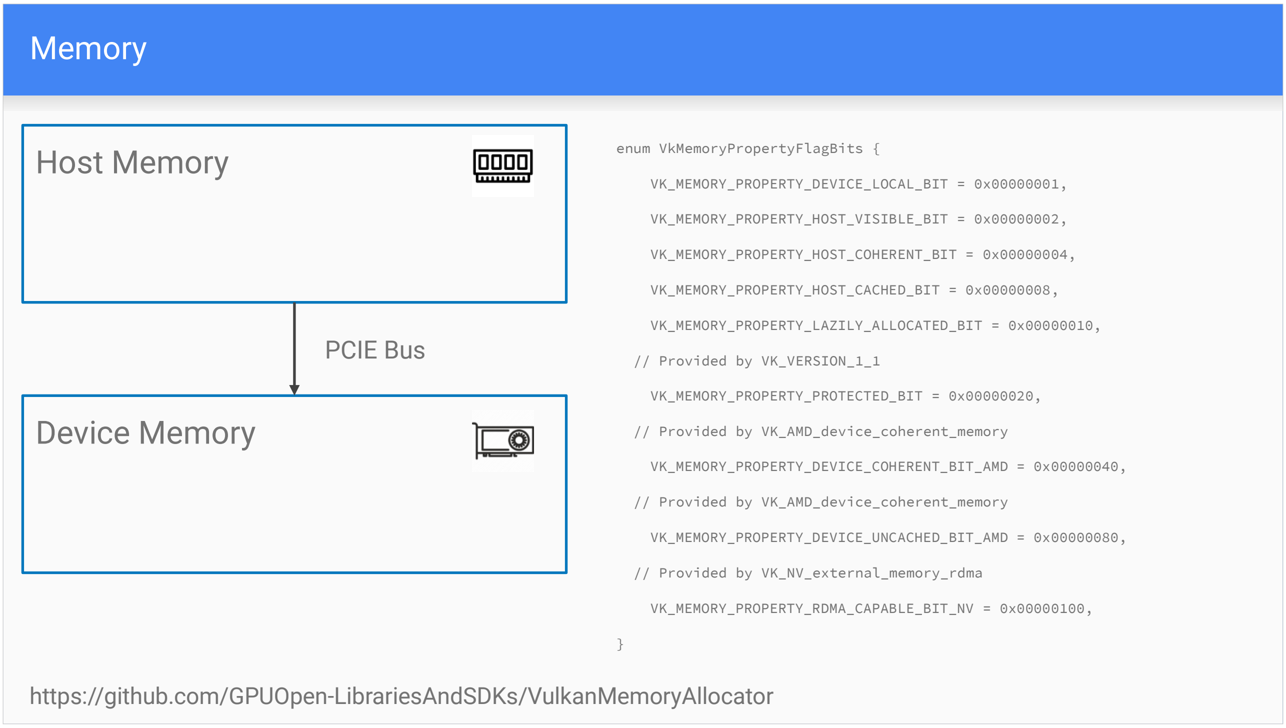
Device and Queues

VkInstance

VkDevice

```
VkResult vkCreateDevice( ... )

struct VkDeviceCreateInfo {

    VkDeviceCreateFlags              flags;

    uint32_t                         queueCreateInfoCount;

    const VkDeviceQueueCreateInfo*   pQueueCreateInfos;

    uint32_t                         enabledLayerCount;

    const char* const*               ppEnabledLayerNames;

    uint32_t                         enabledExtensionCount;

    const char* const*               ppEnabledExtensionNames;

    const VkPhysicalDeviceFeatures*  pEnabledFeatures;

}
```

Some of you might ask why do we need to know about physical devices. There are a couple of reasons: this allows you to leverage all the available devices on your system: you could for instance offload some work to the integrated GPU and then pass the result to your discrete device. The other use case is to treat multiple GPUs a single logical device. As you can see, we can also enable layers and extensions per device. We need this because different devices might support different extensions and we can control which ones get enabled per device. Always remember to query for support before using an extension. The validation layer will remind you if you forget :)



Memory

Host Memory

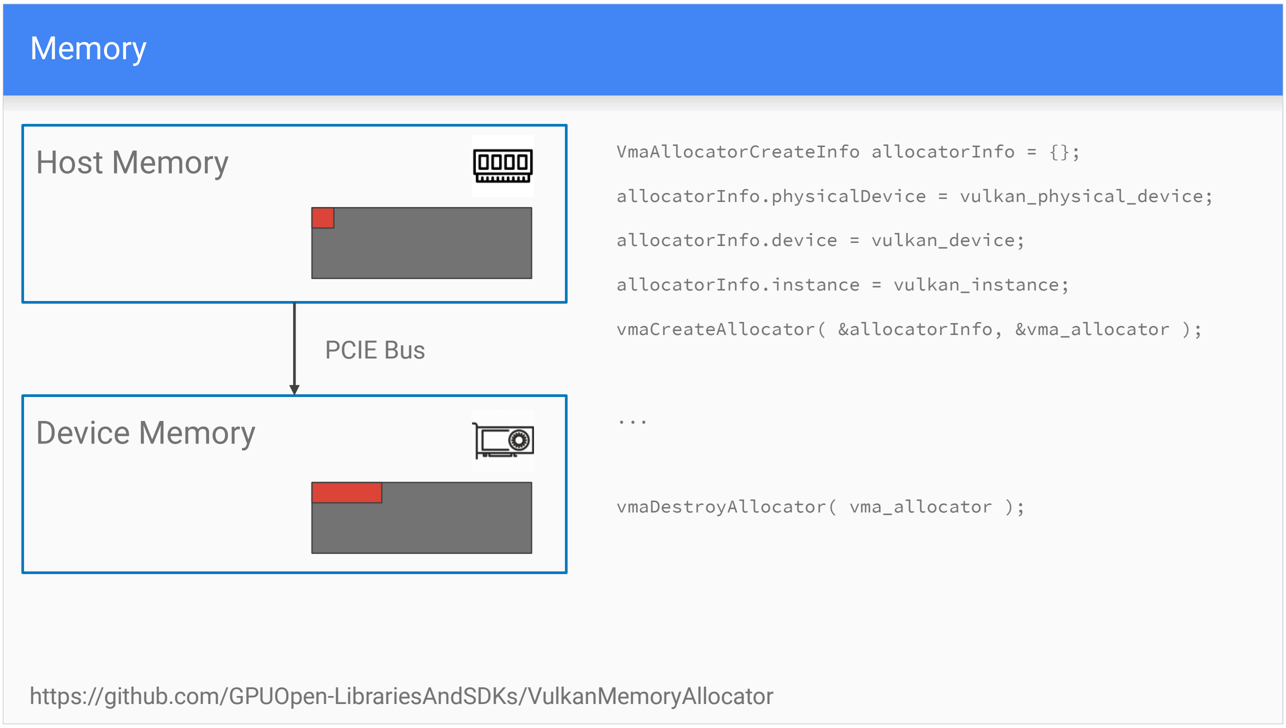
PCIE Bus

Device Memory

```
enum VkMemoryPropertyFlagBits {
    VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT = 0x00000001,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT = 0x00000002,
    VK_MEMORY_PROPERTY_HOST_COHERENT_BIT = 0x00000004,
    VK_MEMORY_PROPERTY_HOST_CACHED_BIT = 0x00000008,
    VK_MEMORY_PROPERTY_LAZILY_ALLOCATED_BIT = 0x00000010,
  // Provided by VK_VERSION_1_1
    VK_MEMORY_PROPERTY_PROTECTED_BIT = 0x00000020,
  // Provided by VK_AMD_device_coherent_memory
    VK_MEMORY_PROPERTY_DEVICE_COHERENT_BIT_AMD = 0x00000040,
  // Provided by VK_AMD_device_coherent_memory
    VK_MEMORY_PROPERTY_DEVICE_UNCACHED_BIT_AMD = 0x00000080,
  // Provided by VK_NV_external_memory_rdma
    VK_MEMORY_PROPERTY_RDMA_CAPABLE_BIT_NV = 0x00000100,
}
```

https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator

In OpenGL memory was managed for you by the driver. Vulkan allows developers to have much finer control over when memory is allocated an how it’s used. We don’t recommend writing your own memory allocator unless that’s something you would like to experiment with. We suggest you use the Vulkan Memory Allocator, which is the de-facto standard for managing memory for Vukan applications. We still provide an overview of the details so that you know what happens under the hood. We describe how memory works for a system with a discrete GPU. Things are a bit different for mobile and integrated GPUs, which share the same physical memory. Host memory is basically your RAM, while device memory is the memory available on your GPU. Device memory is usually not accessible directly. We need to create resources in host-coherent memory and then copy it to a resource backed by device local memory. On modern systems you can enable ReBAR (Resizable BAR), which allows you to copy data directly into GPU memory, thus saving the cost of a copy.



Memory

Host Memory

PCIE Bus

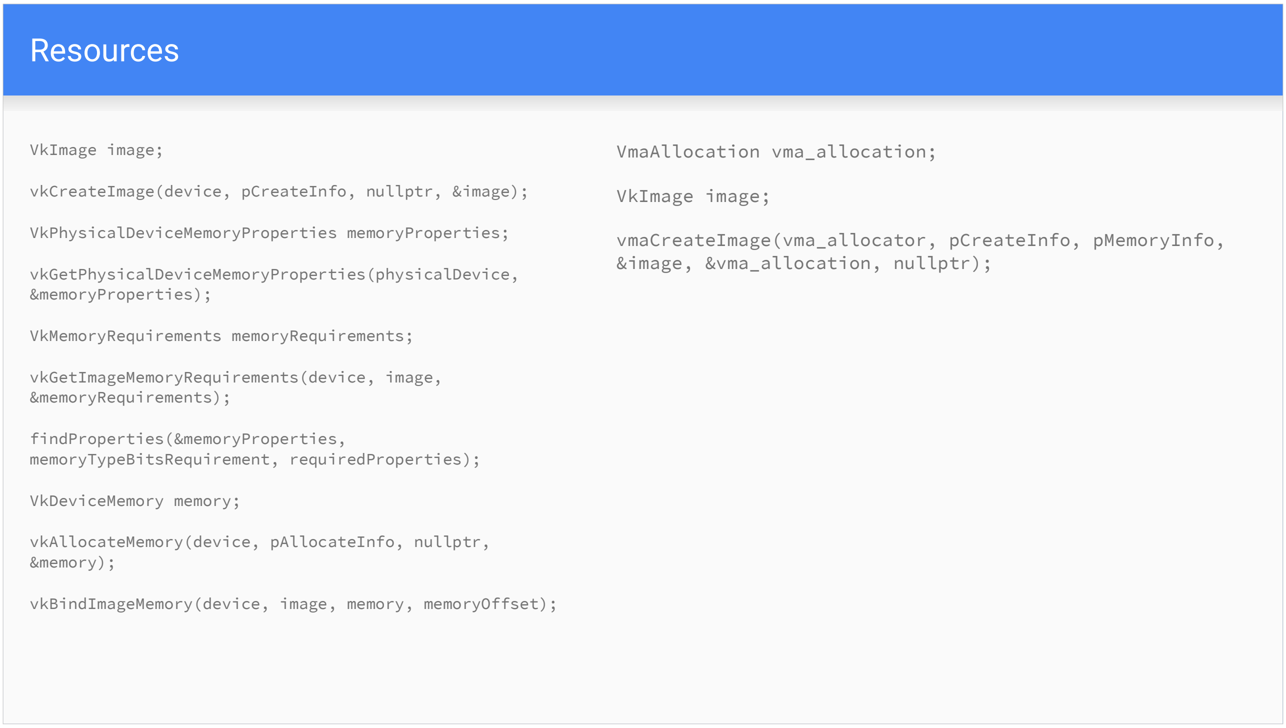
Device Memory

```
VmaAllocatorCreateInfo allocatorInfo = {};

allocatorInfo.physicalDevice = vulkan_physical_device;

allocatorInfo.device = vulkan_device;

allocatorInfo.instance = vulkan_instance;

vmaCreateAllocator( &allocatorInfo, &vma_allocator );


...


vmaDestroyAllocator( vma_allocator );
```

https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator

Under the hood, VMA creates multiple memory pools that it sub-allocates from. Allocating memory on the GPU can be a slow operation and If you were to implement this naively your application would perform poorly. Like with a CPU memory pool, this also allows you to reuse memory that has been freed. VMA also also implements some useful features like resource tracking so that it can warn you if some resources haven’t been freed when the application terminates.

Here we show how to initialize the VMA allocator. At this point we have already created all the Vulkan objects we need. Before the application terminates we need to destroy the allocator. This follows the Vulkan paradigm nicely.

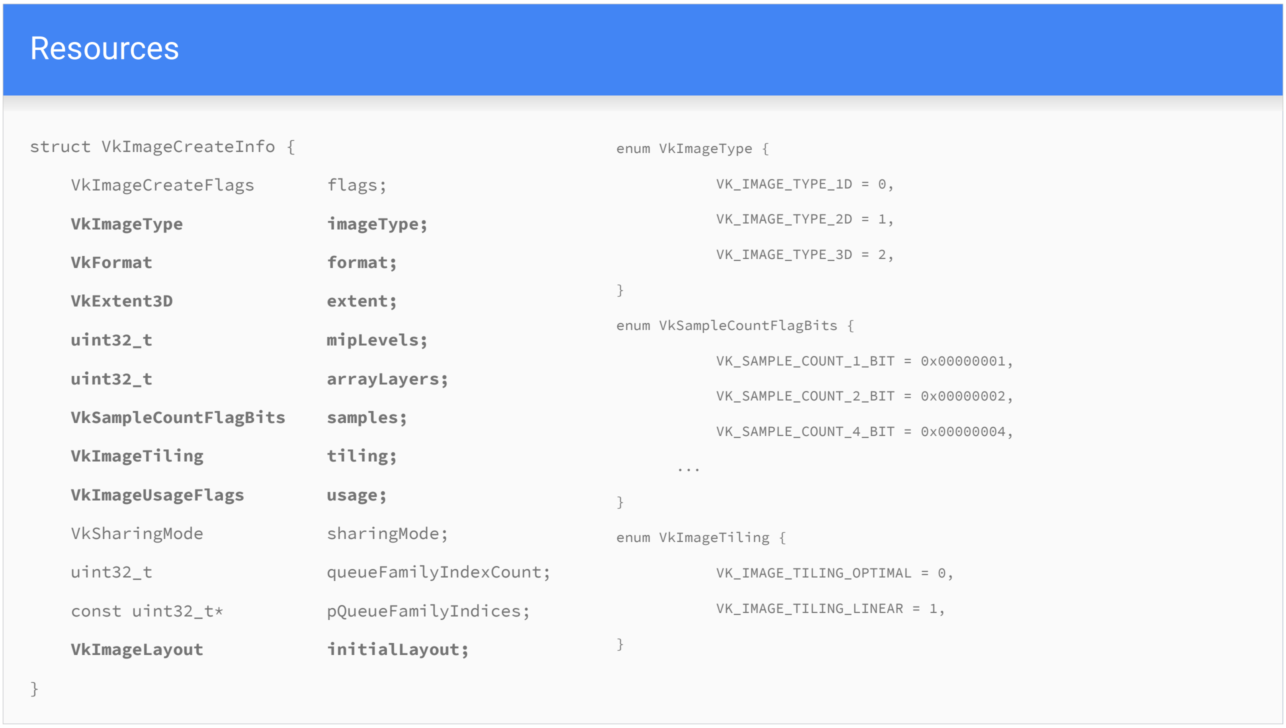


Resources

```
VkImage image;

vkCreateImage(device, pCreateInfo, nullptr, &image);

VkPhysicalDeviceMemoryProperties memoryProperties;

vkGetPhysicalDeviceMemoryProperties(physicalDevice,
&memoryProperties);

VkMemoryRequirements memoryRequirements;

vkGetImageMemoryRequirements(device, image,
&memoryRequirements);

findProperties(&memoryProperties,
memoryTypeBitsRequirement, requiredProperties);

VkDeviceMemory memory;

vkAllocateMemory(device, pAllocateInfo, nullptr,
&memory);

vkBindImageMemory(device, image, memory, memoryOffset);
```

```
VmaAllocation vma_allocation;

VkImage image;

vmaCreateImage(vma_allocator, pCreateInfo, pMemoryInfo,
&image, &vma_allocation, nullptr);
```

VMA simplifies greatly the code we have to write to create resources (image and buffers). On the left we show all the steps we would have to perform manually to create a resource - thankfully all of this is handled by VMA for us and all we have to do is use the code on the right.



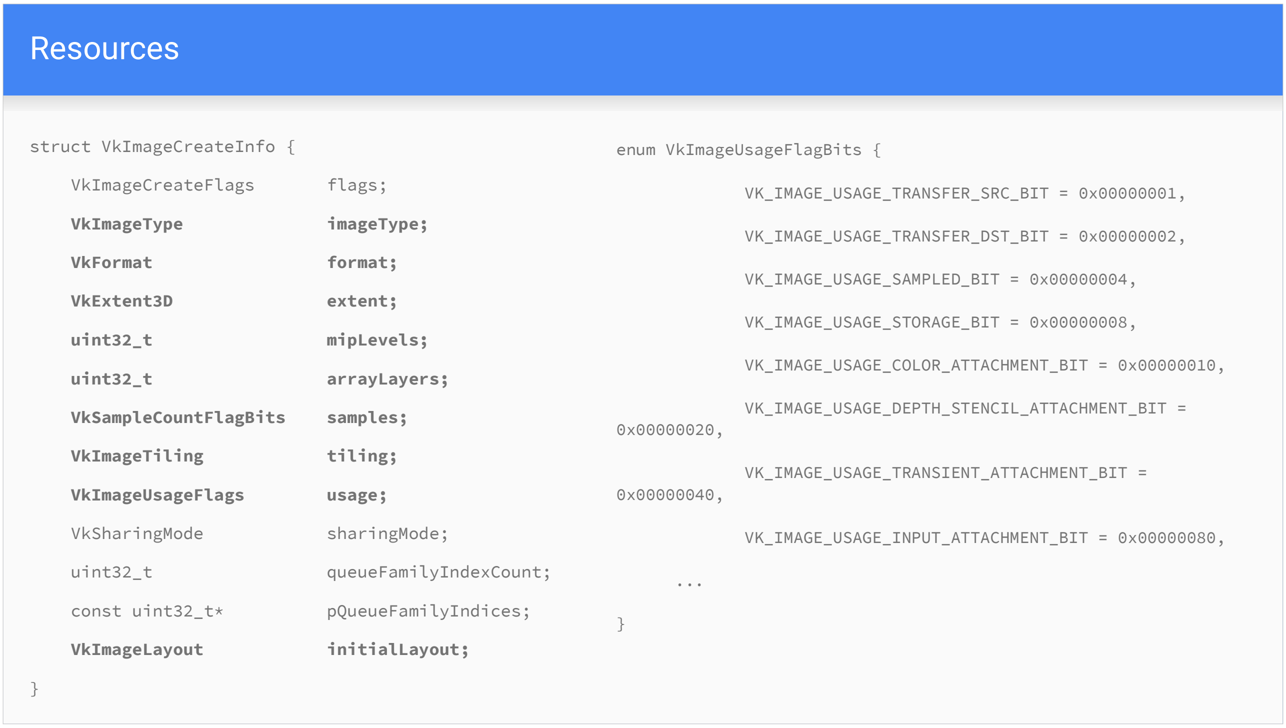
Resources

```
struct VkImageCreateInfo {
    VkImageCreateFlags        flags;
    VkImageType               imageType;
    VkFormat                  format;
    VkExtent3D                extent;
    uint32_t                  mipLevels;
    uint32_t                  arrayLayers;
    VkSampleCountFlagBits     samples;
    VkImageTiling             tiling;
    VkImageUsageFlags         usage;
    VkSharingMode             sharingMode;
    uint32_t                  queueFamilyIndexCount;
    const uint32_t*           pQueueFamilyIndices;
    VkImageLayout             initialLayout;
}
```

```
enum VkImageType {
            VK_IMAGE_TYPE_1D = 0,
            VK_IMAGE_TYPE_2D = 1,
            VK_IMAGE_TYPE_3D = 2,
}
enum VkSampleCountFlagBits {
            VK_SAMPLE_COUNT_1_BIT = 0x00000001,
            VK_SAMPLE_COUNT_2_BIT = 0x00000002,
            VK_SAMPLE_COUNT_4_BIT = 0x00000004,
       ...
}
enum VkImageTiling {
            VK_IMAGE_TILING_OPTIMAL = 0,
            VK_IMAGE_TILING_LINEAR = 1,
}
```

We still have to define the properties of our image. This structure is quite large, but really it’s quite simple. We have highlighted the fields that you will likely need most of the the time.

- imageType defines the dimensions of your image
- format is self-explanatory, although we need to make sure the combination of flags is valid for this format. We’ll explain in a moment
- extent is the size of your image. It has 3 dimensions because you can define 3D images as well as 2D and 1D image
- mipLevels is the number of mipmaps we want to create for this resource
- arrayLayers is the number of entries, for instance, for an image array
- samples determines how many samples we want in our image
- tiling is quite important: it determines how the image is stored in device memory. Optimal is the one you want. To upload a texture to the GPU we recommend copying the data to a host visible buffer and then use vkCmdCopyBufferToImage to upload it to the GPU. It also possible to create a linear image and copy it to an optimal image.



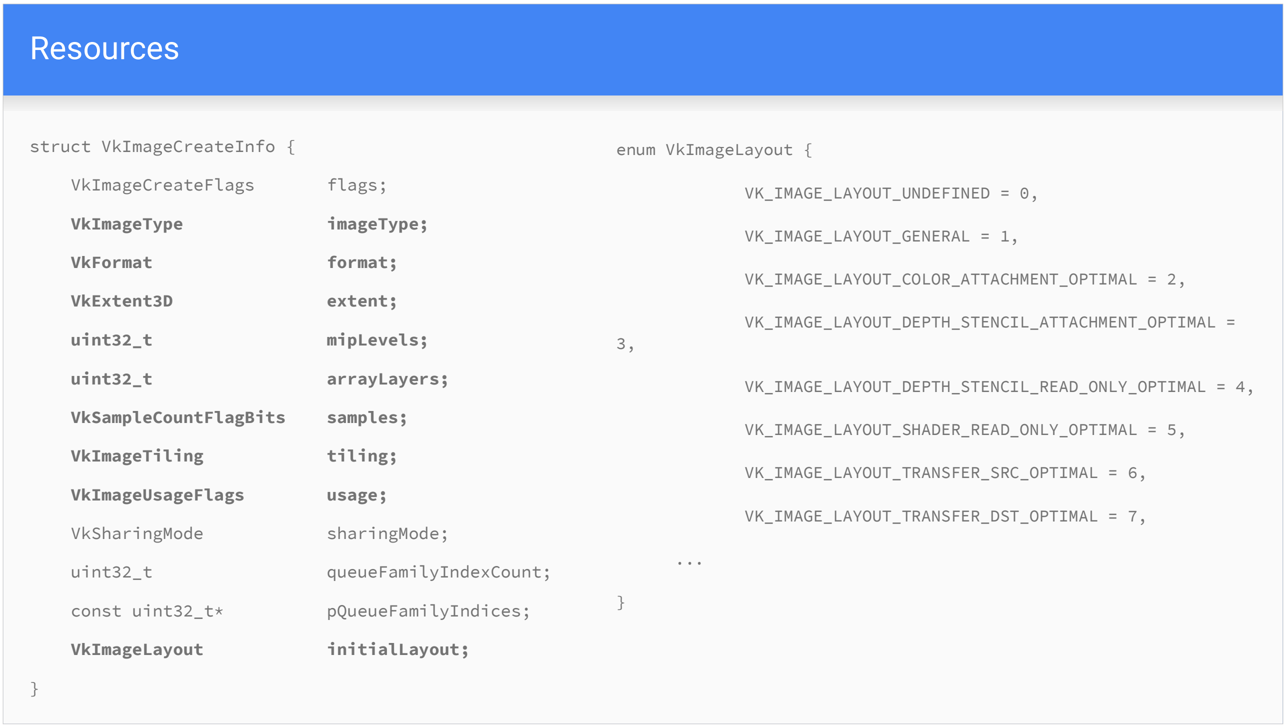
Resources

```
struct VkImageCreateInfo {
    VkImageCreateFlags       flags;
    VkImageType              imageType;
    VkFormat                 format;
    VkExtent3D               extent;
    uint32_t                 mipLevels;
    uint32_t                 arrayLayers;
    VkSampleCountFlagBits    samples;
    VkImageTiling            tiling;
    VkImageUsageFlags        usage;
    VkSharingMode            sharingMode;
    uint32_t                 queueFamilyIndexCount;
    const uint32_t*          pQueueFamilyIndices;
    VkImageLayout            initialLayout;
}
```

```
enum VkImageUsageFlagBits {
             VK_IMAGE_USAGE_TRANSFER_SRC_BIT = 0x00000001,
             VK_IMAGE_USAGE_TRANSFER_DST_BIT = 0x00000002,
             VK_IMAGE_USAGE_SAMPLED_BIT = 0x00000004,
             VK_IMAGE_USAGE_STORAGE_BIT = 0x00000008,
             VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT = 0x00000010,
             VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT =
0x00000020,
             VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT =
0x00000040,
             VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT = 0x00000080,
      ...
}
```

Usage tells the driver how we plan to use this image, for instance whether we just want to use it as a texture or if we are also going to render into it (color attachment). It’s important to get this right as some drivers can optimize how the image is stored in the GPU. Validation layers are really helpful in this case as they will warn you if you are trying to use a resource in a way that wasn’t specified at creation time.

You have to check that your combination of usage and tiling flags is valid for the given format. The spec has some mandatory formats, but the best approach is to always check for support.



Resources

```
struct VkImageCreateInfo {
    VkImageCreateFlags        flags;
    VkImageType               imageType;
    VkFormat                  format;
    VkExtent3D                extent;
    uint32_t                  mipLevels;
    uint32_t                  arrayLayers;
    VkSampleCountFlagBits     samples;
    VkImageTiling             tiling;
    VkImageUsageFlags         usage;
    VkSharingMode             sharingMode;
    uint32_t                  queueFamilyIndexCount;
    const uint32_t*           pQueueFamilyIndices;
    VkImageLayout             initialLayout;
}
```

```
enum VkImageLayout {
            VK_IMAGE_LAYOUT_UNDEFINED = 0,
            VK_IMAGE_LAYOUT_GENERAL = 1,
            VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL = 2,
            VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL =
3,
            VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL = 4,
            VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL = 5,
            VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL = 6,
            VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL = 7,
      ...
}
```

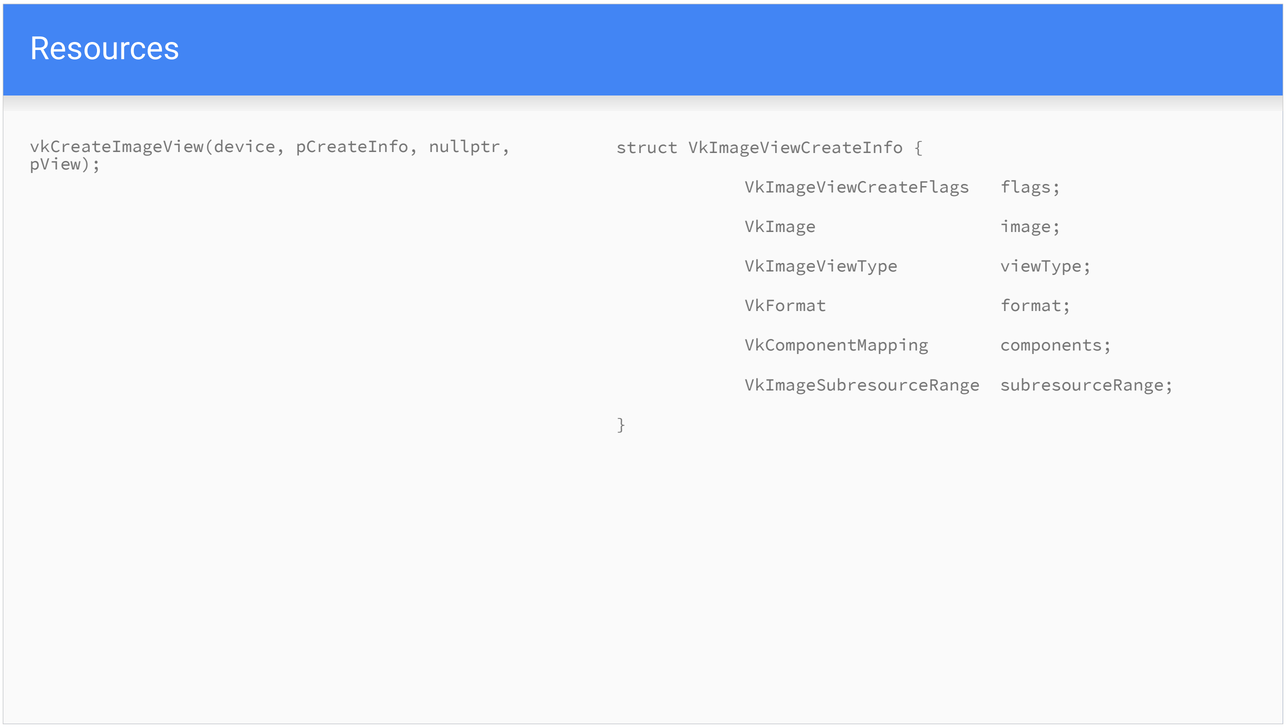
Finally, image layout defines the initial state of this resource. We are going to go into more details about image layout transitions in the synchronization section.

We are going to mention image layouts a few times during this presentation as they are an important aspect when using Vulkan.

Images can be used in many ways: we can render into them, we can read from them as sampled textures, we can copy them, we can write and read to them in compute shaders and a few other use cases.

Each of these use cases require the image to be in the correct layout. When going from writing to reading an image, for instance, the driver has to instruct the GPU to flush its caches so that the final image data is in main memory and can be read correctly in the next stage.

We are going to cover how to perform this transition in the section on synchronization.



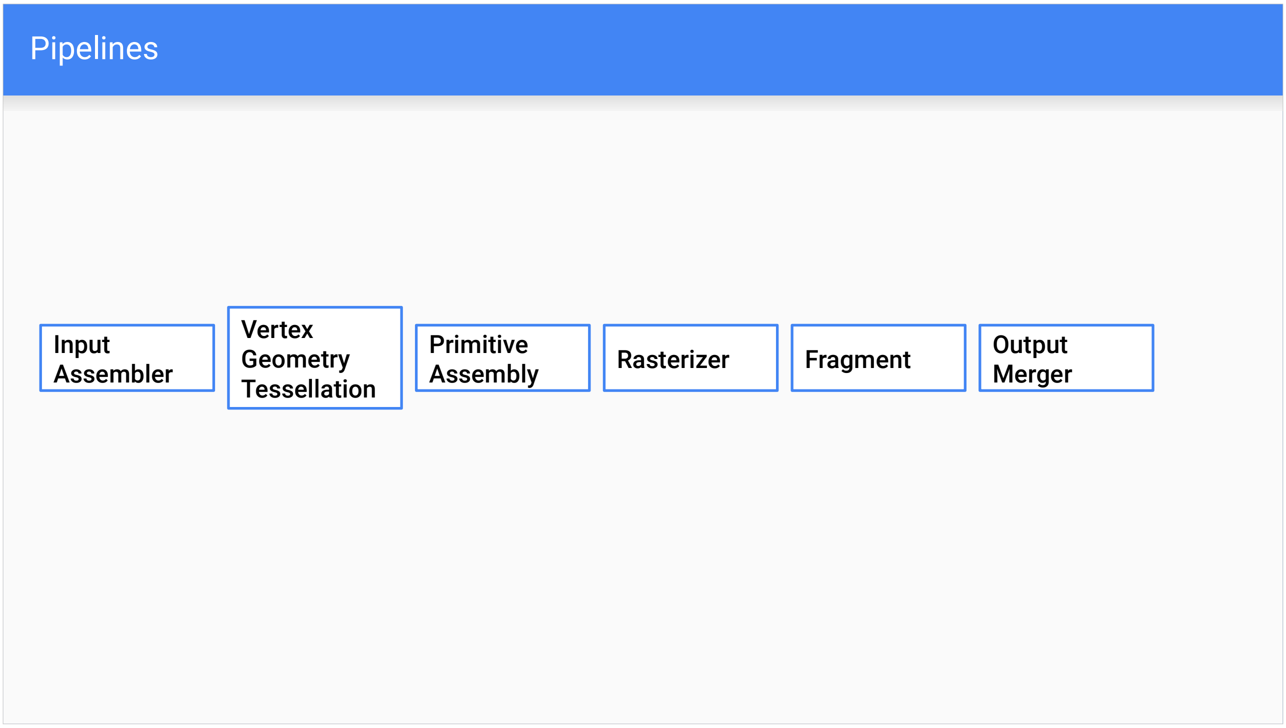
Resources

```
vkCreateImageView(device, pCreateInfo, nullptr,
pView);
```

```
struct VkImageViewCreateInfo {
            VkImageViewCreateFlags    flags;
            VkImage                   image;
            VkImageViewType           viewType;
            VkFormat                  format;
            VkComponentMapping        components;
            VkImageSubresourceRange   subresourceRange;
}
```

Once you have created an image, you need also to create an image view. This is needed, for instance, when using a resource in a shader. The view determines how we want to access the underlying resource. You might want to access only a sub-region of the image, or you might want to access it with a different format (as long as it’s compatible with the underlying image format), you can also change the channel order or you can limit the number of mips that are used.

And with this we conclude the section on resources. Buffers work pretty much in the same way, although they are a lot simpler, as we don’t have to worry about format and layout. We still need to be careful about the usage flags though.



Pipelines

Input Assembler

Vertex Geometry Tessellation

Primitive Assembly

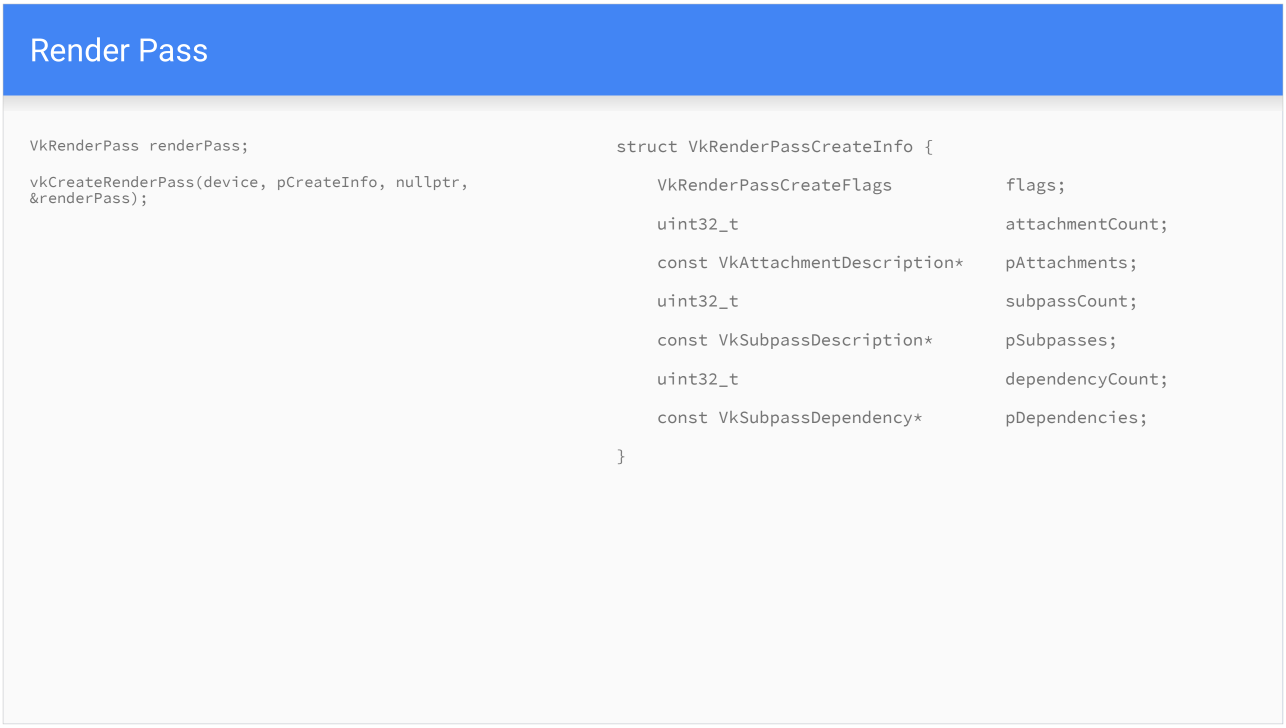
Rasterizer

Fragment

Output Merger

Before proceeding to describe how to create and use pipelines, we provide a brief overview of how the underlying HW is organized. This will help to understand why the Vulkan API is structured the way it is.

1. The input assembler is responsible for reading indices and feeding the corresponding vertices to the next stage
2. Vertex, Geometry and Tessellation shaders process vertices to provide the final position in normalized device coordinates (NDC). Geometry and Tessellation shaders can also amplify the original geometry to create new primitives
3. The primitive assembly takes individual vertices and combines them into geometry primitives (triangles strips, fans, lines, etc.)
4. These primitives are then fed to the rasterizer, which will determine which fragments these primitives covers
5. The fragments that are covered by the primitive are then processed by the fragment shader
6. Finally the output merger is responsible for storing the final color value to the bound render targets. It’s also responsible for blending (if blending is enabled), writing depth, etc.



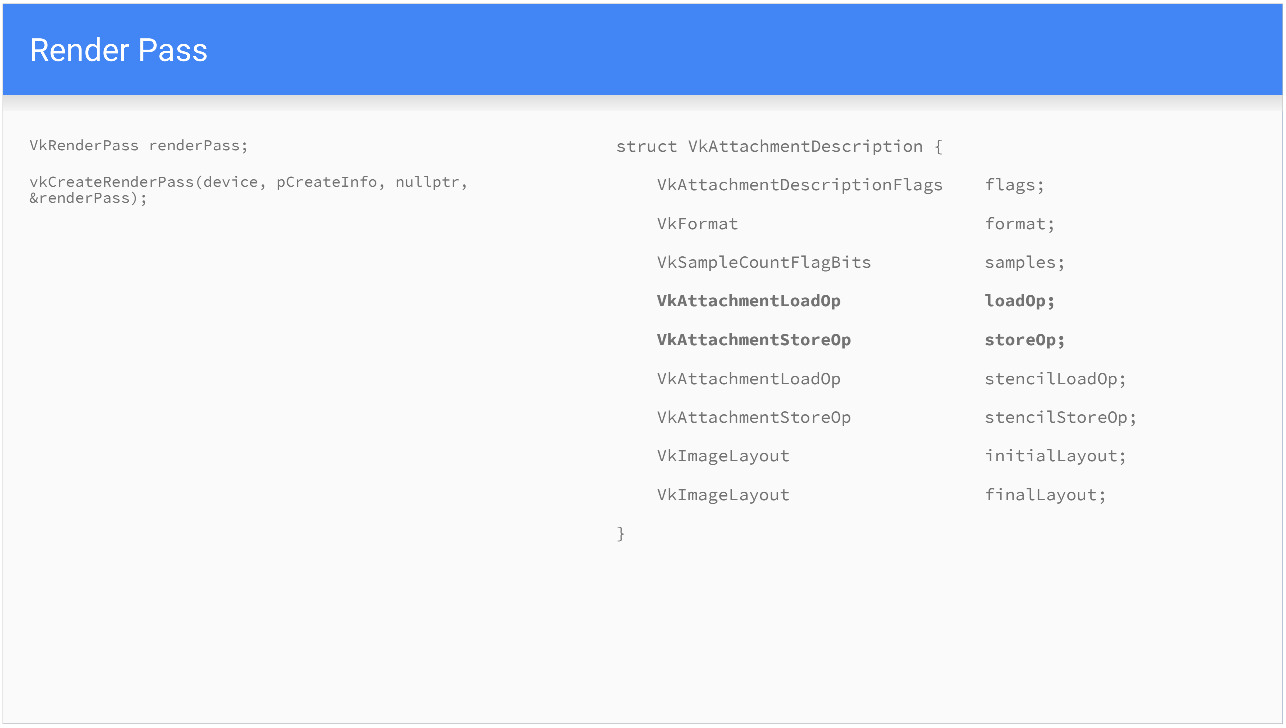
Render Pass

```
VkRenderPass renderPass;

vkCreateRenderPass(device, pCreateInfo, nullptr,
&renderPass);
```

```
struct VkRenderPassCreateInfo {
    VkRenderPassCreateFlags          flags;
    uint32_t                         attachmentCount;
    const VkAttachmentDescription*   pAttachments;
    uint32_t                         subpassCount;
    const VkSubpassDescription*      pSubpasses;
    uint32_t                         dependencyCount;
    const VkSubpassDependency*       pDependencies;
}
```

Vulkan requires we provide a fair bit of information up front. This is needed to avoid state changes during rendering, which can be quite expensive. Compared to OpenGL, this is a lot more rigid, although we'll see later that some state can still be changed at runtime. The API is organized this way to keep the driver as thin as possible. This also pushes you to organize the order in which your programs are used for maximum performance. One of the first objects that we need is called a render pass. A render pass simply describes the render target(s) we are going to render into.

As you can see here, we need to provide the number of attachments (render targets) and also sub-passes. Sub-passes are rarely used outside of mobile and we are not going to cover them here.

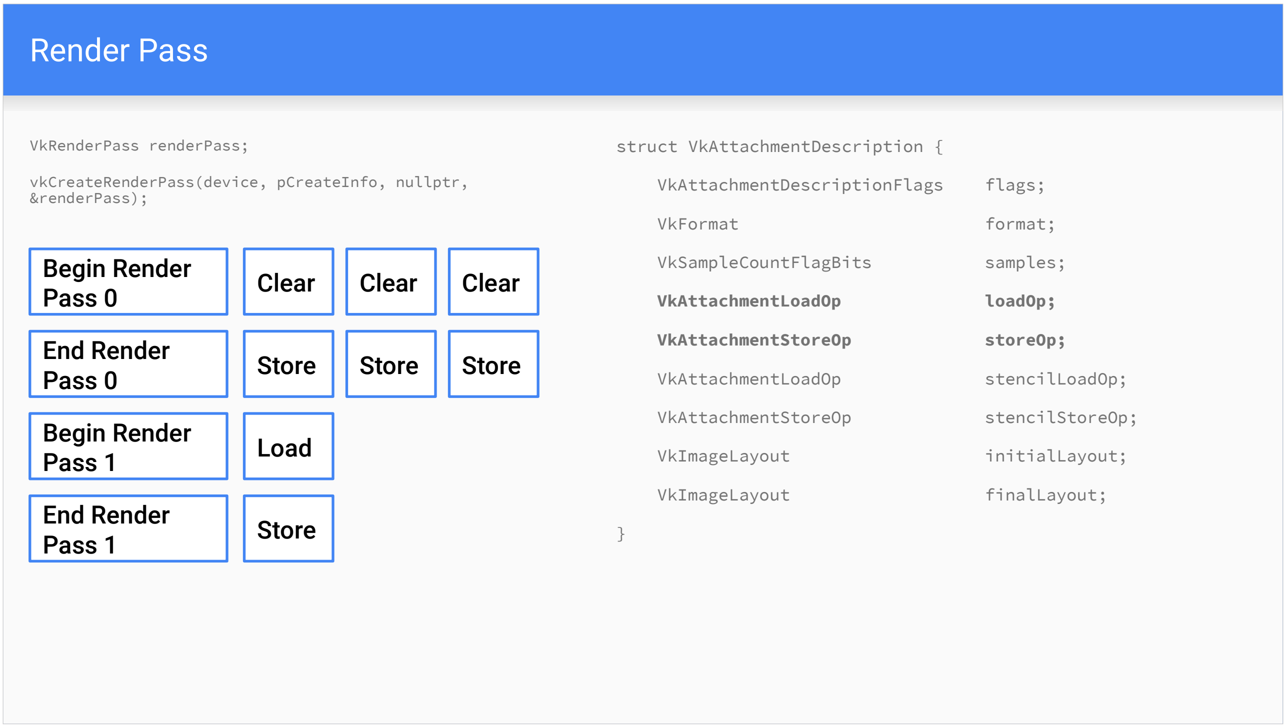


Render Pass

```
VkRenderPass renderPass;

vkCreateRenderPass(device, pCreateInfo, nullptr,
&renderPass);
```

```
struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags    flags;
    VkFormat                        format;
    VkSampleCountFlagBits           samples;
    VkAttachmentLoadOp              loadOp;
    VkAttachmentStoreOp             storeOp;
    VkAttachmentLoadOp              stencilLoadOp;
    VkAttachmentStoreOp             stencilStoreOp;
    VkImageLayout                   initialLayout;
    VkImageLayout                   finalLayout;
}
```

The attachment description has a few fields, but we have seen most of them when we described how to create an image. We need to pay particular attention to the attachment store and load op fields. These determine what happens to the contents of the image at the beginning and at the end of the render pass.



Render Pass

```
VkRenderPass renderPass;

vkCreateRenderPass(device, pCreateInfo, nullptr,
&renderPass);
```

Begin Render Pass 0	Clear	Clear	Clear
End Render Pass 0	Store	Store	Store
Begin Render Pass 1	Load		
End Render Pass 1	Store		

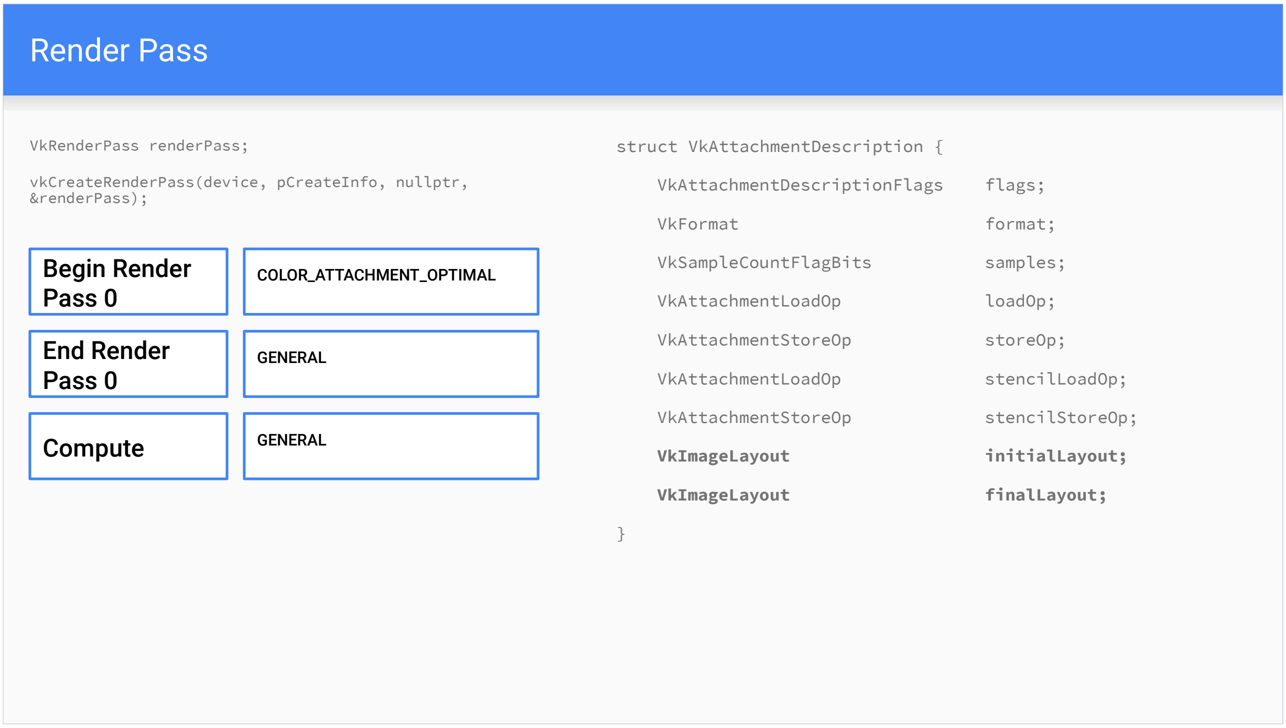
```
struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags    flags;
    VkFormat                        format;
    VkSampleCountFlagBits           samples;
    VkAttachmentLoadOp              loadOp;
    VkAttachmentStoreOp             storeOp;
    VkAttachmentLoadOp              stencilLoadOp;
    VkAttachmentStoreOp             stencilStoreOp;
    VkImageLayout                   initialLayout;
    VkImageLayout                   finalLayout;
}
```

Suppose we are doing a GBuffer pass. At the beginning of the render pass, we probably want to clear the image to a default color, so we are going to use the VK_ATTACHMENT_LOAD_OP_CLEAR op. At the end of the render pass we want to store our image data so that it can be used, for example, during our lighting pass. In that case we use VK_ATTACHMENT_STORE_OP_STORE op.

In the next render pass we want to add other elements to our image (i.e. transparent objects). In this case we want to load the existing content of the image using the VK_ATTACHMENT_LOAD_OP_LOAD op. We still want to store the results at the end of this render pass, so we use the same op as before.

If you know you are going to touch all the fragments of an image, you could use the VK_ATTACHMENT_LOAD_OP_DONT_CARE op at the beginning of the first render pass. Depending on the HW you are running on, a CLEAR might perform better.

It’s important to get these operations right as otherwise you will get the wrong results.



Render Pass

```
VkRenderPass renderPass;

vkCreateRenderPass(device, pCreateInfo, nullptr,
&renderPass);
```

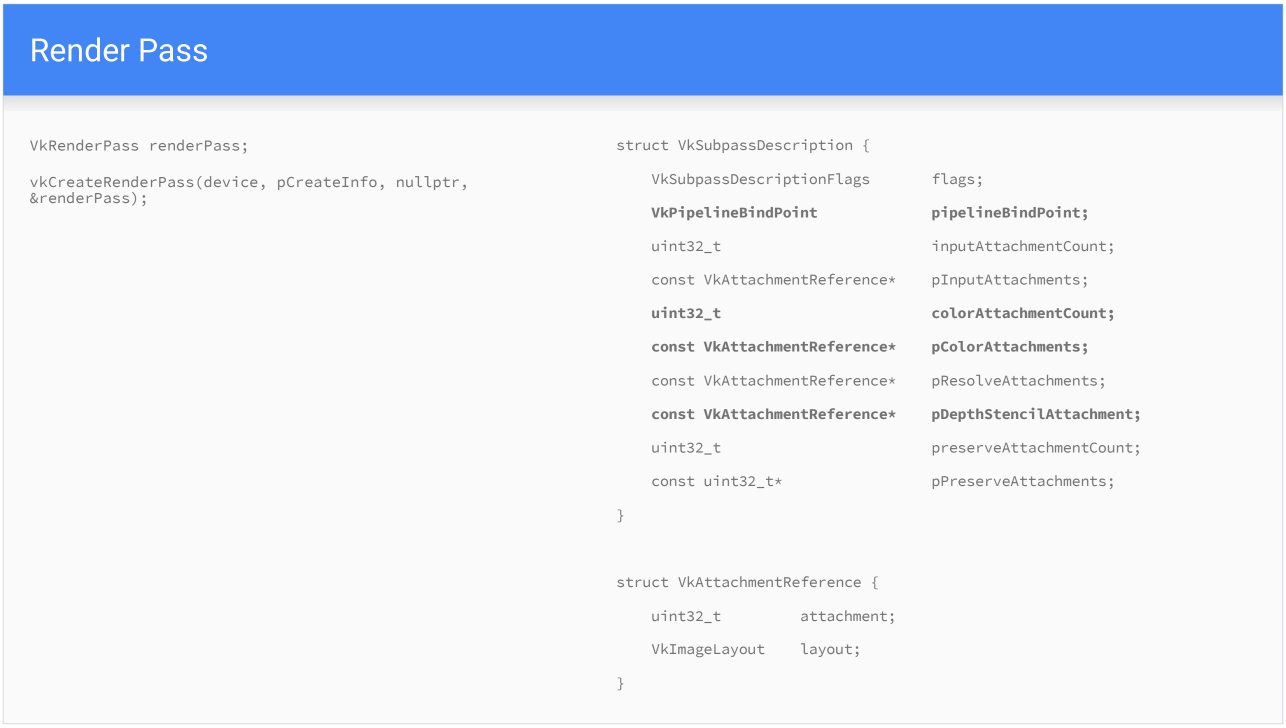
Begin Render Pass 0	COLOR_ATTACHMENT_OPTIMAL
End Render Pass 0	GENERAL
Compute	GENERAL

```
struct VkAttachmentDescription {
    VkAttachmentDescriptionFlags    flags;
    VkFormat                        format;
    VkSampleCountFlagBits           samples;
    VkAttachmentLoadOp              loadOp;
    VkAttachmentStoreOp             storeOp;
    VkAttachmentLoadOp              stencilLoadOp;
    VkAttachmentStoreOp             stencilStoreOp;
    VkImageLayout                   initialLayout;
    VkImageLayout                   finalLayout;
}
```

Next are the initial and final layout. These define the layout of the attachment at the beginning of the render pass and at the end.

The programmer is responsible to make sure the attachment is in the correct initial layout - the validation layers will kindly let you know if you got it wrong :)

On the other hand, you want to set the final layout so that the image is ready for the next rendering step. This way you can avoid having to transition the image manually.



Render Pass

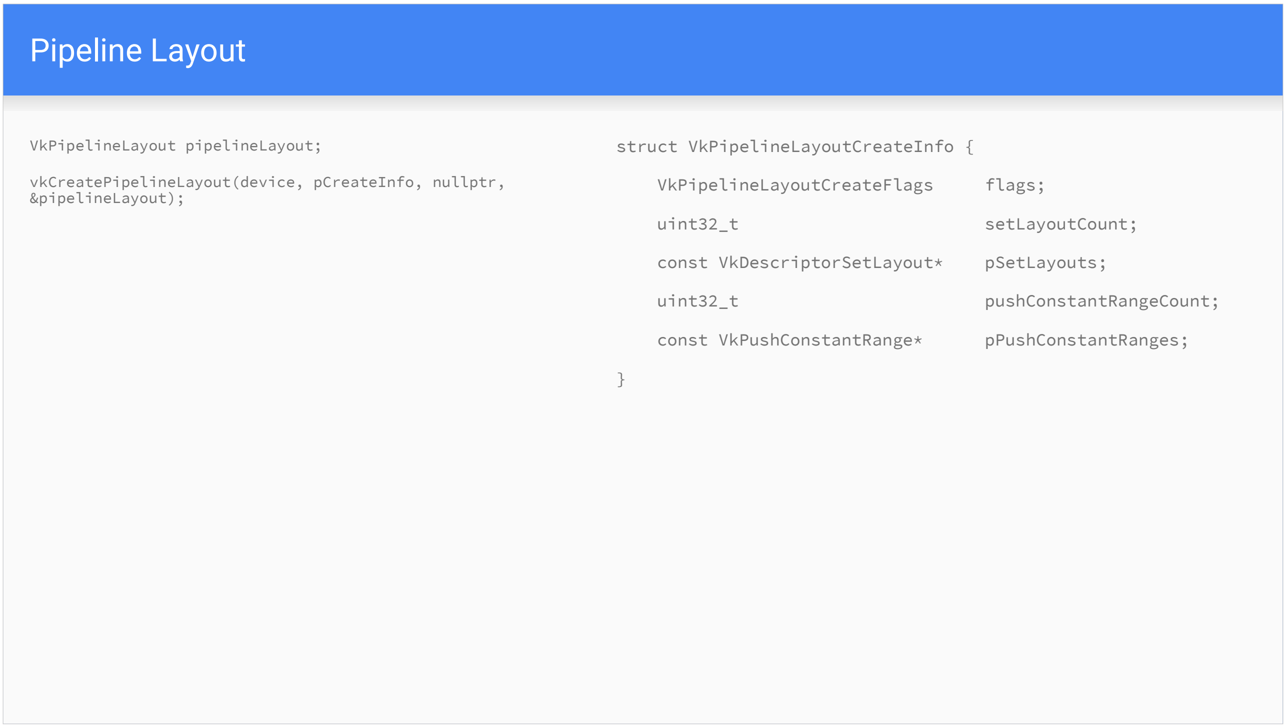
```
VkRenderPass renderPass;

vkCreateRenderPass(device, pCreateInfo, nullptr,
&renderPass);
```

```
struct VkSubpassDescription {
    VkSubpassDescriptionFlags       flags;
    VkPipelineBindPoint             pipelineBindPoint;
    uint32_t                        inputAttachmentCount;
    const VkAttachmentReference*    pInputAttachments;
    uint32_t                        colorAttachmentCount;
    const VkAttachmentReference*    pColorAttachments;
    const VkAttachmentReference*    pResolveAttachments;
    const VkAttachmentReference*    pDepthStencilAttachment;
    uint32_t                        preserveAttachmentCount;
    const uint32_t*                 pPreserveAttachments;
}

struct VkAttachmentReference {
    uint32_t        attachment;
    VkImageLayout   layout;
}
```

Even if we are not using subpasses, we still need to specify at least one subpass. The fields in bold are the ones you will care about most of the time. We need to populate the array with the index of the attachment from pAttachments in the VkRenderPassCreateInfo structure. The layout is the same as the initial layout.

This concludes the render pass creation. We covered traditional render passes as they are still used in many renderers. If you are starting a new project, we recommend using the dynamic rendering extension instead. With dynamic rendering you don’t have to worry about creating a render pass object and framebuffer.

The next element we need is the pipeline layout.



Pipeline Layout

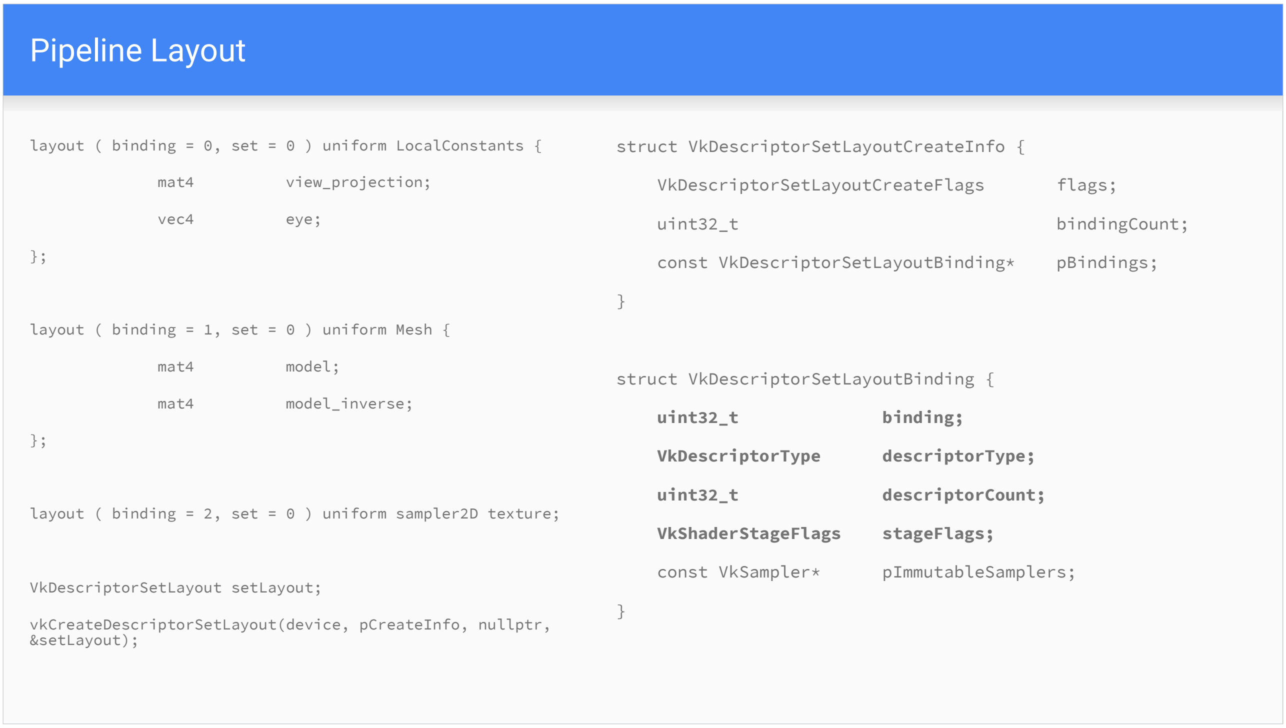
```
VkPipelineLayout pipelineLayout;

vkCreatePipelineLayout(device, pCreateInfo, nullptr,
&pipelineLayout);
```

```
struct VkPipelineLayoutCreateInfo {
    VkPipelineLayoutCreateFlags     flags;
    uint32_t                        setLayoutCount;
    const VkDescriptorSetLayout*    pSetLayouts;
    uint32_t                        pushConstantRangeCount;
    const VkPushConstantRange*      pPushConstantRanges;
}
```

A pipeline layout describes which resources are going to be used at render time. As with the render pass, this is just a description, no actual resources are bound at this point. Resource binding will happen during rendering and we'll cover it later.

As you can see here, we can use more than one set. The API allows for multiple sets to, once again, reduce the number of state changes. This is a rough guideline on how to use sets:

- 0: frame data
- 1: material data
- 2: per-object data

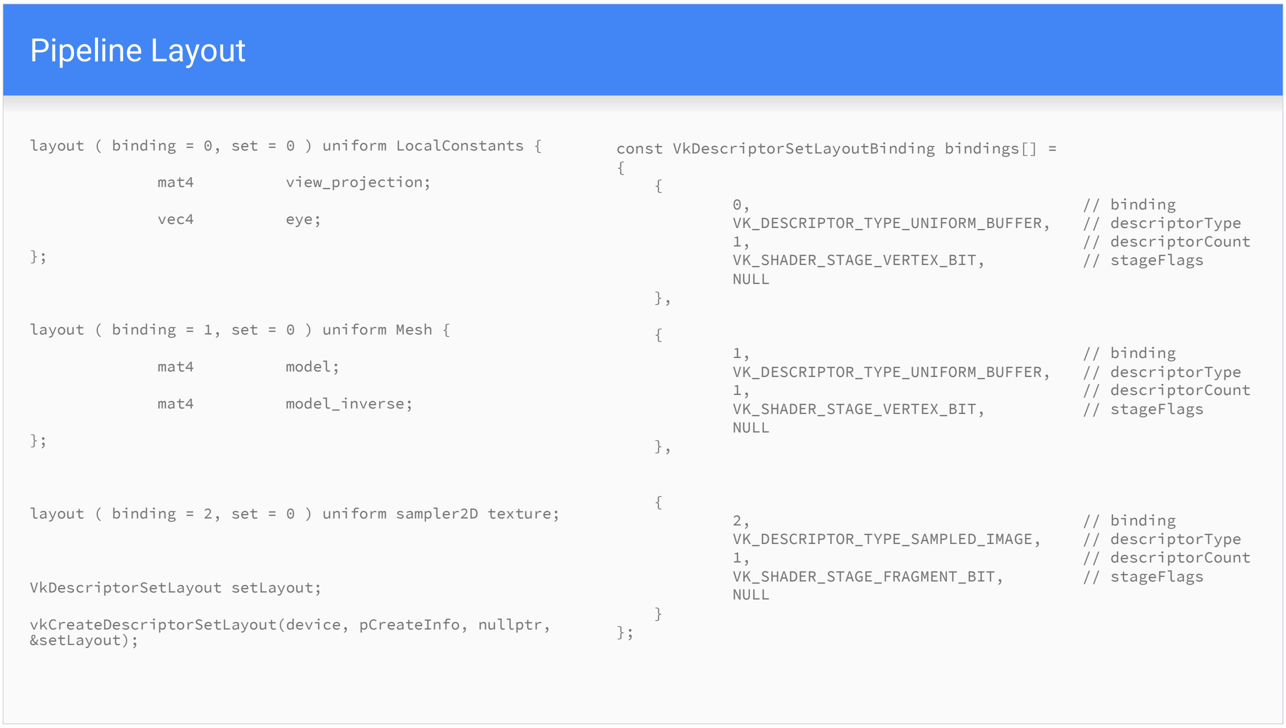


Pipeline Layout

```
layout ( binding = 0, set = 0 ) uniform LocalConstants {
              mat4          view_projection;
              vec4          eye;
};

layout ( binding = 1, set = 0 ) uniform Mesh {
              mat4          model;
              mat4          model_inverse;
};

layout ( binding = 2, set = 0 ) uniform sampler2D texture;

VkDescriptorSetLayout setLayout;
vkCreateDescriptorSetLayout(device, pCreateInfo, nullptr,
&setLayout);
```

```
struct VkDescriptorSetLayoutCreateInfo {
    VkDescriptorSetLayoutCreateFlags       flags;
    uint32_t                               bindingCount;
    const VkDescriptorSetLayoutBinding*    pBindings;
}

struct VkDescriptorSetLayoutBinding {
    uint32_t              binding;
    VkDescriptorType      descriptorType;
    uint32_t              descriptorCount;
    VkShaderStageFlags    stageFlags;
    const VkSampler*      pImmutableSamplers;
}
```

Let’s look at an example to make this more concrete. Here we have 3 bindings (0, 1, 2). We explicitly define the set, if omitted the resource will default to set 0. The highlighted fields are the one we are going to use. Immutable samplers are a useful feature if you know you have a fixed set of samplers you are going to use.

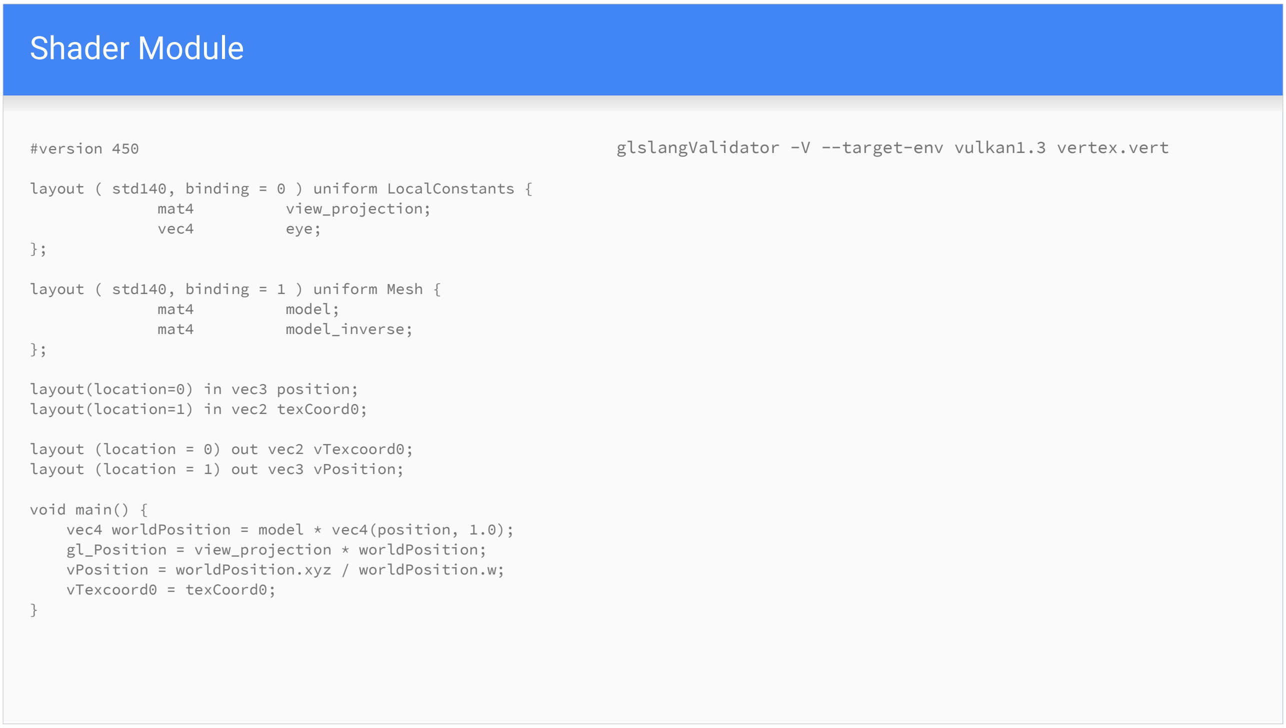


Pipeline Layout

```
layout ( binding = 0, set = 0 ) uniform LocalConstants {

              mat4          view_projection;

              vec4          eye;

};


layout ( binding = 1, set = 0 ) uniform Mesh {

              mat4          model;

              mat4          model_inverse;

};


layout ( binding = 2, set = 0 ) uniform sampler2D texture;


VkDescriptorSetLayout setLayout;

vkCreateDescriptorSetLayout(device, pCreateInfo, nullptr,
&setLayout);
```

```
const VkDescriptorSetLayoutBinding bindings[] =
{
    {
            0,                                  // binding
            VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,  // descriptorType
            1,                                  // descriptorCount
            VK_SHADER_STAGE_VERTEX_BIT,         // stageFlags
            NULL
    },

    {
            1,                                  // binding
            VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,  // descriptorType
            1,                                  // descriptorCount
            VK_SHADER_STAGE_VERTEX_BIT,         // stageFlags
            NULL
    },


    {
            2,                                  // binding
            VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE,   // descriptorType
            1,                                  // descriptorCount
            VK_SHADER_STAGE_FRAGMENT_BIT,       // stageFlags
            NULL
    }
};
```

Here we provide an example of how the bindings array is going to be defined for set 0. If we were using more than one set, we would have a separate bindings array for each set.



Shader Module

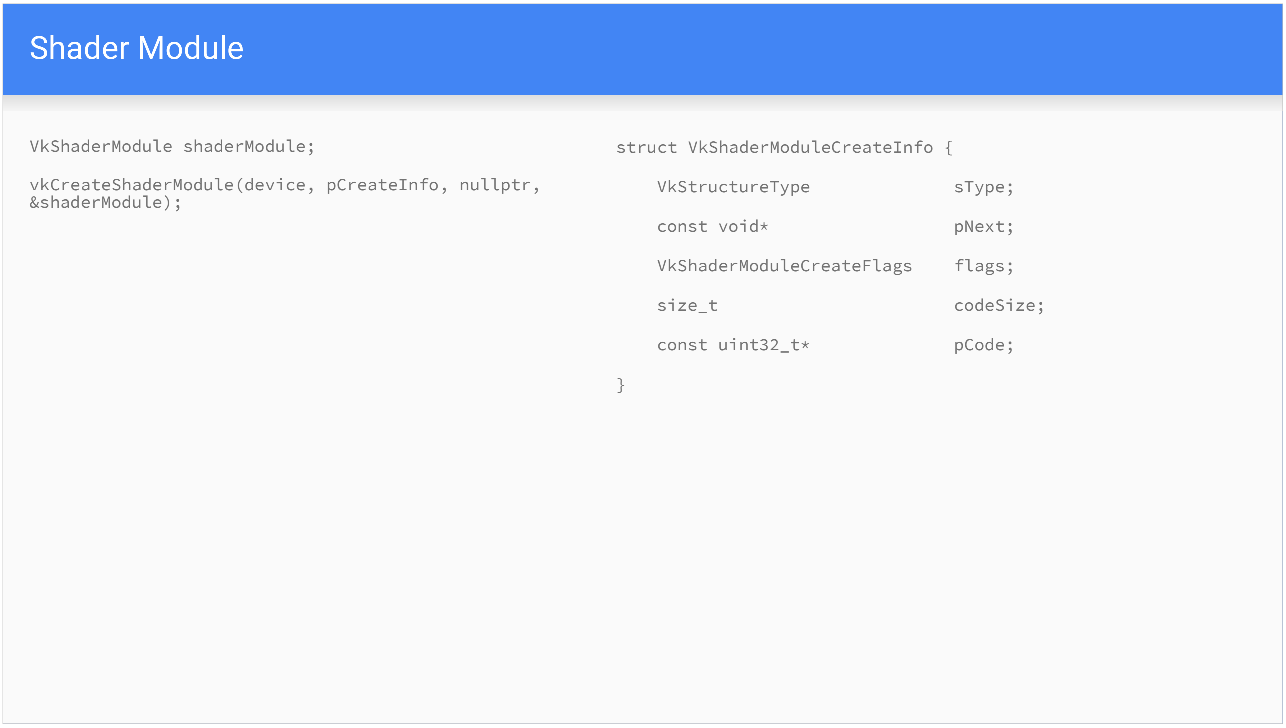
```
#version 450

layout ( std140, binding = 0 ) uniform LocalConstants {
              mat4          view_projection;
              vec4          eye;
};

layout ( std140, binding = 1 ) uniform Mesh {
              mat4          model;
              mat4          model_inverse;
};

layout(location=0) in vec3 position;
layout(location=1) in vec2 texCoord0;

layout (location = 0) out vec2 vTexcoord0;
layout (location = 1) out vec3 vPosition;

void main() {
    vec4 worldPosition = model * vec4(position, 1.0);
    gl_Position = view_projection * worldPosition;
    vPosition = worldPosition.xyz / worldPosition.w;
    vTexcoord0 = texCoord0;
}
```

```
glslangValidator -V --target-env vulkan1.3 vertex.vert
```

We are now ready to create our first program! As you have probably realize by now, Vulkan is quite verbose, and creating a pipeline requires a few steps. We are going to show how to create a graphics pipeline. Creating a compute pipeline follows a similar pattern but it’s a lot simpler as we only have one stage.

Vulkan can use GLSL code directly, however the recommended approach is to pre-compile shaders to SPIR-V, a binary format that can then be consumed by driver. This will save some time at compilation time and it also allows you to parse the SPIR-V code to automatically generate pipeline layouts and to map buffer structures to CPU code (we won’t cover this here, but we’ll provide links in the references).

Here we show a simple vertex shader which we compile to SPIR-V using the compiler provided by the Vulkan SDK. This will produce a .spv file that we can then use to create a pipeline.

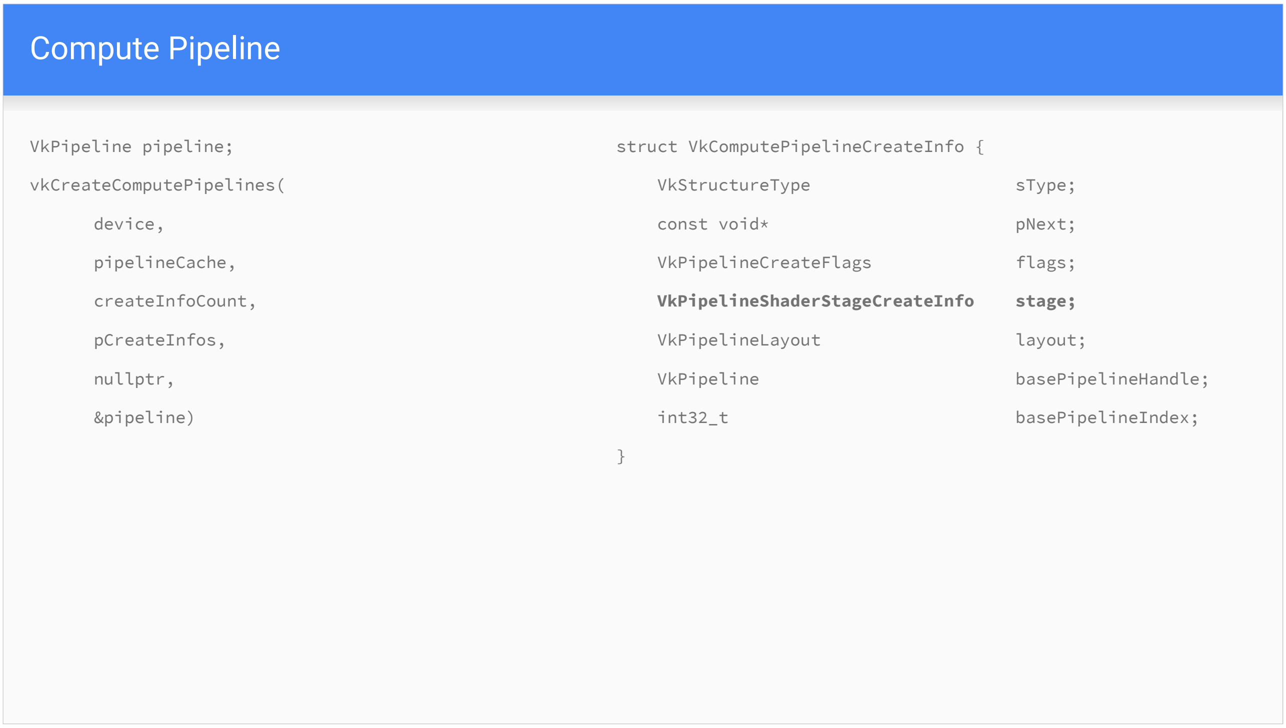


Shader Module

```
VkShaderModule shaderModule;

vkCreateShaderModule(device, pCreateInfo, nullptr,
&shaderModule);
```

```
struct VkShaderModuleCreateInfo {

    VkStructureType              sType;

    const void*                  pNext;

    VkShaderModuleCreateFlags    flags;

    size_t                       codeSize;

    const uint32_t*              pCode;

}
```

Now that we have our shader binary, we proceed to create a shader module. We simply read the file we created in the previous step and pass it to the API. We have to create a module for each active stage in the pipeline (vertex, fragment, etc.)

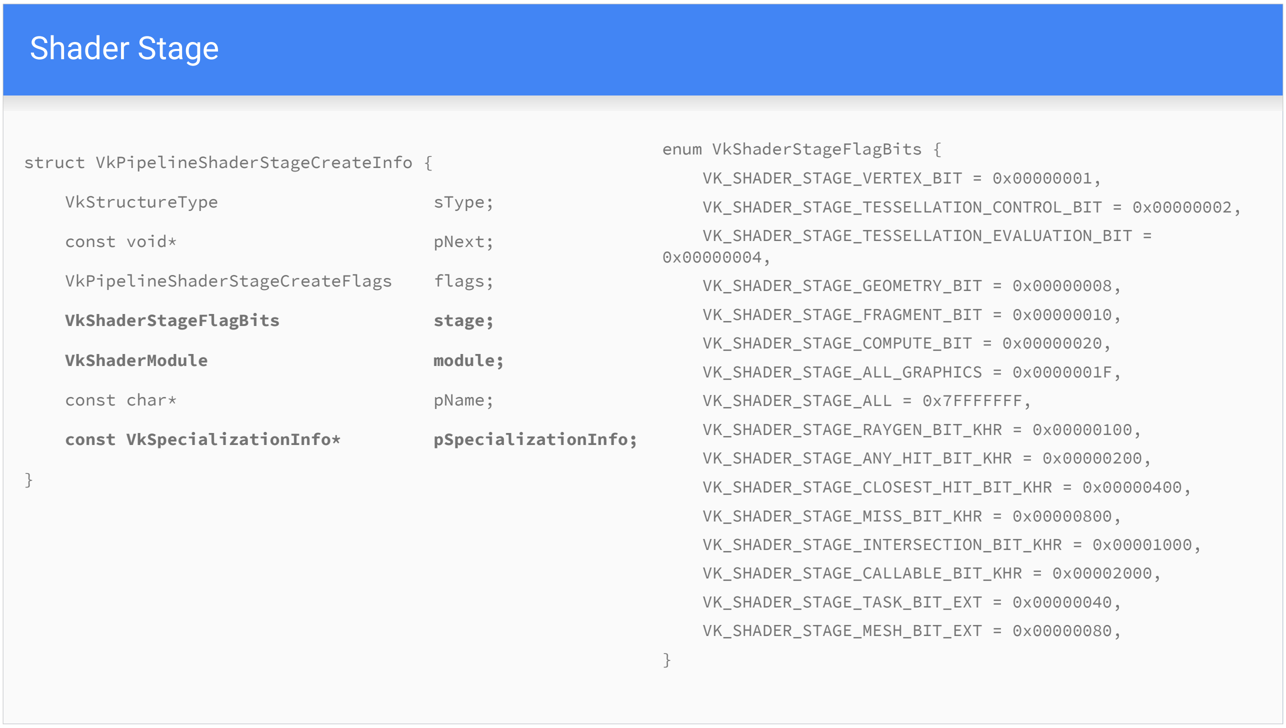


Compute Pipeline

```
VkPipeline pipeline;
vkCreateComputePipelines(
       device,
       pipelineCache,
       createInfoCount,
       pCreateInfos,
       nullptr,
       &pipeline)
```

```
struct VkComputePipelineCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineCreateFlags               flags;
    VkPipelineShaderStageCreateInfo     stage;
    VkPipelineLayout                    layout;
    VkPipeline                          basePipelineHandle;
    int32_t                             basePipelineIndex;
}
```

The easiest pipeline to create is the compute one, as it needs less parameters than the graphics one. In the next slide we will concentrate on the PipelineShaderStageCreateInfo, that is also shared with the graphics pipeline. In the case of a compute pipeline only one shader is used, thus only one stage is needed.



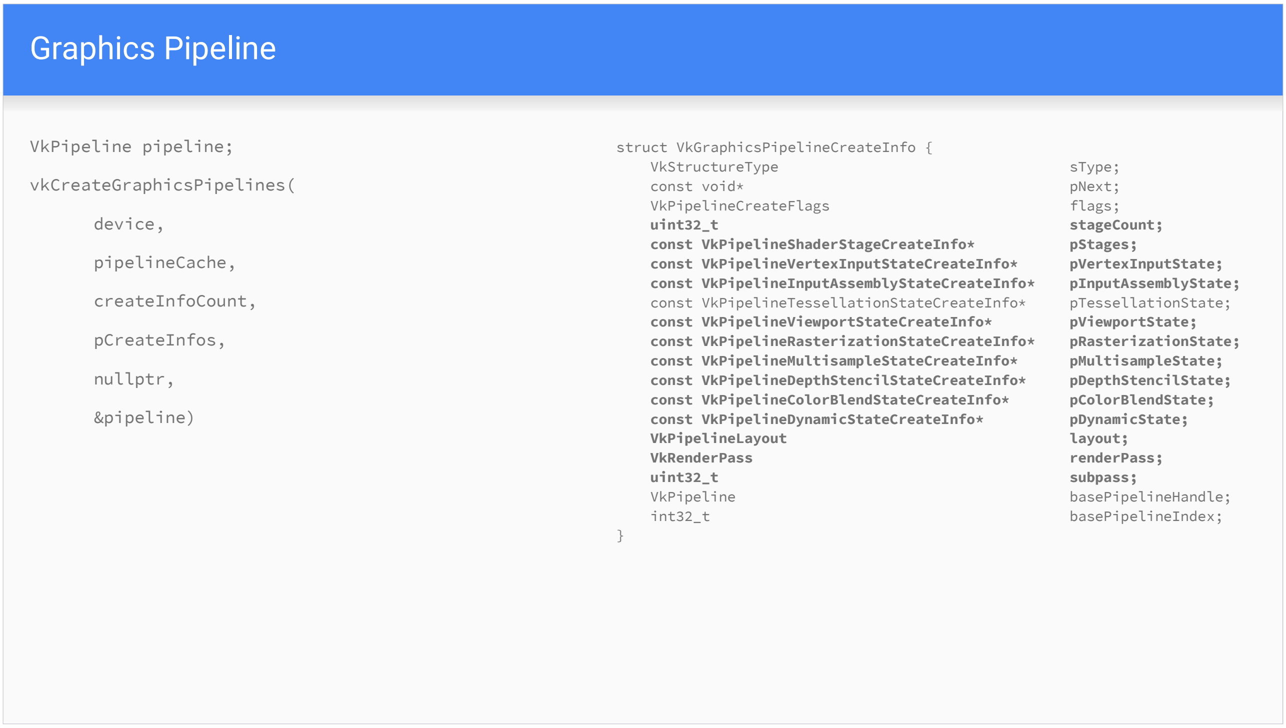
Shader Stage

```
struct VkPipelineShaderStageCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineShaderStageCreateFlags    flags;
    VkShaderStageFlagBits               stage;
    VkShaderModule                      module;
    const char*                         pName;
    const VkSpecializationInfo*         pSpecializationInfo;
}
```

```
enum VkShaderStageFlagBits {
    VK_SHADER_STAGE_VERTEX_BIT = 0x00000001,
    VK_SHADER_STAGE_TESSELLATION_CONTROL_BIT = 0x00000002,
    VK_SHADER_STAGE_TESSELLATION_EVALUATION_BIT =
0x00000004,
    VK_SHADER_STAGE_GEOMETRY_BIT = 0x00000008,
    VK_SHADER_STAGE_FRAGMENT_BIT = 0x00000010,
    VK_SHADER_STAGE_COMPUTE_BIT = 0x00000020,
    VK_SHADER_STAGE_ALL_GRAPHICS = 0x0000001F,
    VK_SHADER_STAGE_ALL = 0x7FFFFFFF,
    VK_SHADER_STAGE_RAYGEN_BIT_KHR = 0x00000100,
    VK_SHADER_STAGE_ANY_HIT_BIT_KHR = 0x00000200,
    VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR = 0x00000400,
    VK_SHADER_STAGE_MISS_BIT_KHR = 0x00000800,
    VK_SHADER_STAGE_INTERSECTION_BIT_KHR = 0x00001000,
    VK_SHADER_STAGE_CALLABLE_BIT_KHR = 0x00002000,
    VK_SHADER_STAGE_TASK_BIT_EXT = 0x00000040,
    VK_SHADER_STAGE_MESH_BIT_EXT = 0x00000080,
}
```

For each shader stage (compute, vertex, fragment, mesh, task…) we need to fill one of these structures. We already created the ShaderModule, we then need just the ShaderStageFlagBits, specifying which stage is corresponding to this module and optionally SpecializationInfo. We show also some of the ShaderStageFlags that can be used.

Specialization constants allow you to define a variable in a shader whose value will be defined at compile time. It's a similar mechanism to C++ templates and it simplifies creating multiple variants of the same code without having to use pre-processor macros.



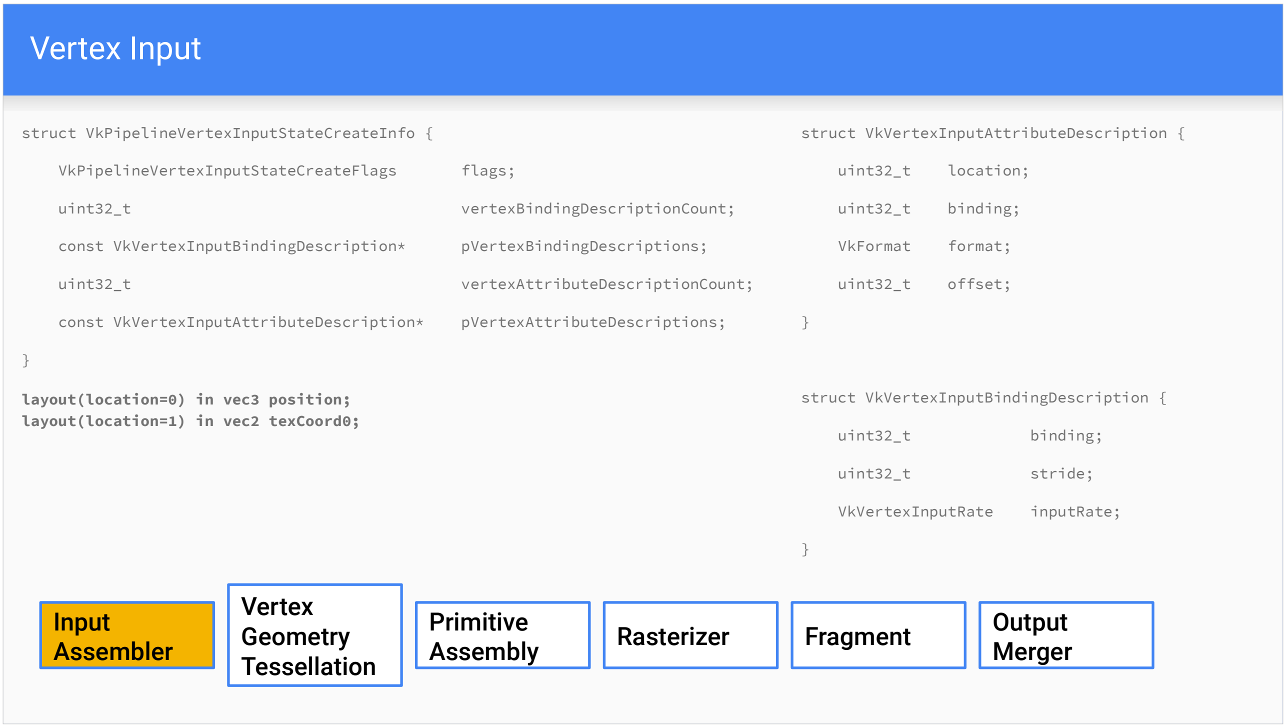
Graphics Pipeline

```
VkPipeline pipeline;

vkCreateGraphicsPipelines(

       device,

       pipelineCache,

       createInfoCount,

       pCreateInfos,

       nullptr,

       &pipeline)
```

```
struct VkGraphicsPipelineCreateInfo {
    VkStructureType                                  sType;
    const void*                                      pNext;
    VkPipelineCreateFlags                            flags;
    uint32_t                                         stageCount;
    const VkPipelineShaderStageCreateInfo*           pStages;
    const VkPipelineVertexInputStateCreateInfo*      pVertexInputState;
    const VkPipelineInputAssemblyStateCreateInfo*    pInputAssemblyState;
    const VkPipelineTessellationStateCreateInfo*     pTessellationState;
    const VkPipelineViewportStateCreateInfo*         pViewportState;
    const VkPipelineRasterizationStateCreateInfo*    pRasterizationState;
    const VkPipelineMultisampleStateCreateInfo*      pMultisampleState;
    const VkPipelineDepthStencilStateCreateInfo*     pDepthStencilState;
    const VkPipelineColorBlendStateCreateInfo*       pColorBlendState;
    const VkPipelineDynamicStateCreateInfo*          pDynamicState;
    VkPipelineLayout                                 layout;
    VkRenderPass                                     renderPass;
    uint32_t                                         subpass;
    VkPipeline                                       basePipelineHandle;
    int32_t                                          basePipelineIndex;
}
```

For a graphics program we need more informations when creating a pipeline. Now that we have our render pass, pipeline layout and shader modules we can (finally!) create the actual pipeline. The API allows you to create multiple pipelines at once, but we assume only one pipeline in this example. Please don’t run away, we promise it’s not as scary as it looks.

We are going to cover each of the highlighted structures individually. We already saw the ShaderStageCreateInfo struct. The main difference from the compute pipeline is that we need to specify more than one stage for a graphics pipeline.



Vertex Input

```
struct VkPipelineVertexInputStateCreateInfo {
    VkPipelineVertexInputStateCreateFlags       flags;
    uint32_t                                    vertexBindingDescriptionCount;
    const VkVertexInputBindingDescription*      pVertexBindingDescriptions;
    uint32_t                                    vertexAttributeDescriptionCount;
    const VkVertexInputAttributeDescription*    pVertexAttributeDescriptions;
}
```

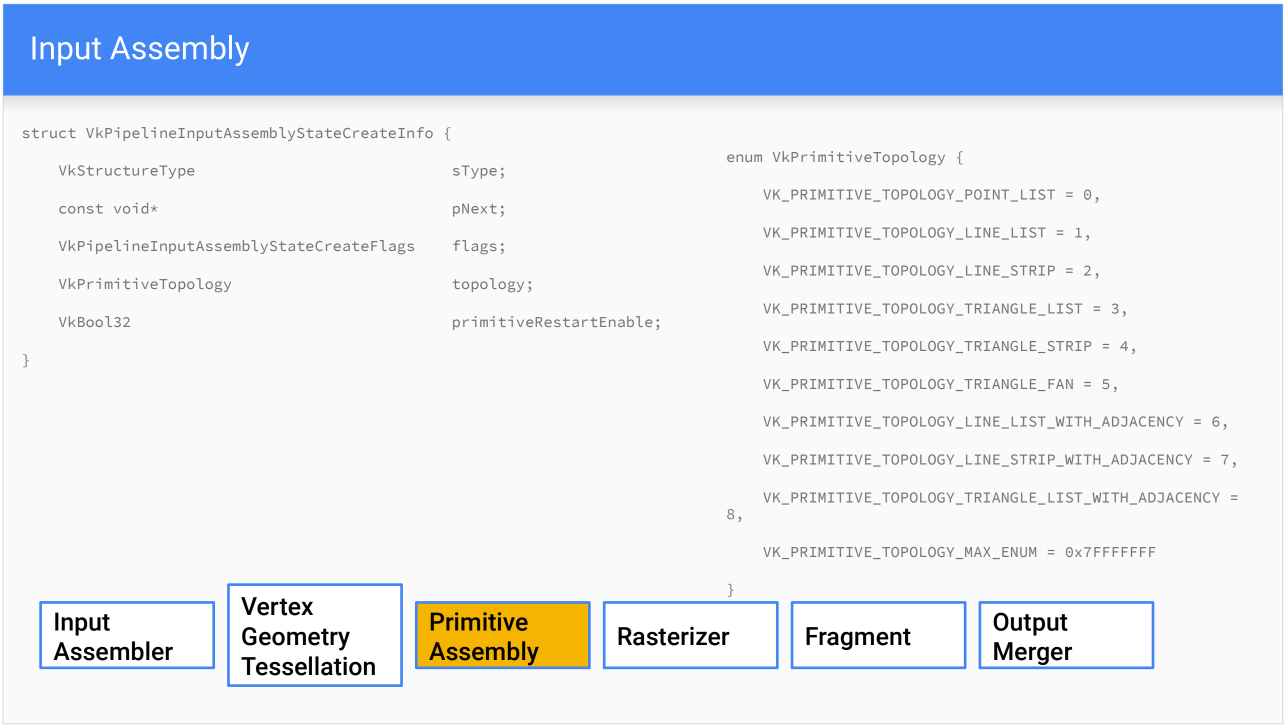
```
layout(location=0) in vec3 position;
layout(location=1) in vec2 texCoord0;
```

```
struct VkVertexInputAttributeDescription {
    uint32_t    location;
    uint32_t    binding;
    VkFormat    format;
    uint32_t    offset;
}
```

```
struct VkVertexInputBindingDescription {
    uint32_t             binding;
    uint32_t             stride;
    VkVertexInputRate    inputRate;
}
```

This struct defines how the vertex data is read into the vertex shader of the pipeline. VertexInputAttribute is each individual stream of vertex data (position, normals, UVs…) with the location (as specified in the shader), binding (specifying which vertex buffer is used to read from), format and offset.

VertexInputBinding specify each vertex buffer used and its InputRate, either vertex or instance, to use hardware instancing.

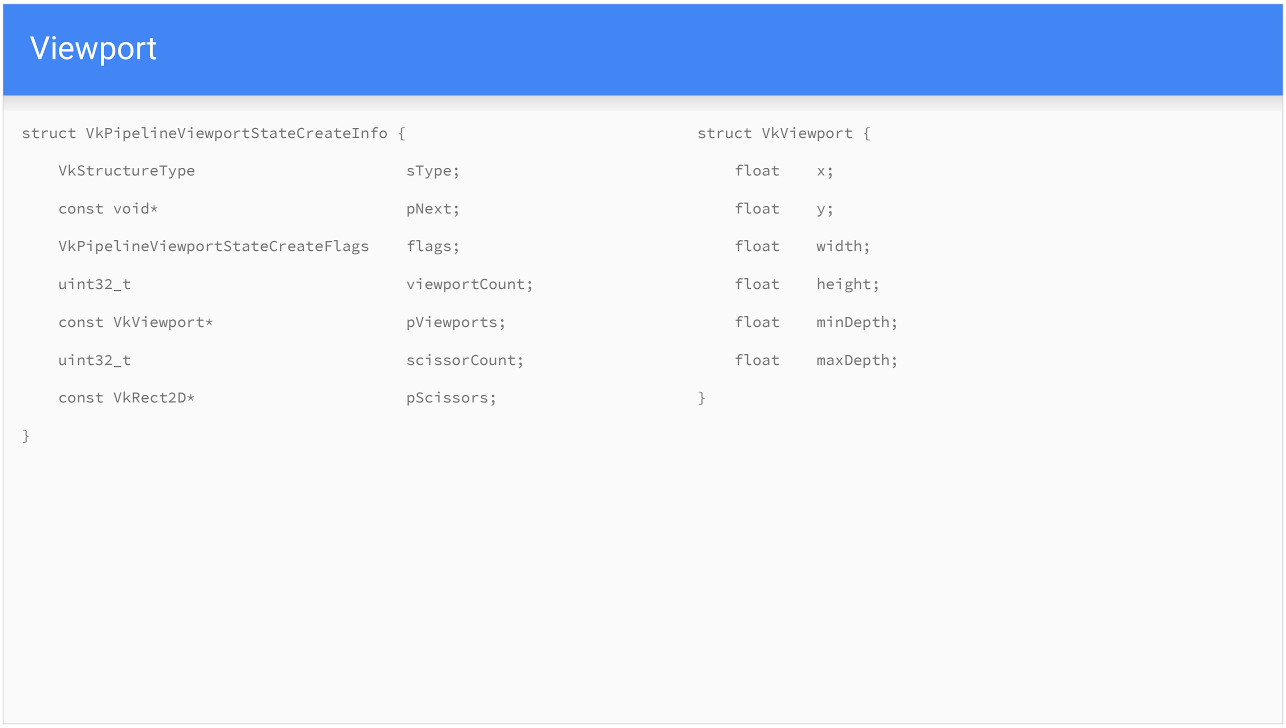


Input Assembly

```
struct VkPipelineInputAssemblyStateCreateInfo {
    VkStructureType                            sType;
    const void*                                pNext;
    VkPipelineInputAssemblyStateCreateFlags    flags;
    VkPrimitiveTopology                        topology;
    VkBool32                                   primitiveRestartEnable;
}
```

```
enum VkPrimitiveTopology {
    VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST = 1,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP = 2,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST = 3,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP = 4,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN = 5,
    VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY = 6,
    VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY = 7,
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY =
8,
    VK_PRIMITIVE_TOPOLOGY_MAX_ENUM = 0x7FFFFFFF
}
```

This struct is mainly used to define the topology of the vertex data: point, line list, triangle list, triangle strip and more.



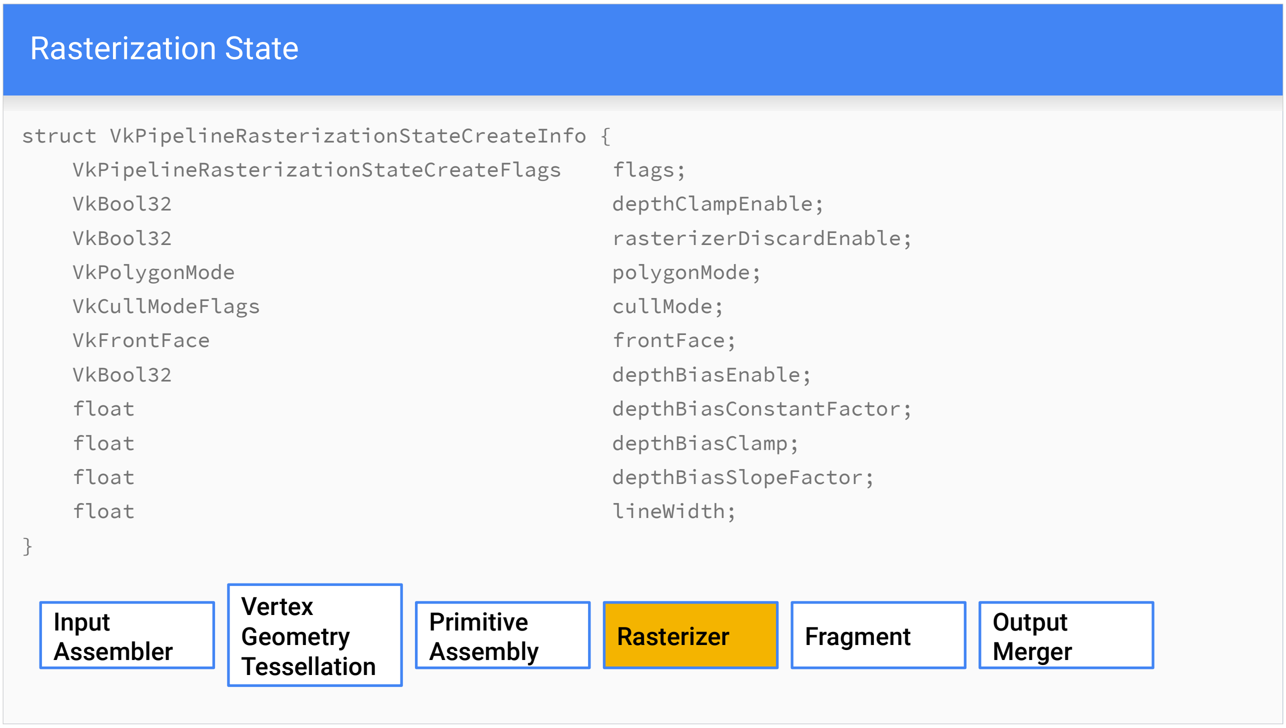
Viewport

```
struct VkPipelineViewportStateCreateInfo {
    VkStructureType                     sType;
    const void*                         pNext;
    VkPipelineViewportStateCreateFlags  flags;
    uint32_t                            viewportCount;
    const VkViewport*                   pViewports;
    uint32_t                            scissorCount;
    const VkRect2D*                     pScissors;
}
```

```
struct VkViewport {
    float    x;
    float    y;
    float    width;
    float    height;
    float    minDepth;
    float    maxDepth;
}
```

This is used to specify the different viewport and scissors that will be used by the pipeline. Unless the Multiviewport feature is used, viewports and scissors counts should be 1.

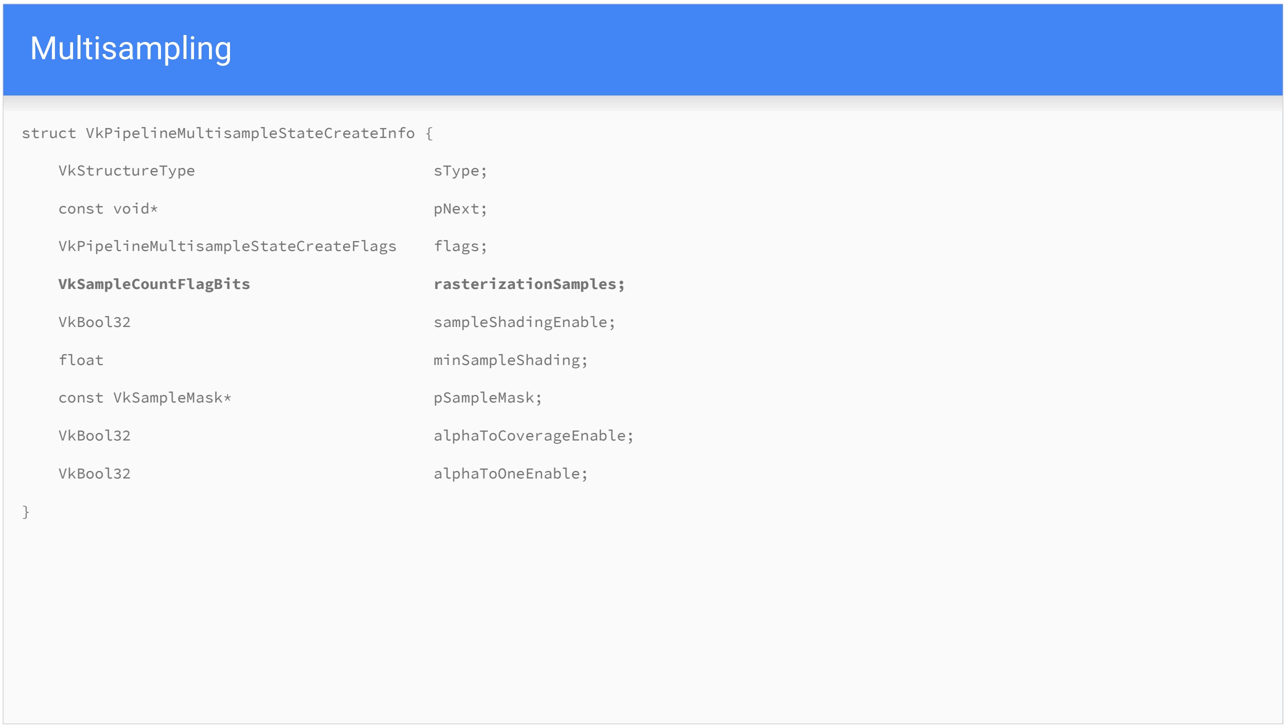
While we can specify the viewport as part of the pipeline, this is usually defined as a dynamic state that can be changed at runtime - image if you had to recreate all of your pipelines when the resolution changes!



Rasterization State

```
struct VkPipelineRasterizationStateCreateInfo {
    VkPipelineRasterizationStateCreateFlags    flags;
    VkBool32                                   depthClampEnable;
    VkBool32                                   rasterizerDiscardEnable;
    VkPolygonMode                              polygonMode;
    VkCullModeFlags                            cullMode;
    VkFrontFace                                frontFace;
    VkBool32                                   depthBiasEnable;
    float                                      depthBiasConstantFactor;
    float                                      depthBiasClamp;
    float                                      depthBiasSlopeFactor;
    float                                      lineWidth;
}
```

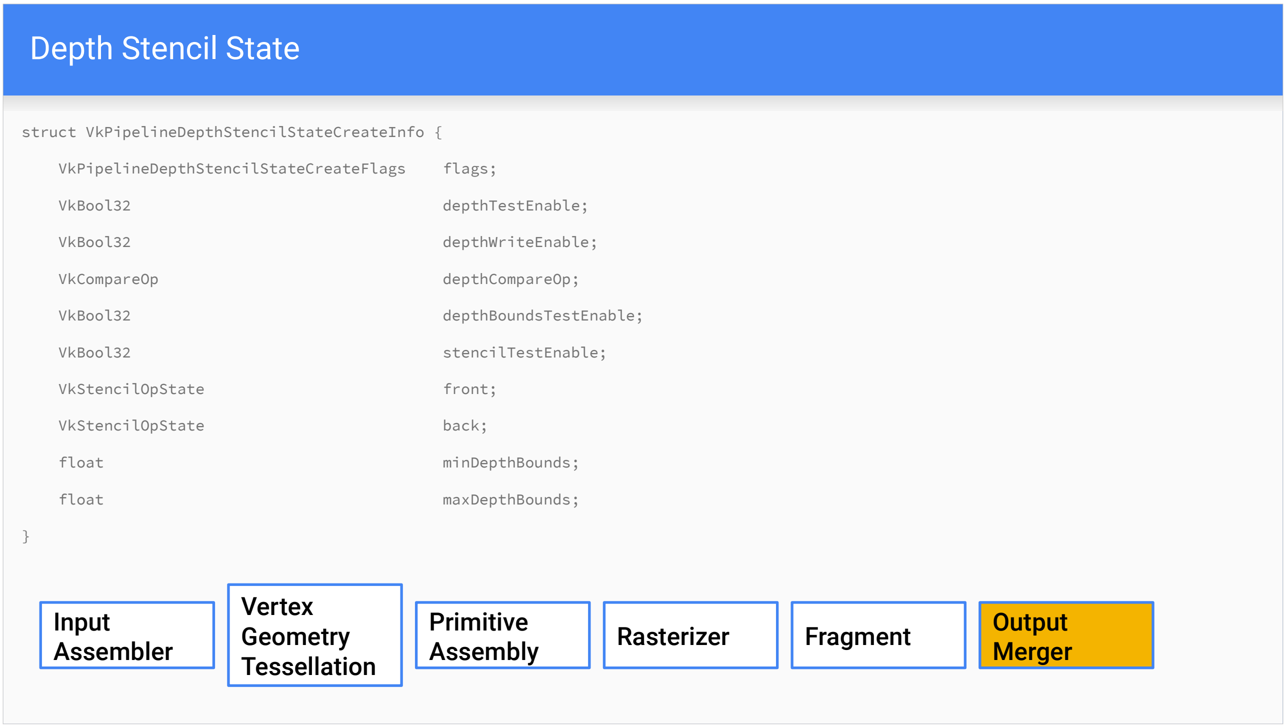
Rasterization state is used just before the fragment program, and let the user specify different things like the PolygonMode (point, fill, line), CullMode as the triangle facing direction, FrontFace as the front face used for culling.
There are also DepthBias controls used to manipulate depth values, used especially when rendering shadows.



Multisampling

```
struct VkPipelineMultisampleStateCreateInfo {
    VkStructureType                          sType;
    const void*                              pNext;
    VkPipelineMultisampleStateCreateFlags    flags;
    VkSampleCountFlagBits                    rasterizationSamples;
    VkBool32                                 sampleShadingEnable;
    float                                    minSampleShading;
    const VkSampleMask*                      pSampleMask;
    VkBool32                                 alphaToCoverageEnable;
    VkBool32                                 alphaToOneEnable;
}
```

The multisample state, as the name implies, is used to control how many samples are used during rasterization. The other fields are not needed most of the time.



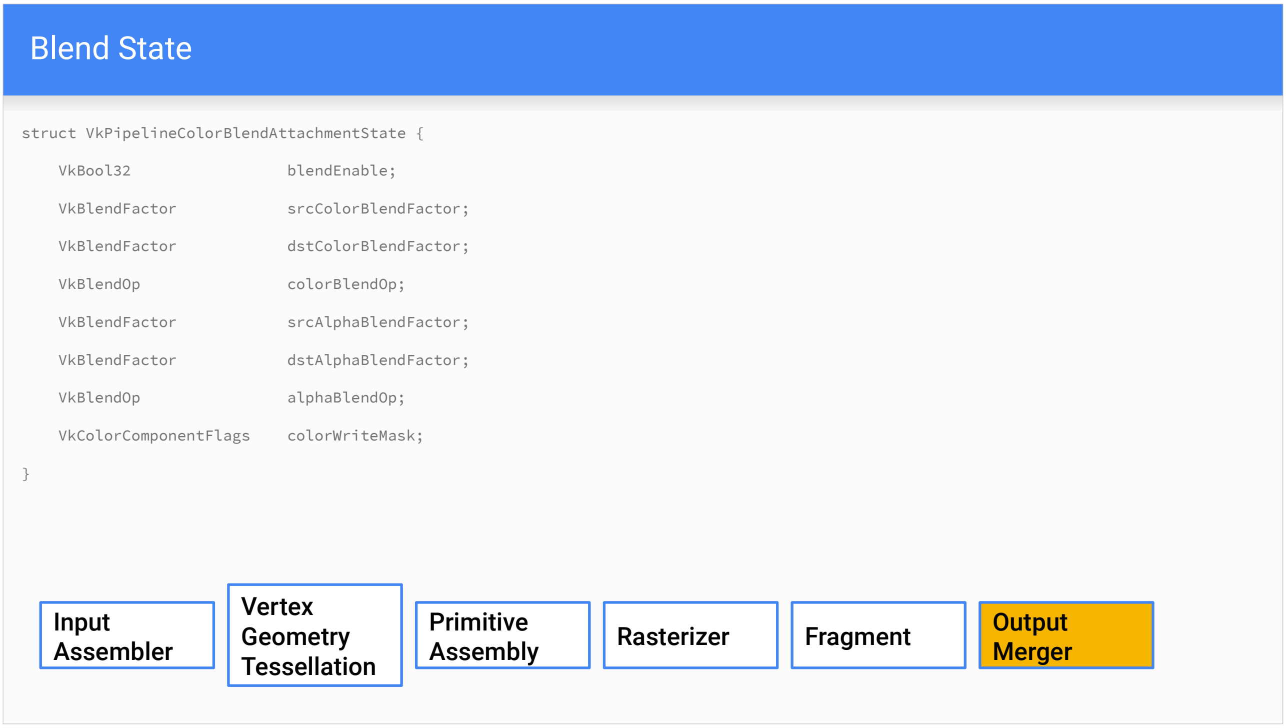
Depth Stencil State

```
struct VkPipelineDepthStencilStateCreateInfo {
    VkPipelineDepthStencilStateCreateFlags    flags;
    VkBool32                                  depthTestEnable;
    VkBool32                                  depthWriteEnable;
    VkCompareOp                               depthCompareOp;
    VkBool32                                  depthBoundsTestEnable;
    VkBool32                                  stencilTestEnable;
    VkStencilOpState                          front;
    VkStencilOpState                          back;
    float                                     minDepthBounds;
    float                                     maxDepthBounds;
}
```

Input Assembler | Vertex Geometry Tessellation | Primitive Assembly | Rasterizer | Fragment | Output Merger

This structure is used to control depth and stencil test and writing. If you have used a depth and/or stencil buffer before, this should all be familiar. The main difference with OpenGL is that we need to define these at pipeline creation time.

We have highlighted the Output Merger as the block affected by these settings. Depending on the setup of your pipeline and the behaviour of your fragment shader, depth writing might happen before the fragment shader runs. This is called early-z and it can improve performance if the fragment about to be shaded fails the depth test.



Blend State

```
struct VkPipelineColorBlendAttachmentState {
    VkBool32                 blendEnable;
    VkBlendFactor            srcColorBlendFactor;
    VkBlendFactor            dstColorBlendFactor;
    VkBlendOp                colorBlendOp;
    VkBlendFactor            srcAlphaBlendFactor;
    VkBlendFactor            dstAlphaBlendFactor;
    VkBlendOp                alphaBlendOp;
    VkColorComponentFlags    colorWriteMask;
}
```

Input Assembler

Vertex Geometry Tessellation

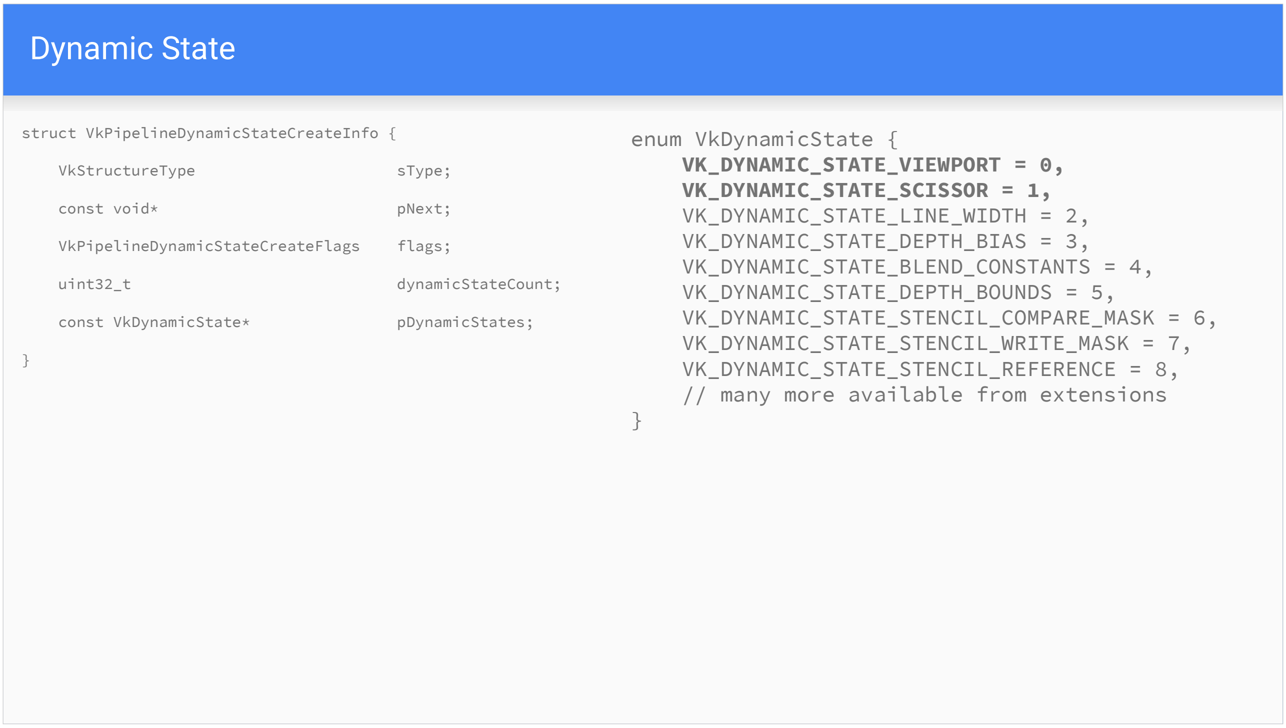
Primitive Assembly

Rasterizer

Fragment

Output Merger

This struct is used to specify alpha blending if needed. Blend factors and operations are used to compose the pipeline rendering into the specified framebuffer. There is also separation between color and alpha operations, as well as which color channel to write to.



Dynamic State

```
struct VkPipelineDynamicStateCreateInfo {

    VkStructureType                     sType;

    const void*                         pNext;

    VkPipelineDynamicStateCreateFlags   flags;

    uint32_t                            dynamicStateCount;

    const VkDynamicState*               pDynamicStates;

}
```

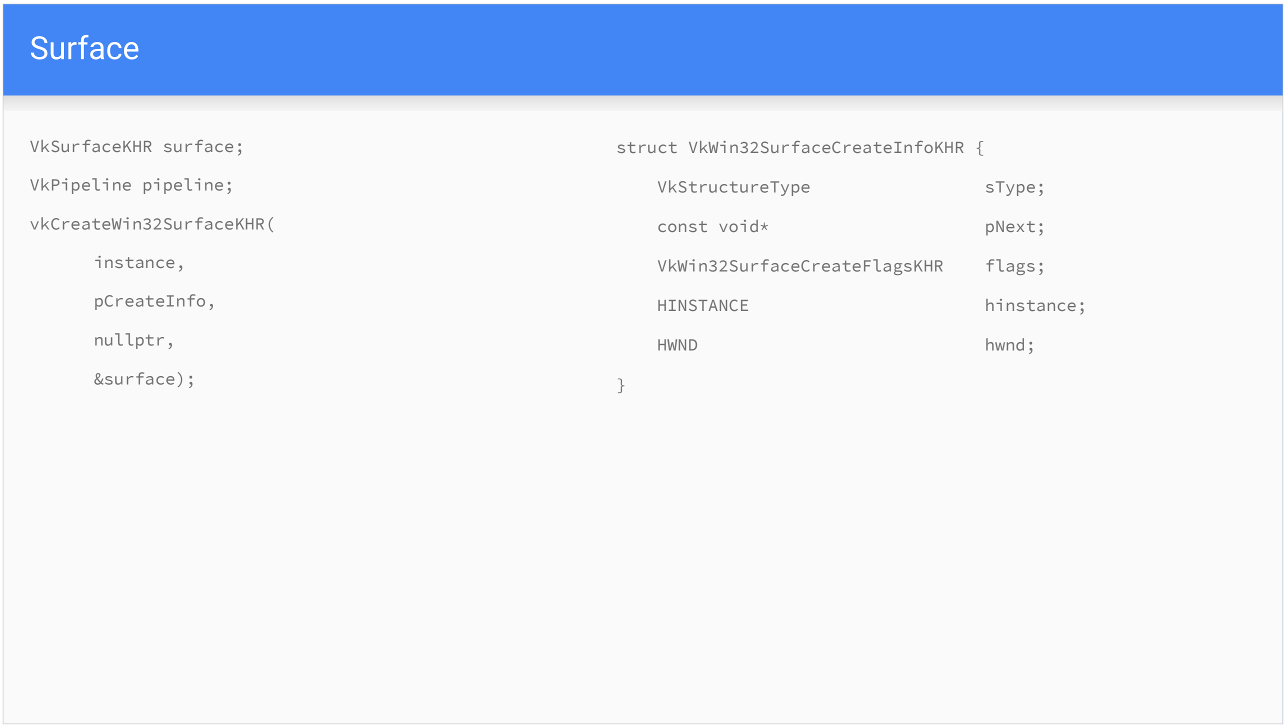
```
enum VkDynamicState {
    VK_DYNAMIC_STATE_VIEWPORT = 0,
    VK_DYNAMIC_STATE_SCISSOR = 1,
    VK_DYNAMIC_STATE_LINE_WIDTH = 2,
    VK_DYNAMIC_STATE_DEPTH_BIAS = 3,
    VK_DYNAMIC_STATE_BLEND_CONSTANTS = 4,
    VK_DYNAMIC_STATE_DEPTH_BOUNDS = 5,
    VK_DYNAMIC_STATE_STENCIL_COMPARE_MASK = 6,
    VK_DYNAMIC_STATE_STENCIL_WRITE_MASK = 7,
    VK_DYNAMIC_STATE_STENCIL_REFERENCE = 8,
    // many more available from extensions
}
```

The monolithic structure passed to create the pipeline can lead to a lot of pipelines duplication based on just different operations like depth testing, cull mode, or viewport/scissor changes.
To overcome this explosion of pipelines, dynamic states can be specified so that these can be specified when recording the command buffer using specific commands.
At the beginning only few states could be dynamic, like viewport, scissor and stencil values, but recently also cull modes, depth and stencil full states, as well as color blend, can be specified at runtime.

And with this we can finally create our graphics pipeline.

Break

Questions?



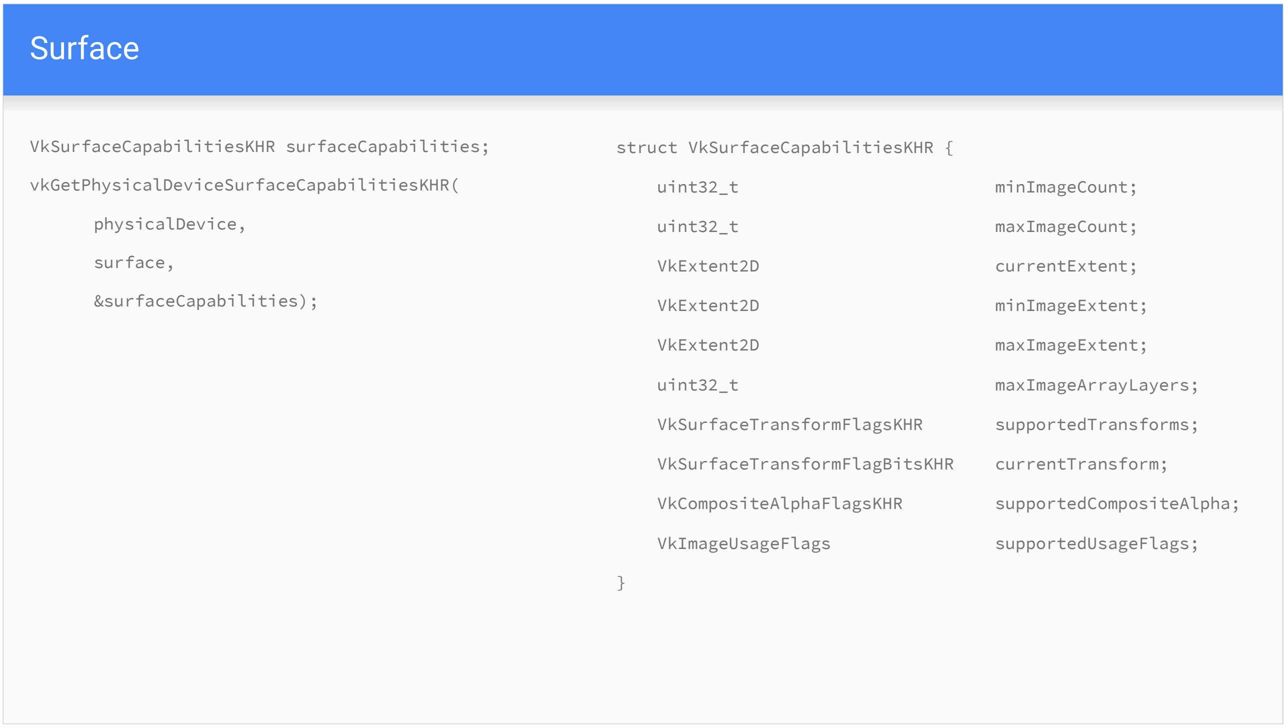
Surface

```
VkSurfaceKHR surface;

VkPipeline pipeline;

vkCreateWin32SurfaceKHR(

      instance,

      pCreateInfo,

      nullptr,

      &surface);
```

```
struct VkWin32SurfaceCreateInfoKHR {

    VkStructureType                 sType;

    const void*                     pNext;

    VkWin32SurfaceCreateFlagsKHR    flags;

    HINSTANCE                       hinstance;

    HWND                            hwnd;

}
```

Vulkan supports rendering offline, so you don’t need this unless you want to display something on screen. For this reason the surface and swapchain features are exposed through extensions. Remember to enable them if you need access to these features (VK_KHR_surface and VK_KHR_swapchain - make sure you also enable the related extension for the platform you are developing on, i.e. VK_KHR_win32_surface)

We are using Windows in this example and we are assuming you have already created a system window. Other platforms work in a similar fashion.

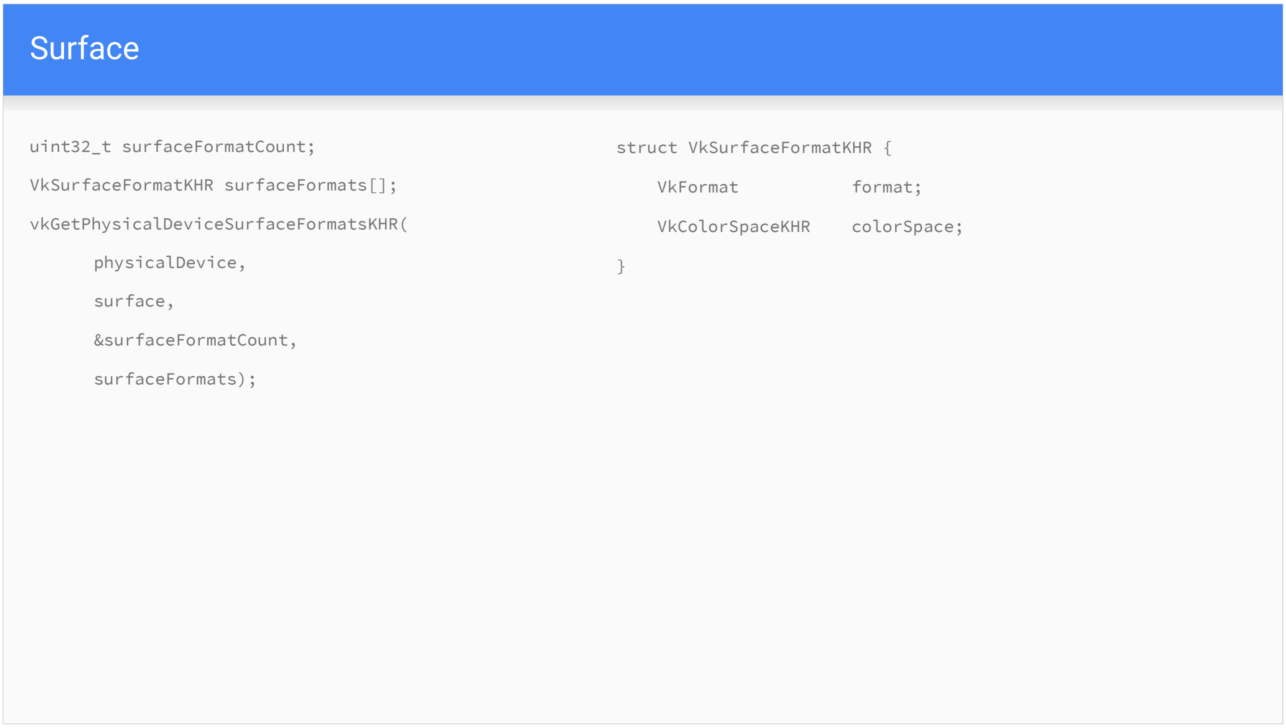


Surface

```
VkSurfaceCapabilitiesKHR surfaceCapabilities;
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(
      physicalDevice,
      surface,
      &surfaceCapabilities);
```

```
struct VkSurfaceCapabilitiesKHR {
    uint32_t                         minImageCount;
    uint32_t                         maxImageCount;
    VkExtent2D                       currentExtent;
    VkExtent2D                       minImageExtent;
    VkExtent2D                       maxImageExtent;
    uint32_t                         maxImageArrayLayers;
    VkSurfaceTransformFlagsKHR       supportedTransforms;
    VkSurfaceTransformFlagBitsKHR    currentTransform;
    VkCompositeAlphaFlagsKHR         supportedCompositeAlpha;
    VkImageUsageFlags                supportedUsageFlags;
}
```

Before we can create the swapchain, we need to query the surface we just created to determine its size, how many swapchain images it supports, etc.

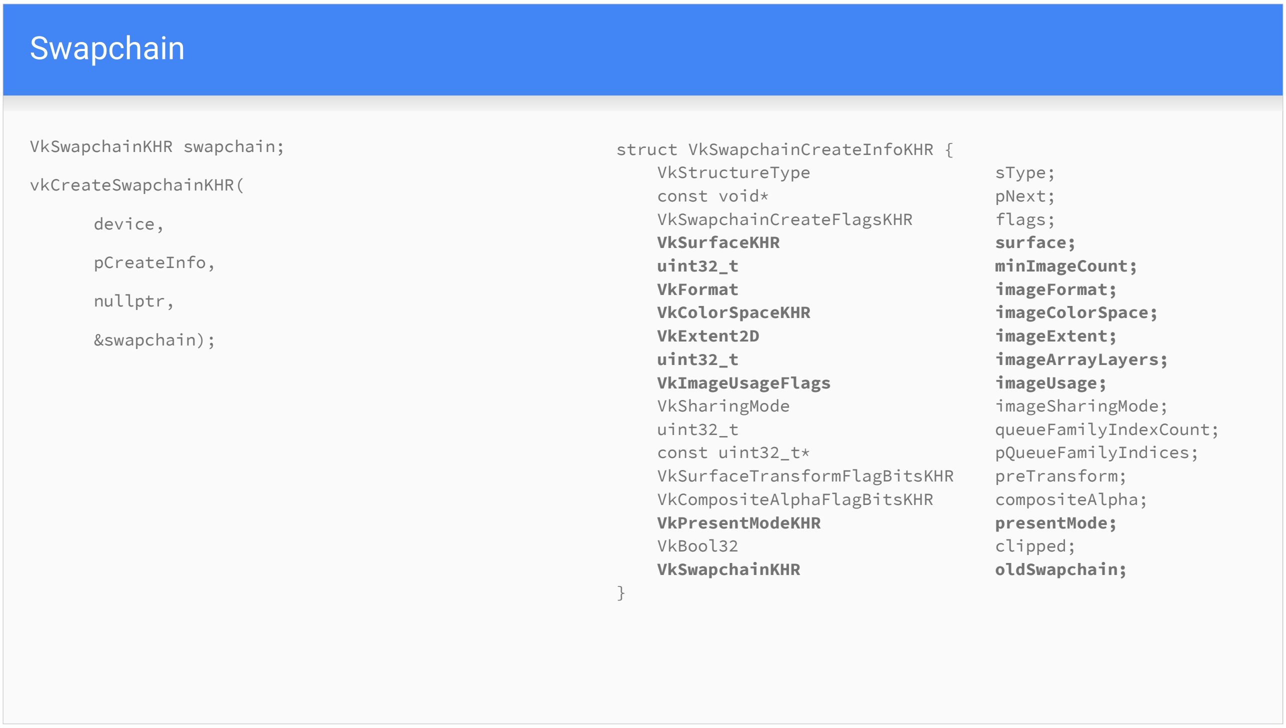


Surface

```
uint32_t surfaceFormatCount;
VkSurfaceFormatKHR surfaceFormats[];
vkGetPhysicalDeviceSurfaceFormatsKHR(
      physicalDevice,
      surface,
      &surfaceFormatCount,
      surfaceFormats);
```

```
struct VkSurfaceFormatKHR {
    VkFormat          format;
    VkColorSpaceKHR   colorSpace;
}
```

Next we have to query which formats and color spaces the surface supports. This is important to make sure we use a valid format to render into.



Swapchain

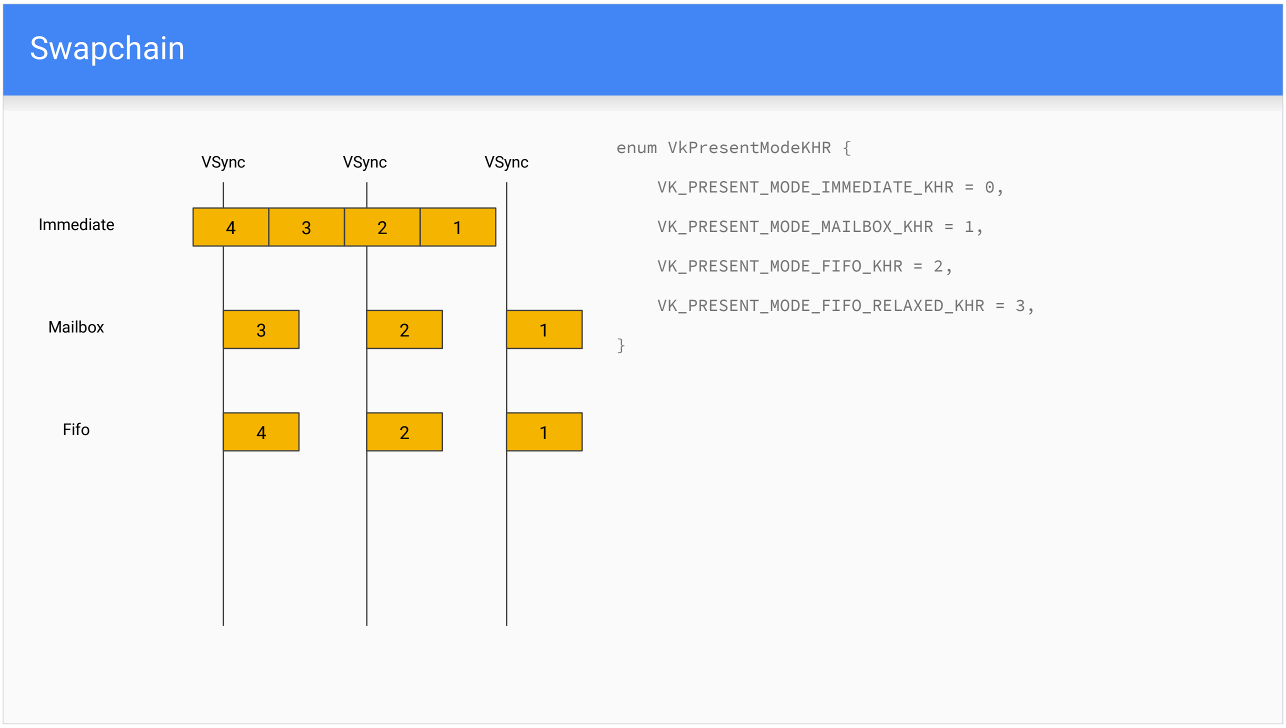
```
VkSwapchainKHR swapchain;

vkCreateSwapchainKHR(

      device,

      pCreateInfo,

      nullptr,

      &swapchain);
```

```
struct VkSwapchainCreateInfoKHR {
    VkStructureType                  sType;
    const void*                      pNext;
    VkSwapchainCreateFlagsKHR        flags;
    VkSurfaceKHR                     surface;
    uint32_t                         minImageCount;
    VkFormat                         imageFormat;
    VkColorSpaceKHR                  imageColorSpace;
    VkExtent2D                       imageExtent;
    uint32_t                         imageArrayLayers;
    VkImageUsageFlags                imageUsage;
    VkSharingMode                    imageSharingMode;
    uint32_t                         queueFamilyIndexCount;
    const uint32_t*                  pQueueFamilyIndices;
    VkSurfaceTransformFlagBitsKHR    preTransform;
    VkCompositeAlphaFlagBitsKHR      compositeAlpha;
    VkPresentModeKHR                 presentMode;
    VkBool32                         clipped;
    VkSwapchainKHR                   oldSwapchain;
}
```

Now we have all of the information we need to create a swapchain. We highlighted the fields that you are going to use most of the time. We are going to explain presentMode next.

When the resolution changes, we need to re-create the swapchain. We can pass the current swapchain to the API as the driver might be able to optimize the creation of the new swapchain. There’s no need to re-create the surface.

We describe what image count does in the next slide.



Swapchain

VSync VSync VSync

Immediate 4 3 2 1

Mailbox 3 2 1

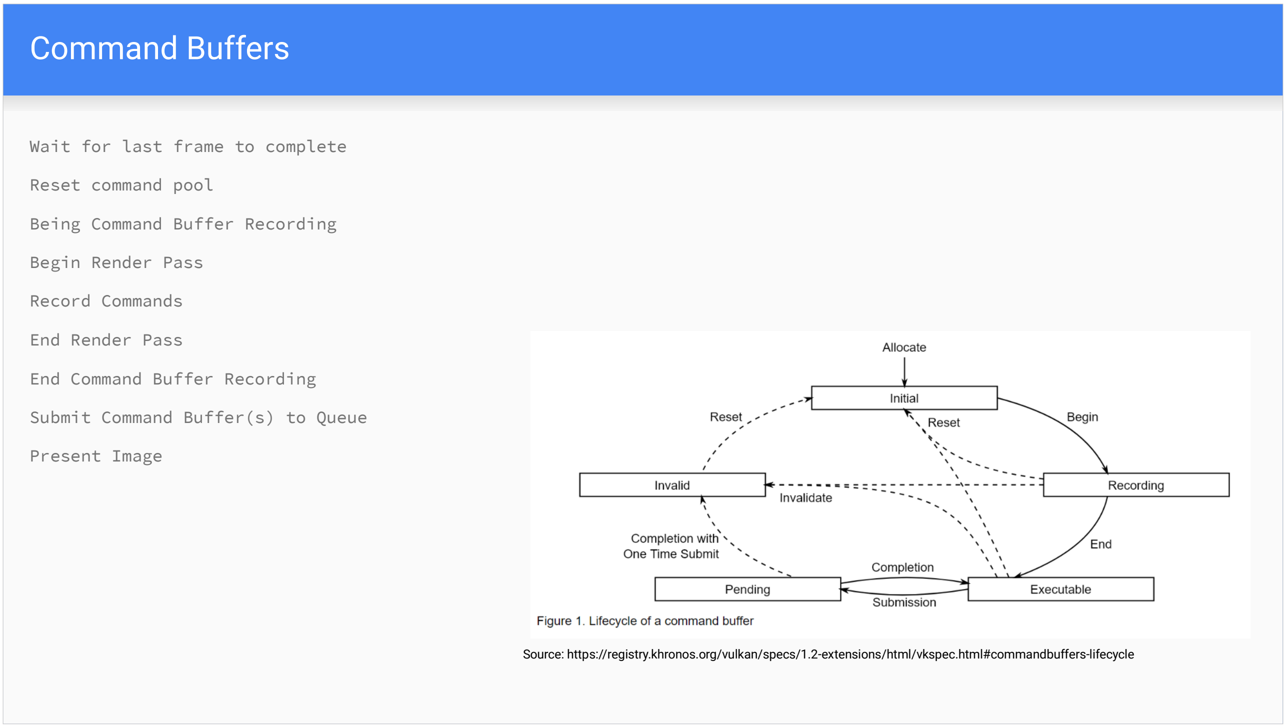
Fifo 4 2 1

```
enum VkPresentModeKHR {
    VK_PRESENT_MODE_IMMEDIATE_KHR = 0,
    VK_PRESENT_MODE_MAILBOX_KHR = 1,
    VK_PRESENT_MODE_FIFO_KHR = 2,
    VK_PRESENT_MODE_FIFO_RELAXED_KHR = 3,
}
```

Present mode controls when swapchain images are displayed on screen relative to vsync.

- Immediate basically means vsync off: the application submits images as fast as it can. This will produce tearing as the compositor might be in the middle of displaying an image when the update update the image to display
- Mailbox instead corresponds to vsync on: each image waits until the next vsync before presenting. No tearing can be observed
- Fifo combine aspect of immediate and mailbox: the compositor keeps a queue of images to present and when we present a swapchain image, the image is added to the queue.
  In immediate mode, the queue has size 1. When presenting an image, the active image is immediately replaced
  In mailbox mode, when presenting an image we add an image at the end of the queue. The compositor will work its way through the queue
  In fifo mode, the queue has size 2: the current image being present, and the next image. If the app renders fast enough, it might be able to update the next image to be displayed. In the example above, we skip 3 as it was replaced by 4 before 3 was presented

Image count controls how many images you can push to the queue before having to wait to re-use a previous image. Most applications use 2 or 3, depending on the workload and how much latency you can tolerate.



Command Buffers

```
Wait for last frame to complete
Reset command pool
Being Command Buffer Recording
Begin Render Pass
Record Commands
End Render Pass
End Command Buffer Recording
Submit Command Buffer(s) to Queue
Present Image
```

Figure 1. Lifecycle of a command buffer

Source: https://registry.khronos.org/vulkan/specs/1.2-extensions/html/vkspec.html#commandbuffers-lifecycle

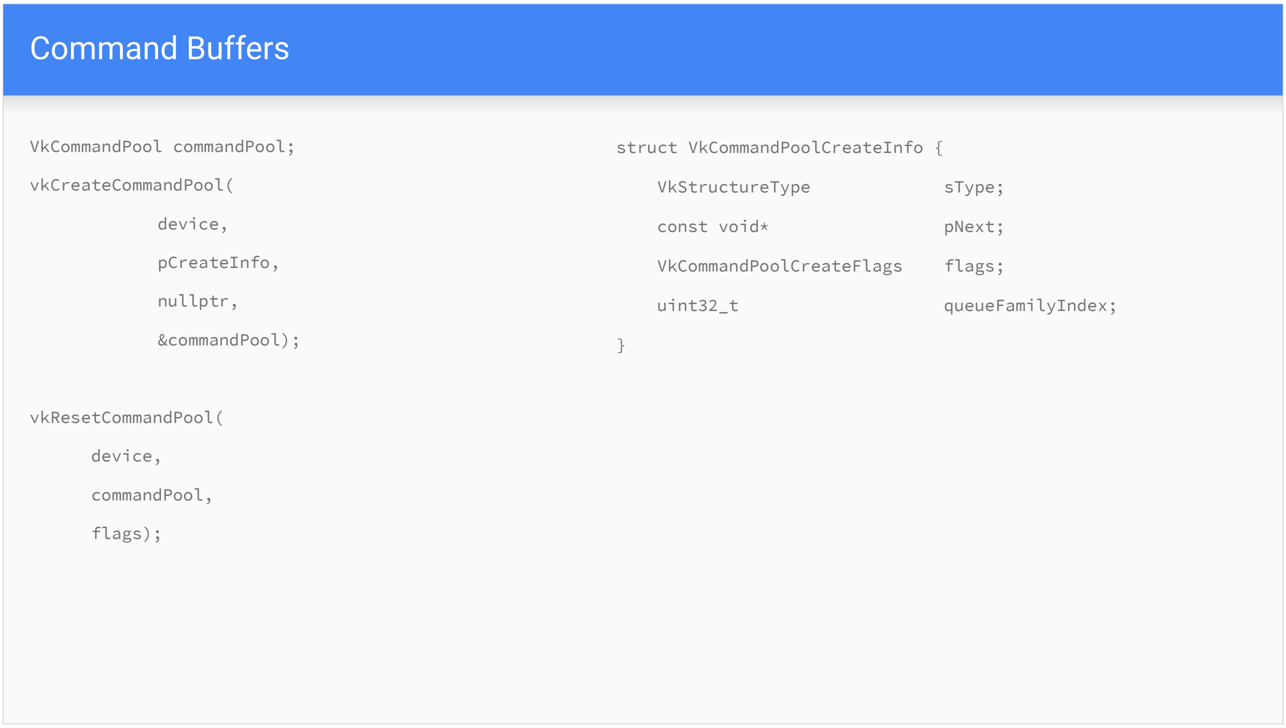
Now that we have created all of the objects we need for rendering is time to issue some commands to get the GPU to do some work! The main interface to do this in Vulkan is the command buffer.

If you’re coming from OpenGL this might be new: the driver managed this for you.

However the concept is quite simple: we record the operation we want to perform (draw, dispatch, etc.) in a buffer and then submit all recorded commands at once. Before we dive into details, we provide an overview of a typical render loop:

- We wait for the last frame rendering to complete. This is needed as in Vulkan we are responsible for synchronizing access to resources like command buffers and queues
- We prepare our command buffer for recording and we begin our render pass (if we are rendering into an image)
- We record the rendering commands
- Once we are done we close the render pass and and end the command buffer
- We submit the command buffer to the device queue we created earlier
- Finally we present our image

The image on the right illustrates the lifecycle of a command buffer.



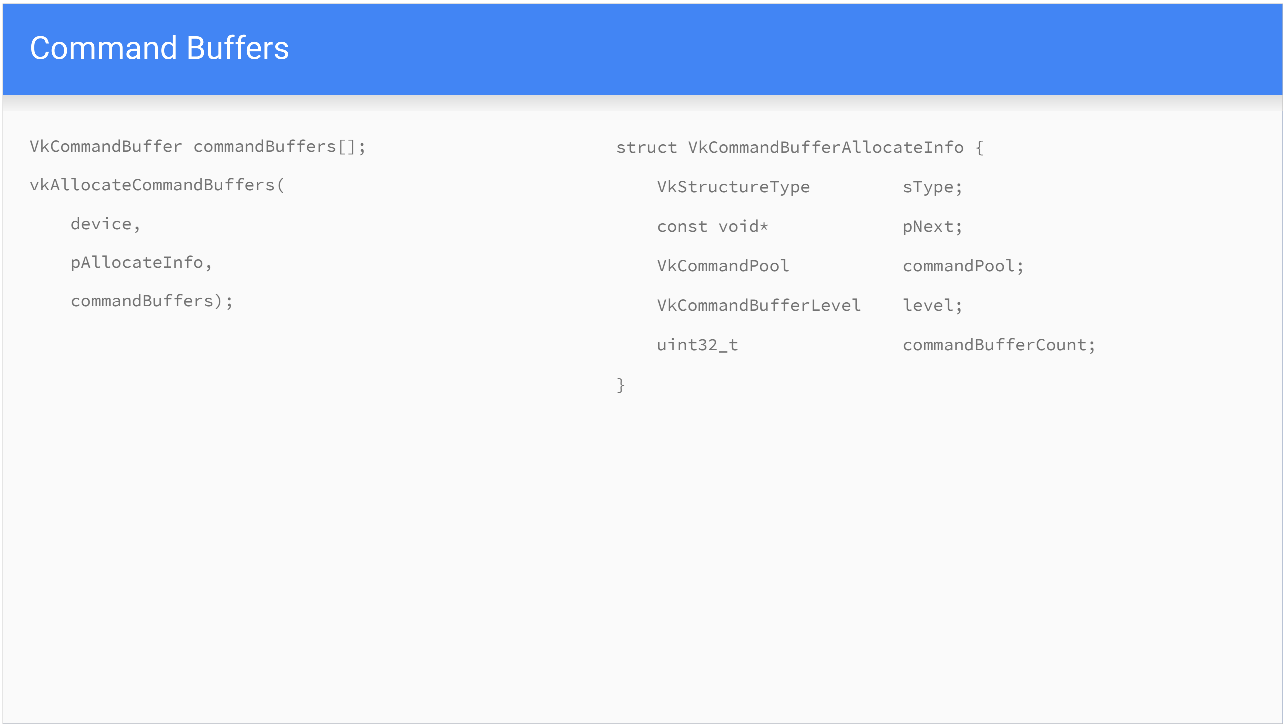
Command Buffers

```
VkCommandPool commandPool;
vkCreateCommandPool(
            device,
            pCreateInfo,
            nullptr,
            &commandPool);

vkResetCommandPool(
      device,
      commandPool,
      flags);
```

```
struct VkCommandPoolCreateInfo {
    VkStructureType            sType;
    const void*                pNext;
    VkCommandPoolCreateFlags   flags;
    uint32_t                   queueFamilyIndex;
}
```

Command buffers are allocated from command pools. As you can see they are really simple to create. We recommend creating 2/3 command pools, depending on how many frames you are pipelining. You also need a command pool per queue type - you can’t submit a command buffer created for a graphics queue to a compute queue.

As we mentioned in the previous slide, you need to reset the command pool before recording. The API allows you to reset individual command buffers, but all HW vendors recommend resetting the whole command pool for performance reasons.

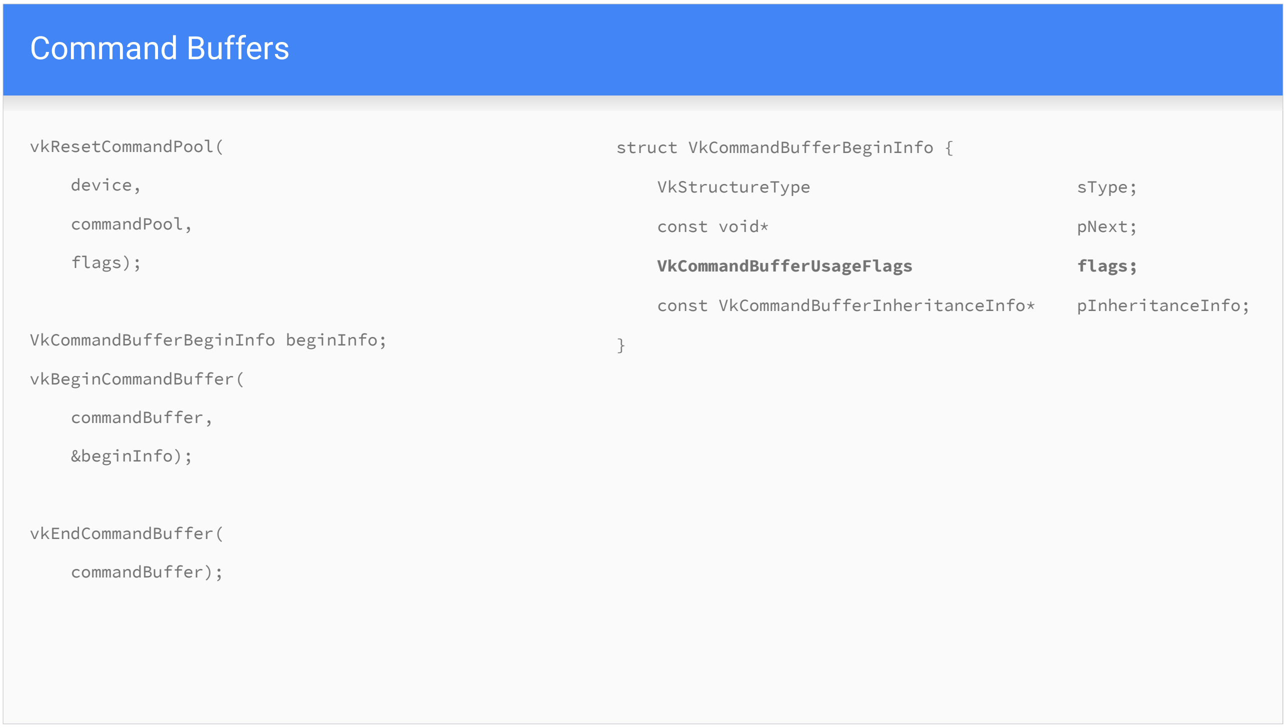


Command Buffers

```
VkCommandBuffer commandBuffers[];
vkAllocateCommandBuffers(
    device,
    pAllocateInfo,
    commandBuffers);
```

```
struct VkCommandBufferAllocateInfo {
    VkStructureType         sType;
    const void*             pNext;
    VkCommandPool           commandPool;
    VkCommandBufferLevel    level;
    uint32_t                commandBufferCount;
}
```

Allocating a command buffer is also simple. You can allocate multiple command buffers from a single pool if you need to. Vulkan distinguishes between primary and secondary buffers, but only primary command buffers are used in practice (that we know of). In theory secondary command buffers were introduced to make it easier to record multiple command buffers in parallel, but they have too many restrictions and in most implementations they are too slow. We’ll show how to multi-thread your code in a moment.

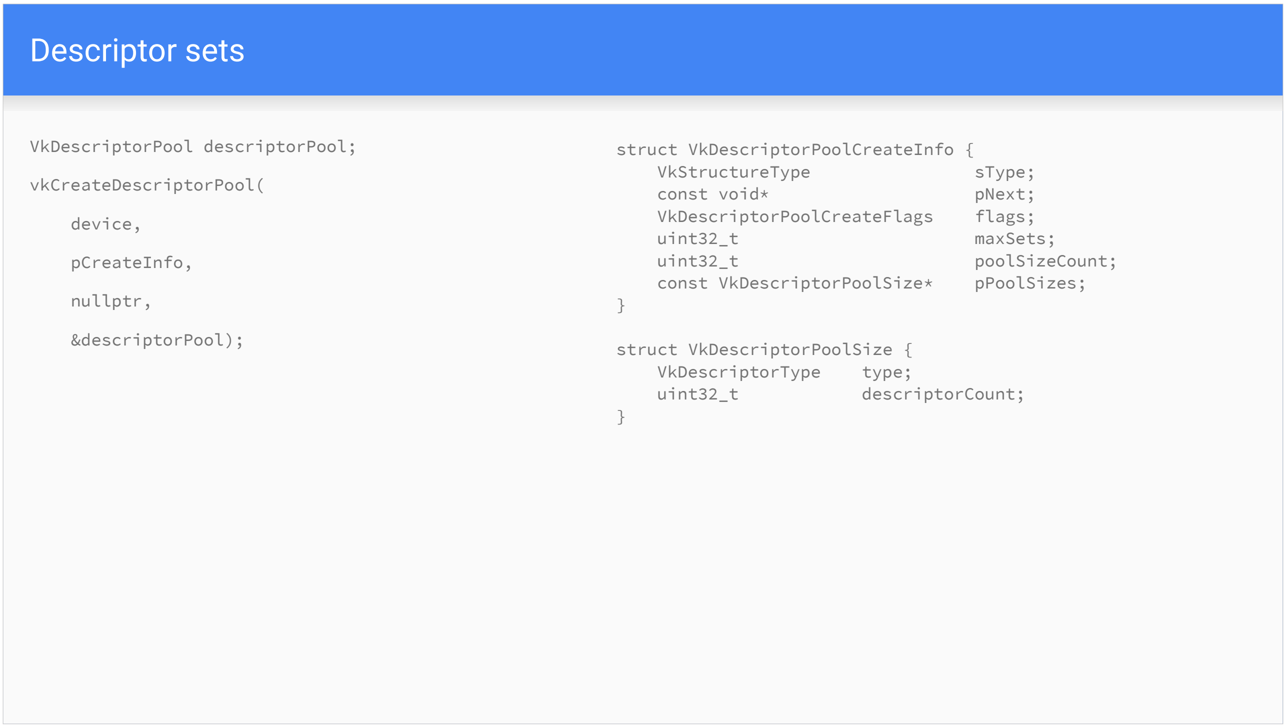


Command Buffers

```
vkResetCommandPool(
    device,
    commandPool,
    flags);

VkCommandBufferBeginInfo beginInfo;
vkBeginCommandBuffer(
    commandBuffer,
    &beginInfo);

vkEndCommandBuffer(
    commandBuffer);
```

```
struct VkCommandBufferBeginInfo {
    VkStructureType                          sType;
    const void*                              pNext;
    VkCommandBufferUsageFlags                flags;
    const VkCommandBufferInheritanceInfo*    pInheritanceInfo;
}
```

Here we show the life cycle of the command buffer in code. For begin, we only care about the flags field, which is always populated with VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT (as we are going to reset the pool each frame).



Descriptor sets

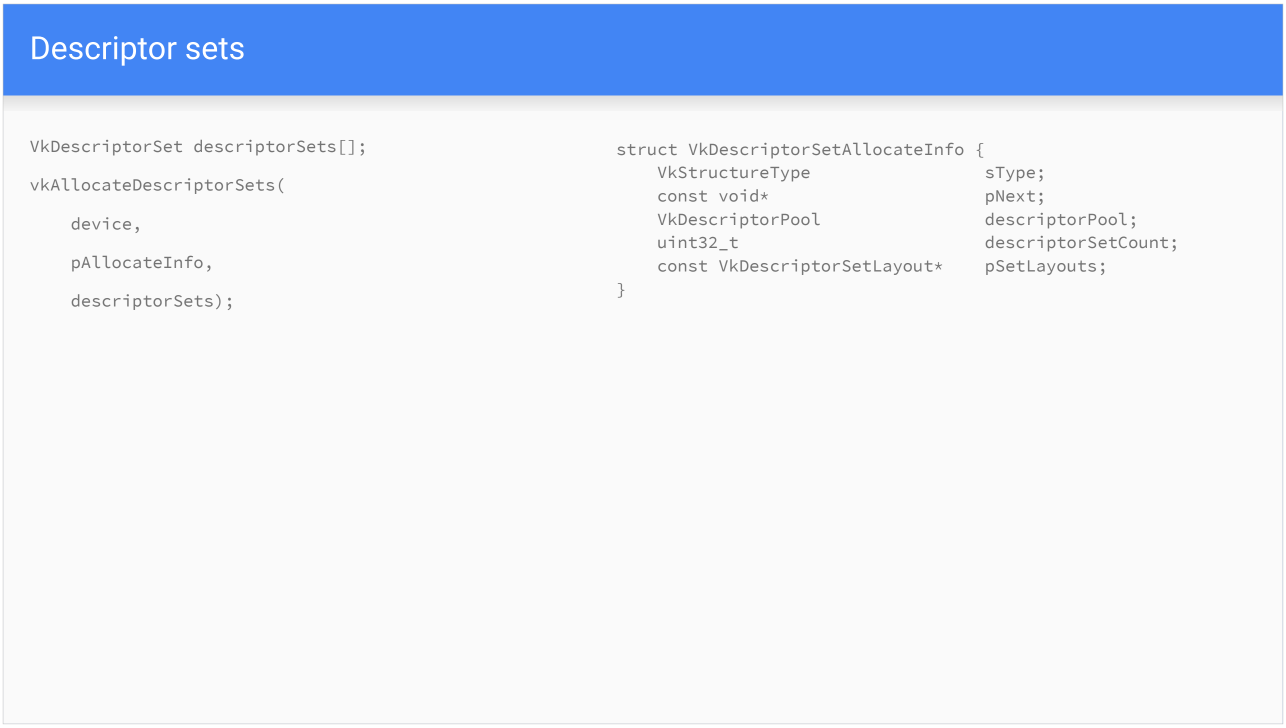
```
VkDescriptorPool descriptorPool;

vkCreateDescriptorPool(

    device,

    pCreateInfo,

    nullptr,

    &descriptorPool);
```

```
struct VkDescriptorPoolCreateInfo {
    VkStructureType               sType;
    const void*                   pNext;
    VkDescriptorPoolCreateFlags   flags;
    uint32_t                      maxSets;
    uint32_t                      poolSizeCount;
    const VkDescriptorPoolSize*   pPoolSizes;
}

struct VkDescriptorPoolSize {
    VkDescriptorType    type;
    uint32_t            descriptorCount;
}
```

Earlier we created a pipeline layout and we mentioned it was only a description of the number and type of resources that would be used at render time. Now the time has come to define the resources we are going to use when the program executes. These resources are defined in descriptor sets.

Like with command buffers, we need to create a descriptor pool to allocate descriptors from. When creating a pool we need to estimate how many descriptor sets we are going to allocate from it. If you exceed the maximum number of descriptor sets, you should either consider creating a new pool at runtime or increasing the size of your pool. Both approaches are valid, choose the one the best suit your needs.

We also need to determine how many descriptors per type can be allocated from the pool. Again, this involves a bit of guessing. As an alternative, you could create one pool per pipeline layout, as in that case you know exactly how many resources are used. However this possibly leads to the creation of many pools, which in turn might increase memory usage. Again, something you need to experiment with based on your needs.



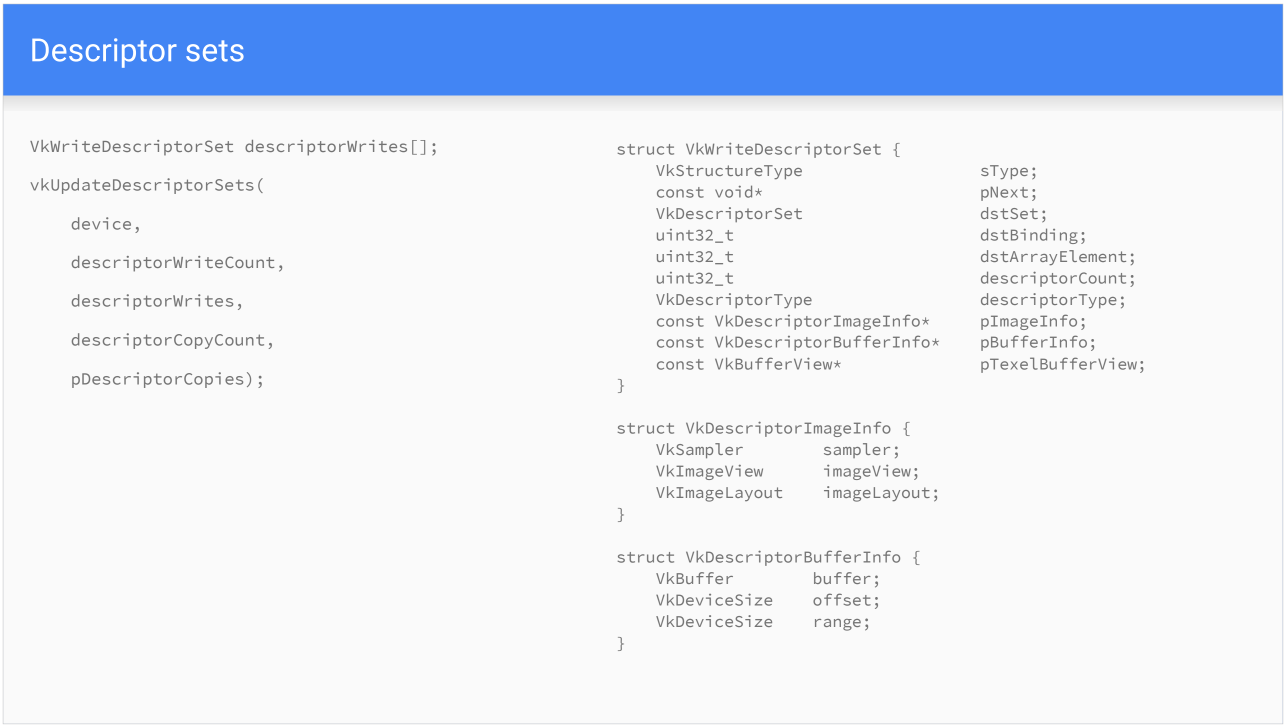
Descriptor sets

```
VkDescriptorSet descriptorSets[];

vkAllocateDescriptorSets(

    device,

    pAllocateInfo,

    descriptorSets);
```

```
struct VkDescriptorSetAllocateInfo {
    VkStructureType                sType;
    const void*                    pNext;
    VkDescriptorPool               descriptorPool;
    uint32_t                       descriptorSetCount;
    const VkDescriptorSetLayout*   pSetLayouts;
}
```

Now that we have a descriptor pool, we can allocate a descriptor set from it. Descriptor sets are expensive to create, our advice it to create them only once and then update them as needed. It is possible to reset a descriptor pool (like command buffer pools) and re-create all your descriptor sets, but this shouldn’t be done at each frame.

When allocating a descriptor set we provide the layout we created earlier.



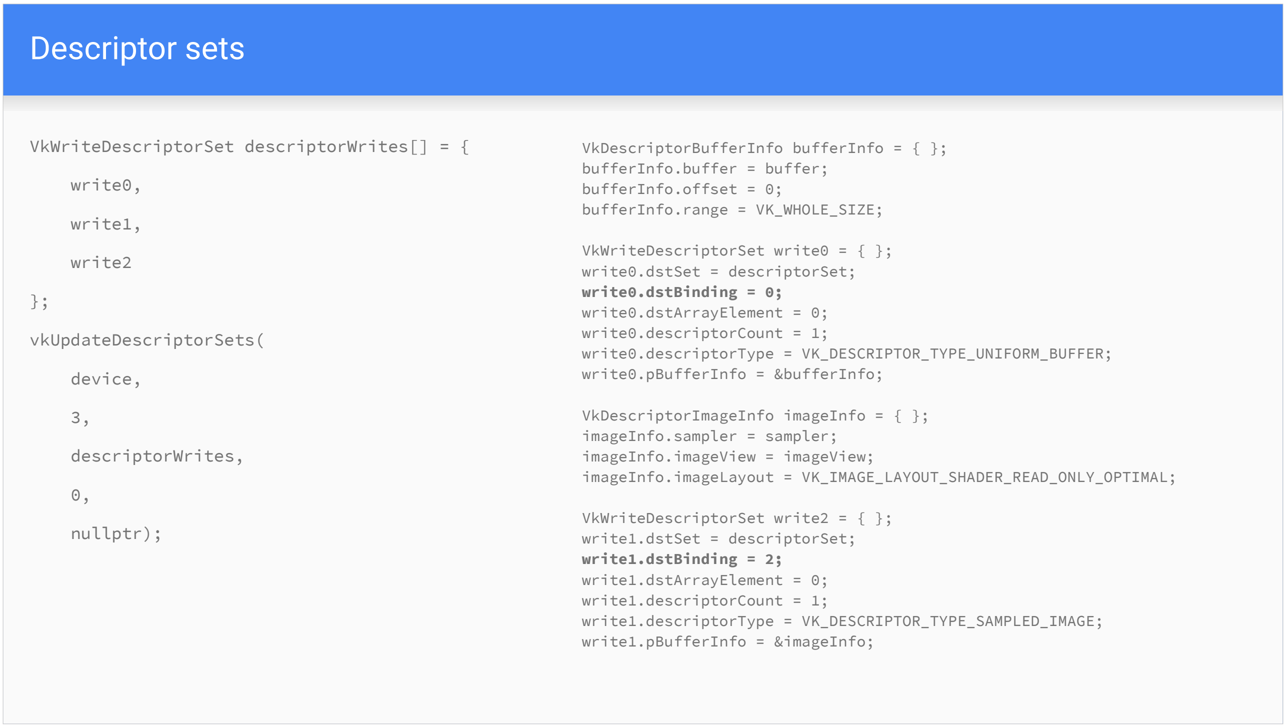
Descriptor sets

```
VkWriteDescriptorSet descriptorWrites[];

vkUpdateDescriptorSets(

    device,

    descriptorWriteCount,

    descriptorWrites,

    descriptorCopyCount,

    pDescriptorCopies);
```

```
struct VkWriteDescriptorSet {
    VkStructureType                 sType;
    const void*                     pNext;
    VkDescriptorSet                 dstSet;
    uint32_t                        dstBinding;
    uint32_t                        dstArrayElement;
    uint32_t                        descriptorCount;
    VkDescriptorType                descriptorType;
    const VkDescriptorImageInfo*    pImageInfo;
    const VkDescriptorBufferInfo*   pBufferInfo;
    const VkBufferView*             pTexelBufferView;
}

struct VkDescriptorImageInfo {
    VkSampler        sampler;
    VkImageView      imageView;
    VkImageLayout    imageLayout;
}

struct VkDescriptorBufferInfo {
    VkBuffer        buffer;
    VkDeviceSize    offset;
    VkDeviceSize    range;
}
```

The next step is to write the data into the descriptor set. This is where we specify the resources associated with this descriptor set. If you are re-using descriptors across multiple sets, you can copy them rather than write them. This should provide some performance improvements - we won’t cover the details here as copies are similar to writes.

The information we provide here is similar to the one we provide when creating the pipeline layout, except now we also specify which resources are bound for each entry.

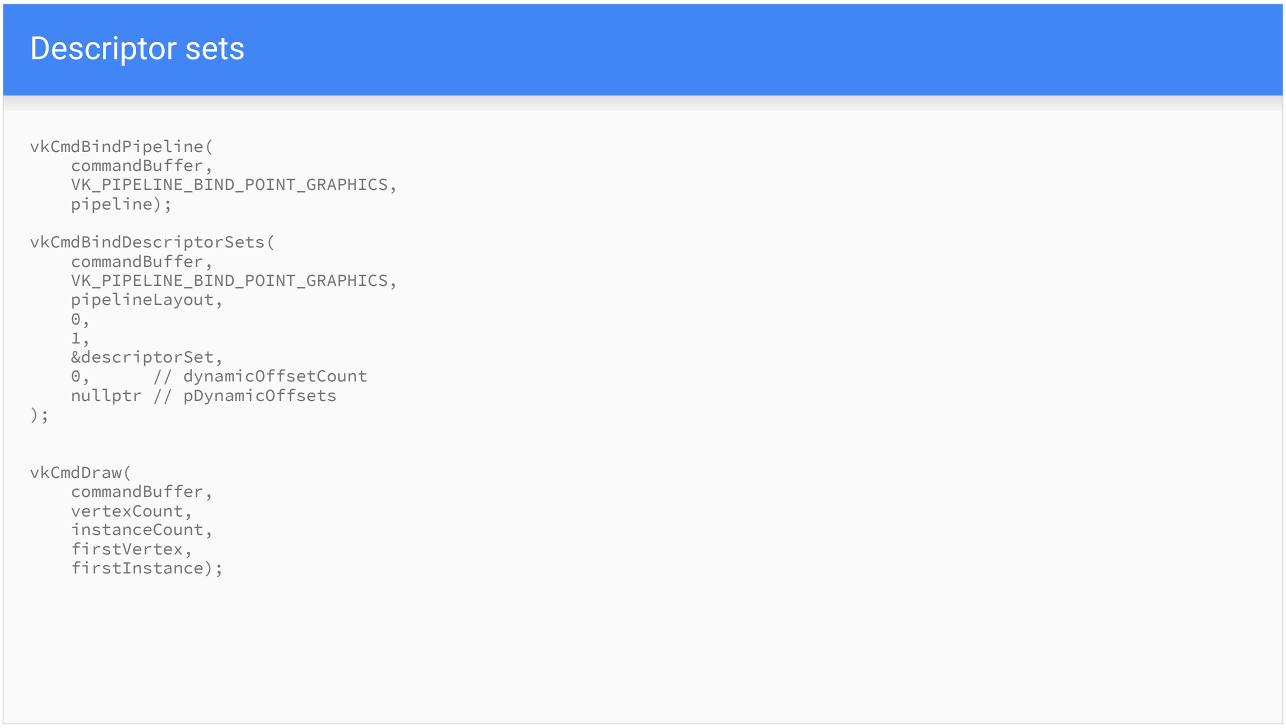


Descriptor sets

```
VkWriteDescriptorSet descriptorWrites[] = {
    write0,
    write1,
    write2
};
vkUpdateDescriptorSets(
    device,
    3,
    descriptorWrites,
    0,
    nullptr);
```

```
VkDescriptorBufferInfo bufferInfo = { };
bufferInfo.buffer = buffer;
bufferInfo.offset = 0;
bufferInfo.range = VK_WHOLE_SIZE;

VkWriteDescriptorSet write0 = { };
write0.dstSet = descriptorSet;
write0.dstBinding = 0;
write0.dstArrayElement = 0;
write0.descriptorCount = 1;
write0.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
write0.pBufferInfo = &bufferInfo;

VkDescriptorImageInfo imageInfo = { };
imageInfo.sampler = sampler;
imageInfo.imageView = imageView;
imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

VkWriteDescriptorSet write2 = { };
write1.dstSet = descriptorSet;
write1.dstBinding = 2;
write1.dstArrayElement = 0;
write1.descriptorCount = 1;
write1.descriptorType = VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE;
write1.pBufferInfo = &imageInfo;
```

Here’s what it would look like for our example shader. We have omitted the second uniform buffer as it’s basically the same as the first one. Now we have all the objects we need to submit commands to the GPU.



Descriptor sets

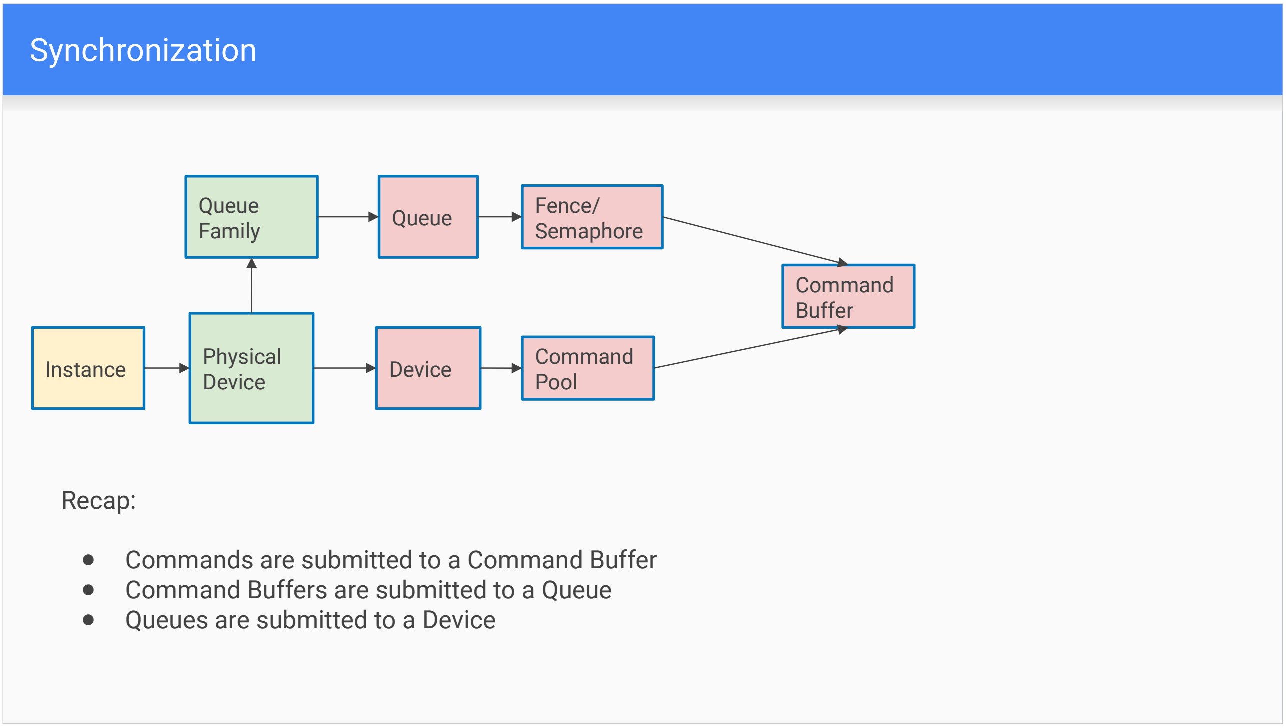
```
vkCmdBindPipeline(
    commandBuffer,
    VK_PIPELINE_BIND_POINT_GRAPHICS,
    pipeline);

vkCmdBindDescriptorSets(
    commandBuffer,
    VK_PIPELINE_BIND_POINT_GRAPHICS,
    pipelineLayout,
    0,
    1,
    &descriptorSet,
    0,      // dynamicOffsetCount
    nullptr // pDynamicOffsets
);


vkCmdDraw(
    commandBuffer,
    vertexCount,
    instanceCount,
    firstVertex,
    firstInstance);
```

And here’s how all the objects we have created are used to submit work to the GPU (assume we are inside a beingCommandBuffer/beginRenderPass section) :

- We have to bind a pipeline, to tell the driver which program is going to be used
- We then bind the active descriptor set, to tell the driver which resources are bound
- We then invoke the draw or compute command as needed

While a pipeline object contains all the state for our draw, we still need to be careful to avoid changing the pipeline (and descriptors) too often. If possible, try to sort your draws by pipeline (i.e. material) to reduce state changes as much as possible

We haven’t talked about dynamic offsets. They are useful to reduce descriptor changes as they allow us to provide buffer offsets at runtime. With this approach you can bind a single buffer and reuse it across draws. We usually use this for per-draw data (i.e. model transform, etc.)



Synchronization

Recap:

- Commands are submitted to a Command Buffer
- Command Buffers are submitted to a Queue
- Queues are submitted to a Device

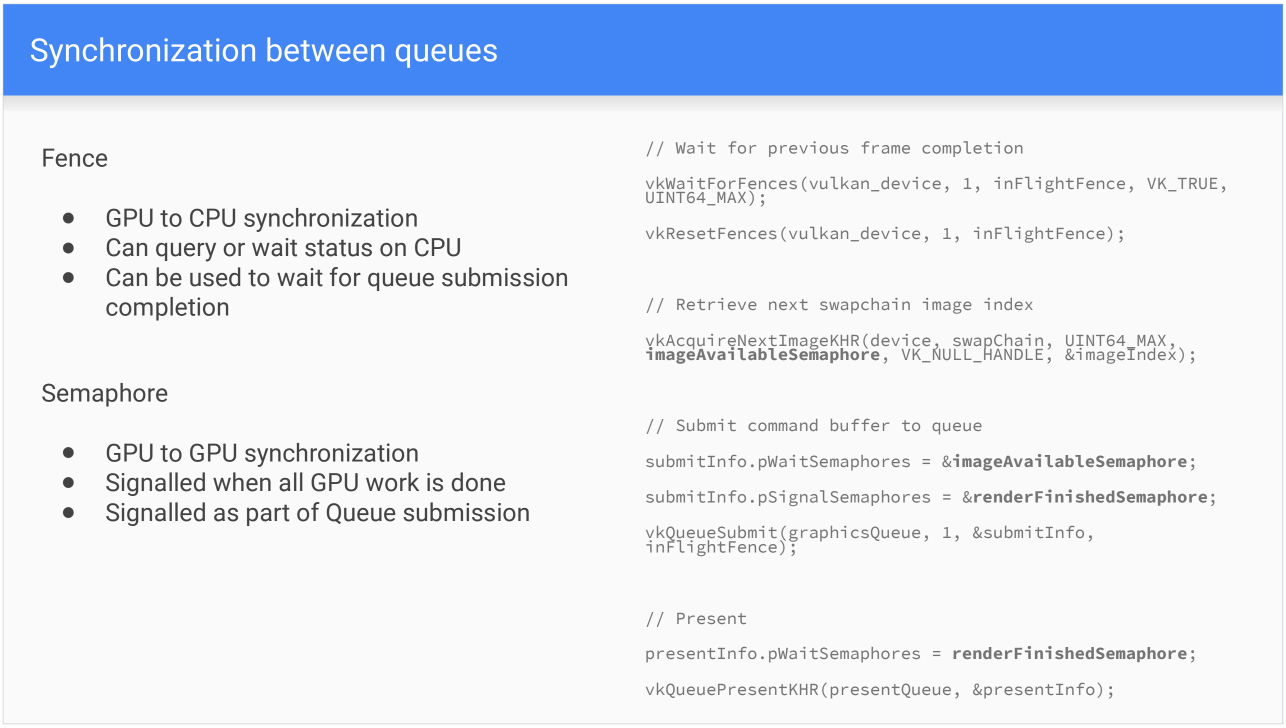
Synchronization is probably the most complex topic in Vulkan. We'll do our best to make all the moving parts as clear as possible, but don't expect to grasp all of it in a single sitting (we certainly didn't!).

We start with the synchronization at queue level. We have two primitives at this level: fences and semaphores.

- Fences are used for CPU<->GPU synchronization
- Semaphores are used for GPU<-> GPU synchronization

Timeline semaphores are the newest synchronization object, and can be used both for CPU<->GPU and GPU<->GPU synchronization.

There are two sets of synchronization primitives: between queues and inside a single queue.



Synchronization between queues

Fence

- GPU to CPU synchronization
- Can query or wait status on CPU
- Can be used to wait for queue submission completion

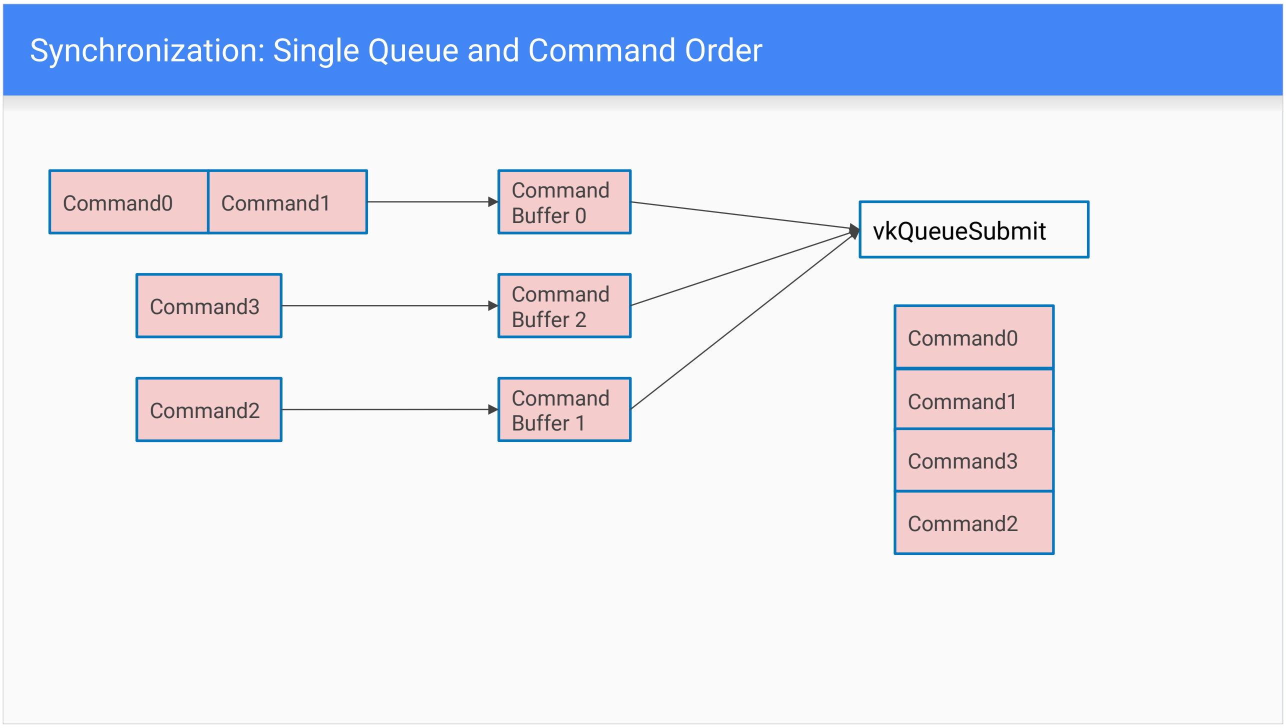
Semaphore

- GPU to GPU synchronization
- Signalled when all GPU work is done
- Signalled as part of Queue submission

```
// Wait for previous frame completion

vkWaitForFences(vulkan_device, 1, inFlightFence, VK_TRUE,
UINT64_MAX);

vkResetFences(vulkan_device, 1, inFlightFence);


// Retrieve next swapchain image index

vkAcquireNextImageKHR(device, swapChain, UINT64_MAX,
imageAvailableSemaphore, VK_NULL_HANDLE, &imageIndex);


// Submit command buffer to queue

submitInfo.pWaitSemaphores = &imageAvailableSemaphore;

submitInfo.pSignalSemaphores = &renderFinishedSemaphore;

vkQueueSubmit(graphicsQueue, 1, &submitInfo,
inFlightFence);


// Present

presentInfo.pWaitSemaphores = renderFinishedSemaphore;

vkQueuePresentKHR(presentQueue, &presentInfo);
```

As synchronization between queues, we mean also queues of different frames.
The simplest example we can give is to have a single queue per frame, and coordinate CPU and GPU to submit a frame.

Here we show some typical usages of fences and semaphores to coordinate a full frame.
We are creating a chain of execution that uses a fence to signal when the current commands submitted have finished. We will wait in the following frame on that on the CPU for that.
Then we create a chain between waiting for the next swapchain image to be available before executing the commands, using the pWaitSemaphores in QueueSubmit struct.
When that is done, it will signal another semaphore that will unlock the actual present on the screen.



Synchronization: Single Queue and Command Order

Command0 Command1

Command3

Command2

Command Buffer 0

Command Buffer 2

Command Buffer 1

vkQueueSubmit

Command0

Command1

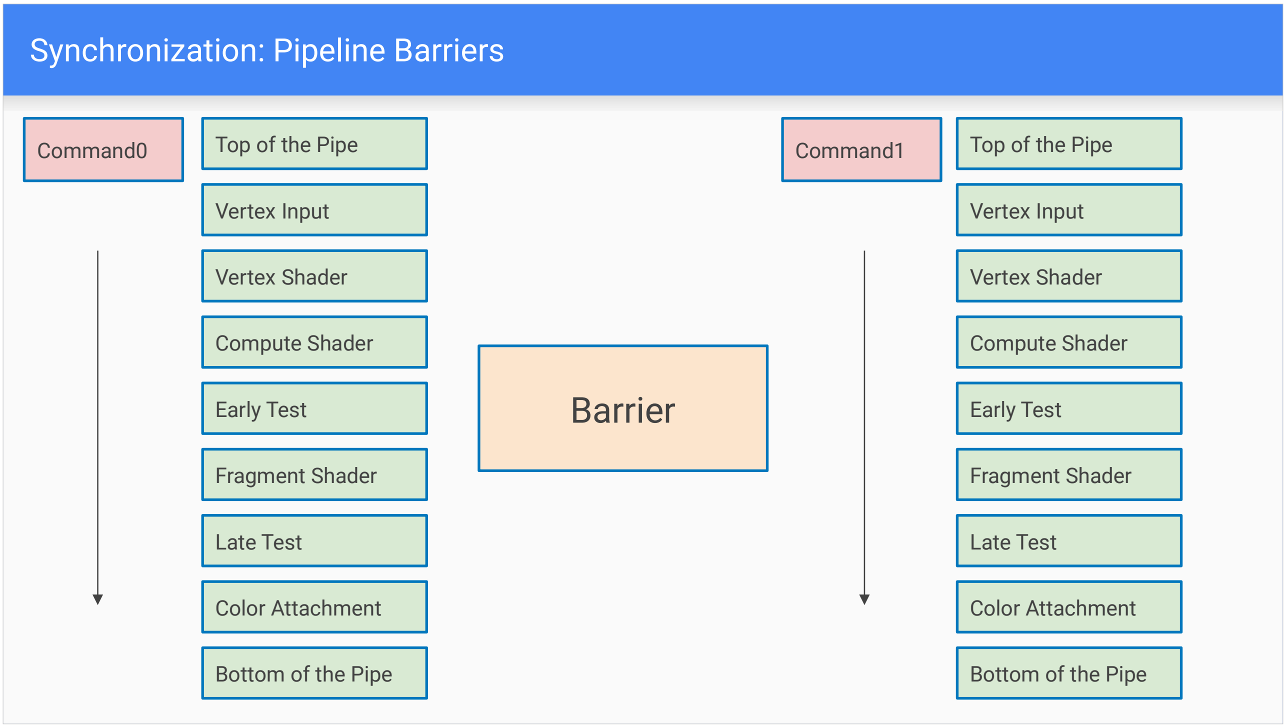
Command3

Command2

It is important to know the command execution order.

Queue submission sends the command buffers to be executed on the GPU. At this stage the commands are just linearly executed based on the order of the command buffers.
Here we see that if we submit command buffers with the order 0,2,1, the final command executed will be command 0,1,3,2.
This is important to remember: only the order of submission of the command buffers determines the order between commands.

The Vulkan spec only guarantees order of execution. This doesn't mean that the work of each command will also complete in that order. So, how can we wait for certain operations to complete on the GPU before doing more work ?
Enter another important element: barriers!



Synchronization: Pipeline Barriers

Command0

- Top of the Pipe
- Vertex Input
- Vertex Shader
- Compute Shader
- Early Test
- Fragment Shader
- Late Test
- Color Attachment
- Bottom of the Pipe

Barrier

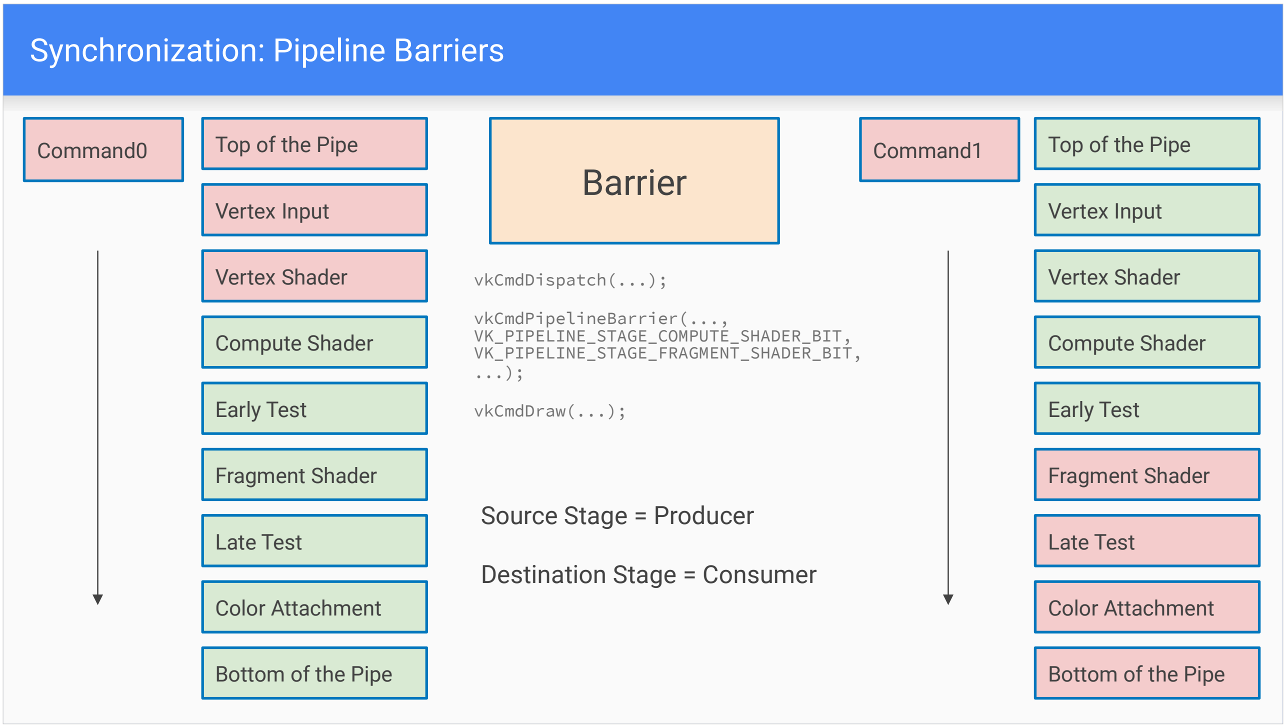
Command1

- Top of the Pipe
- Vertex Input
- Vertex Shader
- Compute Shader
- Early Test
- Fragment Shader
- Late Test
- Color Attachment
- Bottom of the Pipe

A Barrier is an object that ensures an order of execution between commands.
Each command goes through a series of stages that are outlined here: depending on the nature of the command (graphics work or compute work) some stages are present and other not.
But the important takeaway is that each Command executes some stages of the GPU pipeline.

The first type of barriers we see is the Pipeline barrier. Its usage is to enforce the waiting of the following Command in the following Command Buffer depending on some stages of the GPU pipeline execution.
Sometimes a Pipeline barrier is also called Execution barrier.

Let’s see an example.



Synchronization: Pipeline Barriers

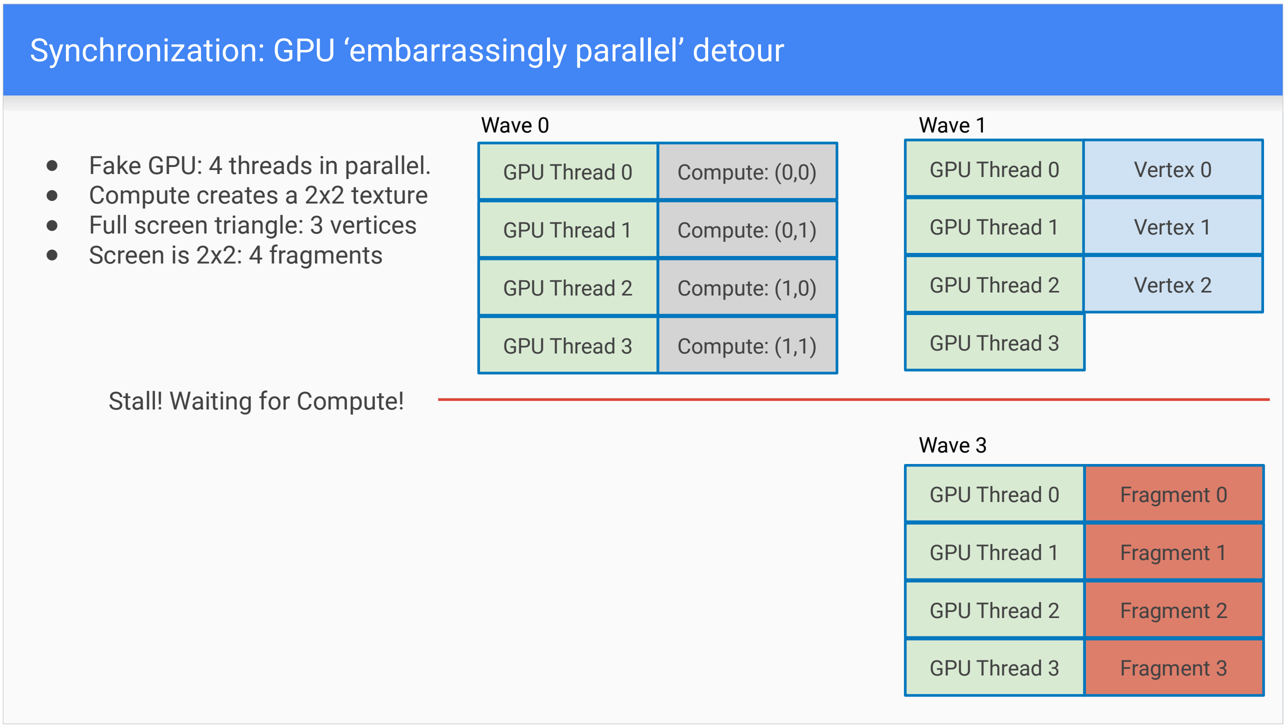
```
vkCmdDispatch(...);

vkCmdPipelineBarrier(...,
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
...);

vkCmdDraw(...);
```

Source Stage = Producer

Destination Stage = Consumer

For example, let’s visualize a use case: we write to a texture in a compute shader, and we then read it in a fragment shader.
We have 2 commands that dispatch or draw (commands like vkSetDescriptorSet are not execution commands, thus are ignored by the barriers): first we dispatch a compute shader that writes to a texture, then we issue a draw that reads that texture.

We add a pipeline barrier and we specify 2 parameters: the first is the source stage, the second the destination stage.
To help creating a different mental model of this, the source can be thought as the Producer while the destination as the Consumer.
Thus we can say that the second command’s fragment shader (vkCmdDraw) will wait until the Compute Shader stage of the previous command has finished all its executions to actually start the shaders. NOTE: the vertex shader of the second command, not having a dependency, can execute even if they some threads of the GPU are still working on the compute shader.



Synchronization: GPU ‘embarrassingly parallel’ detour

- Fake GPU: 4 threads in parallel.
- Compute creates a 2x2 texture
- Full screen triangle: 3 vertices
- Screen is 2x2: 4 fragments

Wave 0

GPU Thread 0	Compute: (0,0)
GPU Thread 1	Compute: (0,1)
GPU Thread 2	Compute: (1,0)
GPU Thread 3	Compute: (1,1)

Wave 1

GPU Thread 0	Vertex 0
GPU Thread 1	Vertex 1
GPU Thread 2	Vertex 2
GPU Thread 3	

Stall! Waiting for Compute!

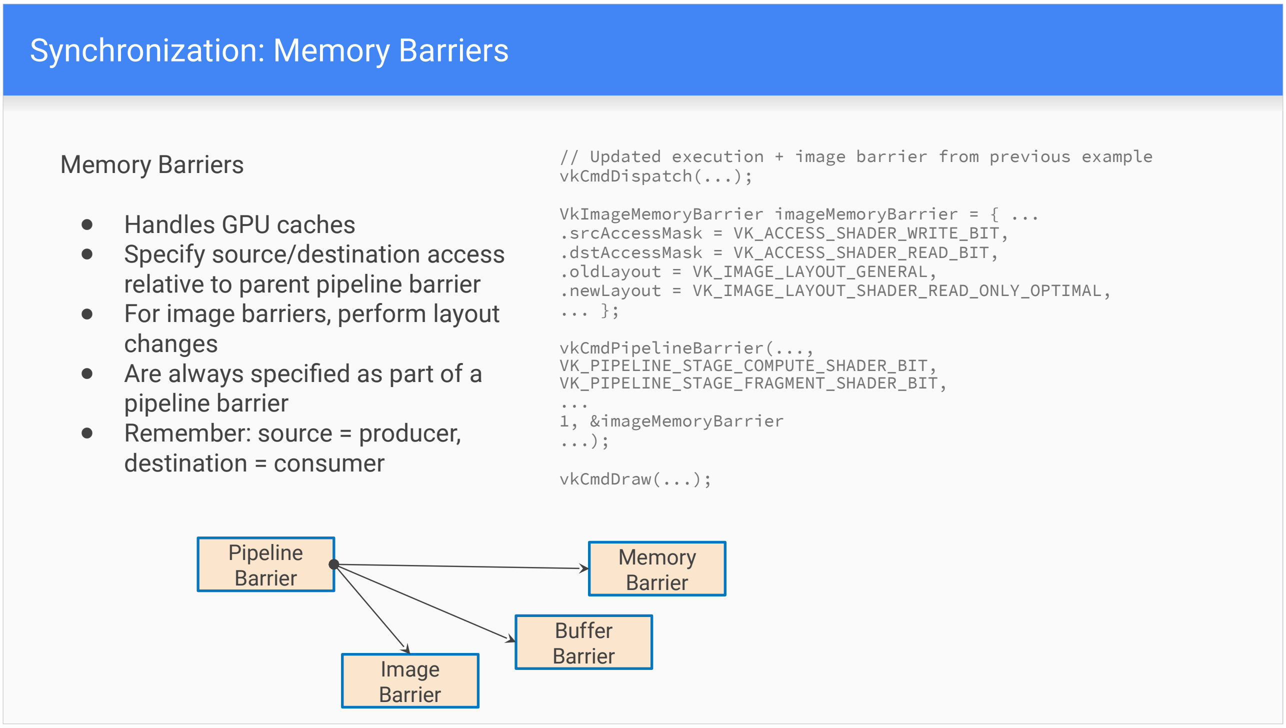
Wave 3

GPU Thread 0	Fragment 0
GPU Thread 1	Fragment 1
GPU Thread 2	Fragment 2
GPU Thread 3	Fragment 3

To help visualize even further the execution, we will have a quick and simplified conceptualization of the GPU.
When the GPU executes commands, it will decompose them in smaller tasks that can occupy 1 wave. Each wave then executes multiple threads in parallel, 4 in our example.
In the previous example, when we execute the compute shader and then ask to draw, when we draw there can be some threads that are still executing some compute work.
A barrier thus enforces some waiting on the GPU so that all the operations of a certain stage are finished.

In this small conceptualization, if we have a GPU with 4 threads (wow!) we can have a workload as in the slide. Back to the dispatch followed by the draw, we can arrive at executing all the vertex work of the second command without waiting, having the GPU running some compute and some vertex work.
If we have threads left unused we need to wait for all the compute work to finish before actually starting executing them.

It is much complex than this, but as a mind model can be helpful to visualize what is happening here.



Synchronization: Memory Barriers

Memory Barriers

- Handles GPU caches
- Specify source/destination access relative to parent pipeline barrier
- For image barriers, perform layout changes
- Are always specified as part of a pipeline barrier
- Remember: source = producer, destination = consumer

```
// Updated execution + image barrier from previous example
vkCmdDispatch(...);

VkImageMemoryBarrier imageMemoryBarrier = { ...
.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
.dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
.oldLayout = VK_IMAGE_LAYOUT_GENERAL,
.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
... };

vkCmdPipelineBarrier(...,
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
...
1, &imageMemoryBarrier
...);

vkCmdDraw(...);
```

Pipeline Barrier

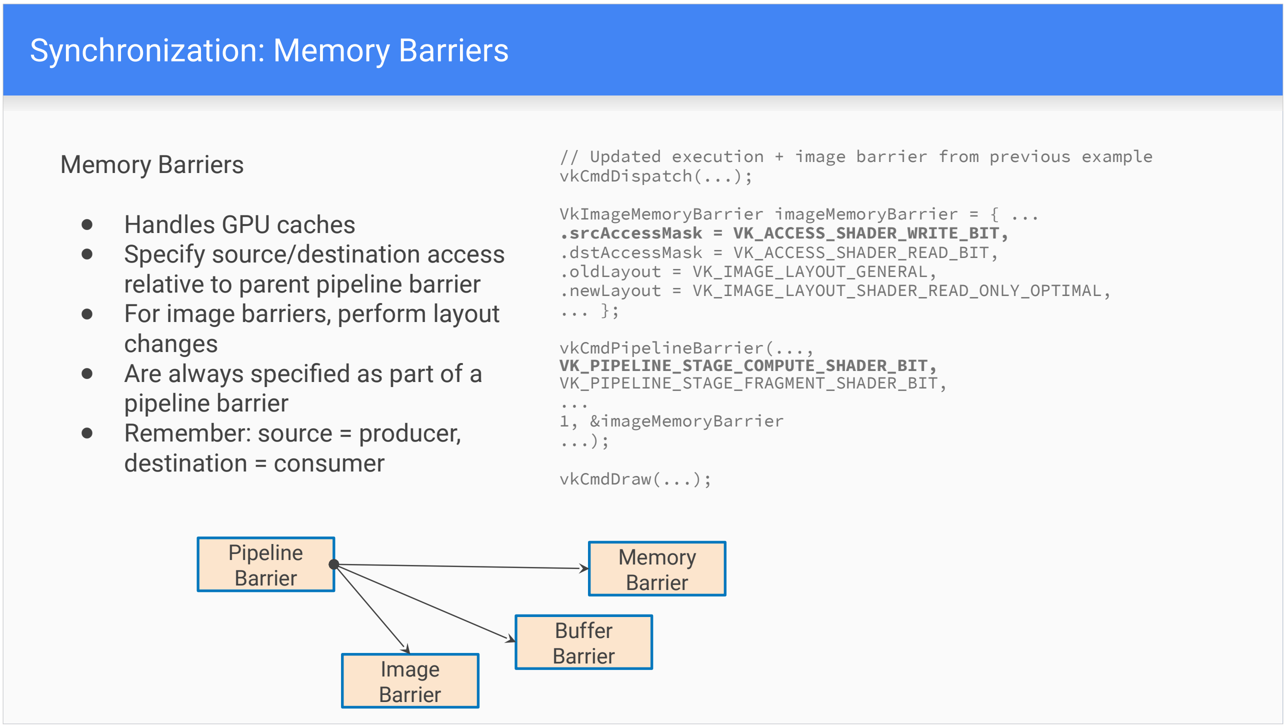
Memory Barrier

Buffer Barrier

Image Barrier

Back to synchronization objects, after we ensure a certain execution order, we need to ensure some memory validation mechanism.
There are three different memory barriers: memory barriers, buffer memory barriers and image memory barriers.

Memory barriers are always specified as part of a pipeline barrier as additional arguments to the vkCmdPipelineBarrier function.
There are three memory barrier: a ‘global’ one, buffer memory and image memory.
They all have in common the sourceAccessMask and destinationAccessMasks: these are telling the GPU how to handle access to that resource before and after the barrier



Synchronization: Memory Barriers

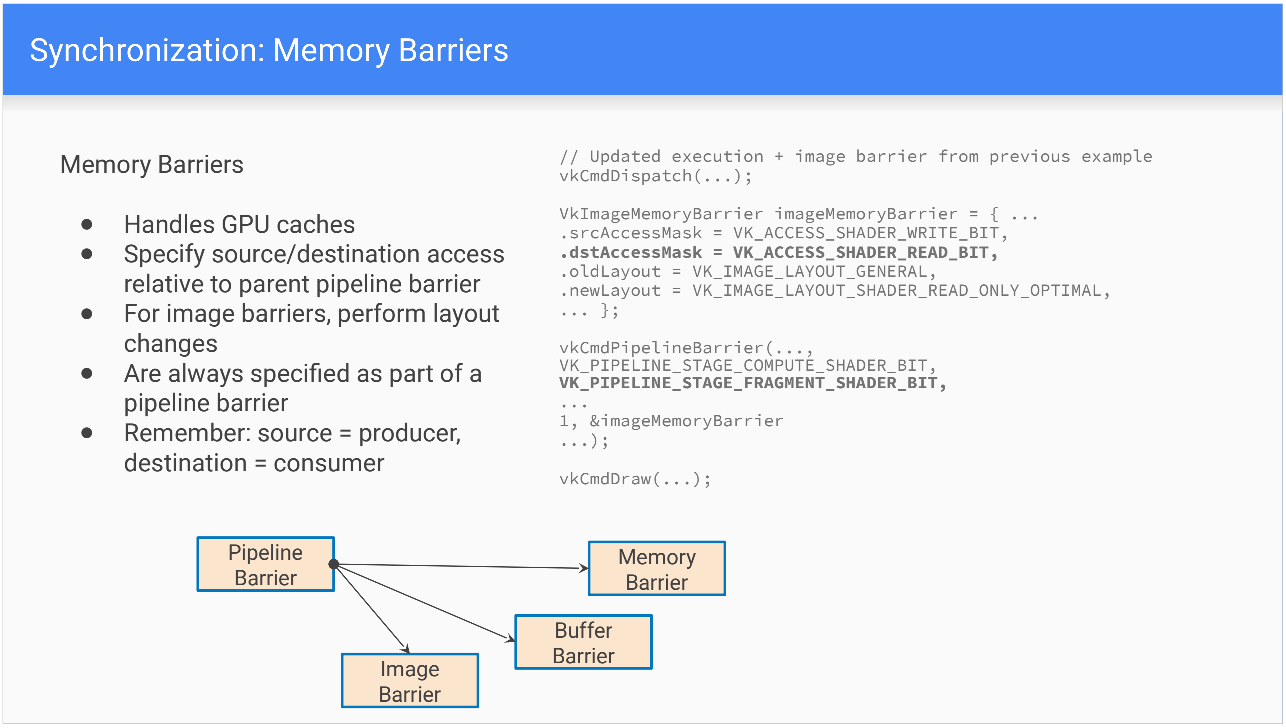
Memory Barriers

- Handles GPU caches
- Specify source/destination access relative to parent pipeline barrier
- For image barriers, perform layout changes
- Are always specified as part of a pipeline barrier
- Remember: source = producer, destination = consumer

```
// Updated execution + image barrier from previous example
vkCmdDispatch(...);

VkImageMemoryBarrier imageMemoryBarrier = { ...
.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
.dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
.oldLayout = VK_IMAGE_LAYOUT_GENERAL,
.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
... };

vkCmdPipelineBarrier(...,
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
...
1, &imageMemoryBarrier
...);

vkCmdDraw(...);
```

Pipeline Barrier
Memory Barrier
Buffer Barrier
Image Barrier

In this case, this image memory barrier is telling the GPU that the compute shader will write into the specified texture.
NOTE: the compute shader called in vkCmdDispatch is the PRODUCER of the resource, writing memory in the compute shader.
After all the threads of that compute executes, GPU can update the cache so that subsequent reads have updated data.



Synchronization: Memory Barriers

Memory Barriers

- Handles GPU caches
- Specify source/destination access relative to parent pipeline barrier
- For image barriers, perform layout changes
- Are always specified as part of a pipeline barrier
- Remember: source = producer, destination = consumer

```
// Updated execution + image barrier from previous example
vkCmdDispatch(...);

VkImageMemoryBarrier imageMemoryBarrier = { ...
.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
.dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
.oldLayout = VK_IMAGE_LAYOUT_GENERAL,
.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
... };

vkCmdPipelineBarrier(...,
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
...
1, &imageMemoryBarrier
...);

vkCmdDraw(...);
```

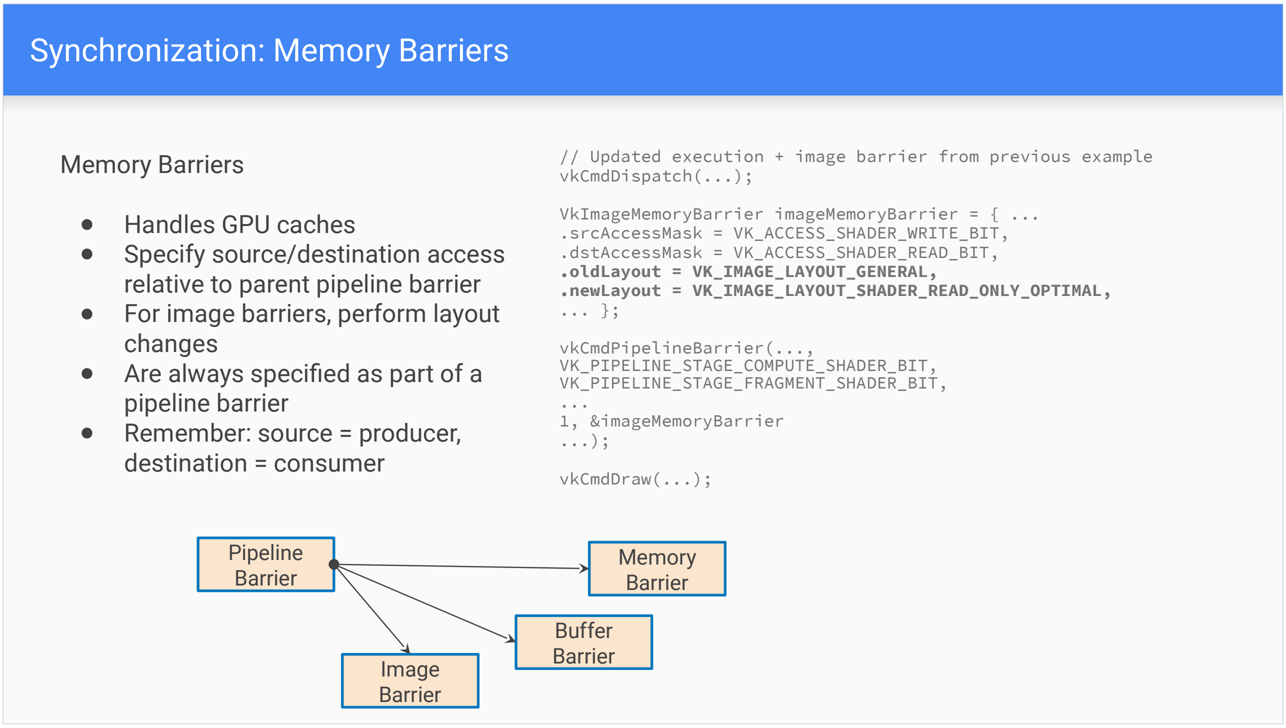
Pipeline Barrier

Memory Barrier

Buffer Barrier

Image Barrier

On the destination side we are telling that the vkCmdDraw’s fragment shader will read the texture (VK_ACCESS_SHADER_READ, STAGE_FRAGMENT_SHADER_BIT).
One final element that only Image Memory Barriers have is the layout transition, that we will see in the next slide.



Synchronization: Memory Barriers

Memory Barriers

- Handles GPU caches
- Specify source/destination access relative to parent pipeline barrier
- For image barriers, perform layout changes
- Are always specified as part of a pipeline barrier
- Remember: source = producer, destination = consumer

```
// Updated execution + image barrier from previous example
vkCmdDispatch(...);

VkImageMemoryBarrier imageMemoryBarrier = { ...
.srcAccessMask = VK_ACCESS_SHADER_WRITE_BIT,
.dstAccessMask = VK_ACCESS_SHADER_READ_BIT,
.oldLayout = VK_IMAGE_LAYOUT_GENERAL,
.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL,
... };

vkCmdPipelineBarrier(...,
VK_PIPELINE_STAGE_COMPUTE_SHADER_BIT,
VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
...
1, &imageMemoryBarrier
...);

vkCmdDraw(...);
```

Layouts are a way of describing how the image will be used. They can be used to determine the access mask (and thus the memory access) that will be used when issuing a barrier.
In Vulkan an image can contain multiple subresources (like mipmaps), and a layout works exactly on one of those subresources.
When reading/writing to an image in a compute shader, the layout used is VK_IMAGE_LAYOUT_GENERAL.
When reading the image in a fragment program the layout is VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL.

Layout transitions ensure that the GPU is aware of the change in use for the resource. Depending on the type of access, it might need to flush caches, decompress memory, etc. More importantly, it ensures a correct behavior of the application. Missing a barrier or using the incorrect layout can cause data corruption. Validation layers will help catch some of these issues. There is also a synchronization validation layer that can be enabled using the tool we mentioned on page 9. The synchronization validation layer will perform a more thorough validation compared to the standard validation layer.

This can be complicated and confusing, and still now it is in evolution.
Luckily there are few resources around that covers most examples needed (like https://github.com/KhronosGroup/Vulkan-Docs/wiki/Synchronization-Examples-(Legacy-synchronization-APIs)
).
The new Synchronization2 extensions simplified a little the code (with examples here https://github.com/KhronosGroup/Vulkan-Docs/wiki/Synchronization-Examples), but the core concepts are the same.

References

- Vulkan spec: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html
- SPIR-V spec: https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html
- GLSL extensions: https://github.com/KhronosGroup/GLSL/tree/master/extensions
- Vulkan guide: https://github.com/KhronosGroup/Vulkan-Guide
- Examples, best practices and much more: https://www.vulkan.org/learn
- 3D Graphics Rendering Cookbook, Sergey Kosarevsky and Viktor Latypov, Packt Publishing, 2021
- Mastering Graphics Programming with Vulkan, Marco Castorina and Gabriel Sassone, Packt Publishing, 2023

Break

Questions?