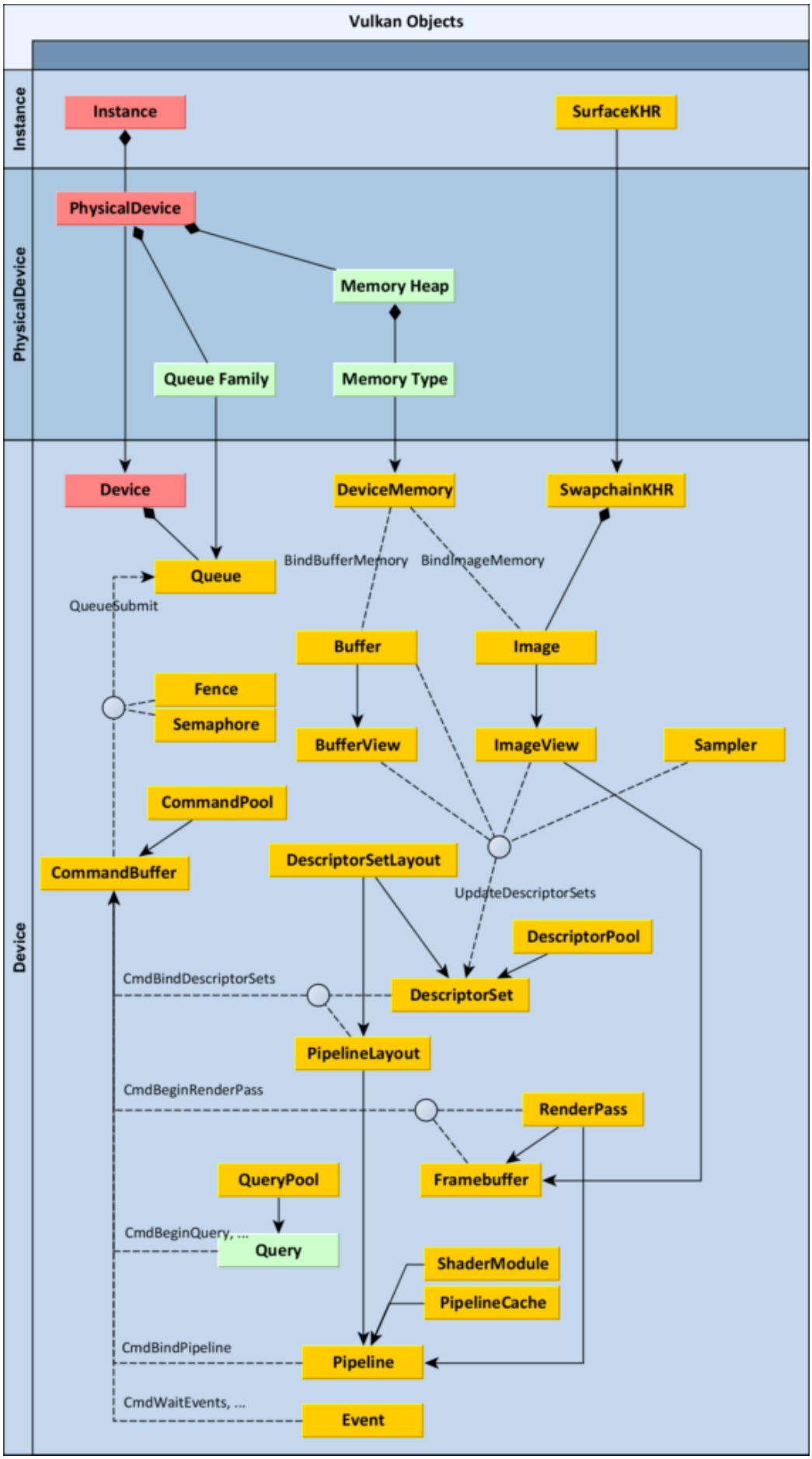
Understanding Vulkan objects

An important part of learning the Vulkan® API – just like any other API – is to understand what types of objects are defined in it, what they represent and how they relate to each other. To help with this, we've created a diagram that shows all of the Vulkan objects and some of their relationships, especially the order in which you create one from another.

Each Vulkan object is a value of a specific type with the prefix `Vk` . These prefixes are omitted in the diagram for clarity, just like the `vk` prefix for function names. For example, Sampler on the diagram means that there is a Vulkan object type called `VkSampler` . These types shouldn't be treated as pointers or ordinal numbers. You shouldn't interpret their values in any way. Just treat them as opaque handles, pass them from one function to another and of course do not forget to destroy them when they are no longer needed. Objects with a green background don't have their own types. Instead they are represented by numeric index of type `uint32_t` inside their parent object, like Queries inside QueryPool.

Solid lines with arrows represent the order of creation. For example, you must specify an existing DescriptorPool to create a DescriptorSet. Solid lines with a diamond represent composition, which means that you don't have to create that object, but it already exists inside its parent object and can be fetched from it. For example, you can enumerate PhysicalDevice objects from an Instance object. Dashed lines represent other relationships, like submitting various commands to a CommandBuffer.

The diagram is divided into three sections. Each section has a main object, shown in red. All other objects in a section are created directly or indirectly from that main object. For example, `vkCreateSampler` – the function that creates a Sampler – takes `VkDevice` as its first parameter. Relationships to main objects are not drawn on this diagram for clarity.


Vulkan Objects
Instance
PhysicalDevice
Device
Instance
SurfaceKHR
PhysicalDevice
Memory Heap
Queue Family
Memory Type
Device
DeviceMemory
SwapchainKHR
Queue
BindBufferMemory
BindImageMemory
QueueSubmit
Buffer
Image
Fence
Semaphore
BufferView
ImageView
Sampler
CommandPool
CommandBuffer
DescriptorSetLayout
UpdateDescriptorSets
DescriptorPool
CmdBindDescriptorSets
DescriptorSet
PipelineLayout
CmdBeginRenderPass
RenderPass
QueryPool
Framebuffer
CmdBeginQuery, ...
Query
ShaderModule
PipelineCache
CmdBindPipeline
Pipeline
CmdWaitEvents, ...
Event

Here is a brief description of all the objects:

Instance is the first object you create. It represents the connection from your application to the Vulkan runtime and therefore only should exist once in your application. It also stores all application specific state required to use Vulkan. Therefore you must specify all layers (like the Validation Layer) and all extensions you want to enable when creating an Instance.

PhysicalDevice represents a specific Vulkan-compatible device, like a graphics card. You enumerate these from "Instance" and you can then query them for their vendorID , deviceID , and supported features, as well as other properties and limits.

PhysicalDevice can enumerate all available types of **Queue Families**. The graphics queue is the main one, but you may also have additional ones that support only Compute or Transfer.

PhysicalDevice can also enumerate Memory Heaps and Memory Types inside them.
A **Memory Heap** represents a specific pool of RAM. It may abstract your system RAM on the motherboard or a certain memory space in video RAM on a dedicated graphics card, or any other host- or device-specific memory the implementation wants to expose. You must specify the **Memory Type** when allocating memory. It holds specific requirements for the memory blob like visible to the host, coherent (between CPU and GPU) and cached. There may be an arbitrary combination of these, depending on the device driver.

Device can be thought of as a logical device, or opened device. It is the main object that represents an initialized Vulkan device that is ready to create all other objects. This is a similar concept to the Device object in DirectX®. During device creation, you need to specify which features you want to enable, and some of them are fundamental like anisotropic texture filtering. You also must state all queues that will be in use, their number and their Queue Families.

Queue is an object representing a queue of commands to be executed on the device. All the actual work to be done by the GPU is requested by filling CommandBuffers and submitting them to Queues, using the function vkQueueSubmit . If you have multiple queues like the main graphics queue and a compute queue, you can submit different CommandBuffers to each of them. This way you can enable asynchronous compute, which can lead to a substantial speed up if done right.

CommandPool is a simple object that is used to allocate CommandBuffers. It's connected to a specific Queue Family.

CommandBuffer is allocated from a specific CommandPool. It represents a buffer of various commands to be executed by a logical Device. You can call various functions on a command buffer, all of them starting with vkCmd . They are used to specify the order, type and parameters of tasks that should be performed when the CommandBuffer is submitted to a Queue and is finally consumed by the Device.

Sampler is not bound to any specific Image. It is rather just a set of state parameters, like filtering mode (nearest or linear) or addressing mode (repeat, clamp-to-edge, clamp-to-border etc.).

Buffer and Image are two types of resources that occupy device memory. **Buffer** is the simpler one. It is a container for any binary data that just has its length, expressed in bytes. **Image**, on the other hand, represents a set of pixels. This is the object known in other graphics APIs as a texture. There are many more parameters needed to specify creation of an Image. It can be 1D, 2D or 3D, have various pixel formats (like R8G8B8A8_UNORM or R32_SFLOAT) and can also consist of many discrete images, because it can have multiple array layers or MIP levels (or both). Image is a separate object type because it doesn't necessarily consist of just a linear set of pixels that can be accessed directly. Images can have a different implementation-specific internal format (tiling and layout) managed by the graphics driver.

Creating a Buffer of certain length or an Image with specific dimensions doesn't automatically allocate memory for it. It is a 3-step process that must be manually performed by you. You can also choose to use our Vulkan Memory Allocator library which takes care of the allocation for you

1. Allocate DeviceMemory,
2. Create Buffer or Image,
3. Bind them together using function vkBindBufferMemory or vkBindImageMemory .

That's why you must also create a **DeviceMemory** object. It represents a block of memory allocated from a specific memory type (as supported by PhysicalDevice) with a specific length in bytes. You shouldn't allocate separate DeviceMemory for each Buffer or Image. Instead, you should allocate bigger chunks of memory and assign parts of them to your Buffers and Images. Allocation is a costly operation and there is a limit on maximum number of allocations as well, all of which can be queried from your PhysicalDevice.

One exception to the obligation to allocate and bind DeviceMemory for every Image is the creation of a Swapchain. This is a concept used to present the final image on the screen or inside the window you're drawing into on your operating system. As such, the way of creating it is platform dependent. If you already have a window initialized using a system API, you first need to create a **SurfaceKHR** object. It needs the Instance object, as well as some system-dependent parameters. For example, on Windows these are: instance handle (HINSTANCE) and window handle (HWND). You can imagine SurfaceKHR object as the Vulkan representation of a window.

From it you can create **SwapchainKHR**. This object requires a Device. It represents a set of images that can be presented on the Surface, e.g. using double- or triple-buffering. From the swapchain you can query it for the Images it contains. These images already have their backing memory allocated by the system.

Buffers and Images aren't always used directly in rendering. On top of them there is another layer, called views. You can think about them somewhat like views in databases –

sets of parameters that can be used to look at a set of underlying data in a desired way. **BufferView** is an object created based on a specific buffer. You can pass offset and range during creation to limit the view to only a subset of buffer data.
Similarly, **ImageView** is a set of parameters referring to a specific image. There you can interpret pixels as having some other (compatible) format, swizzle any components, and limit the view to a specific range of MIP levels or array layers.

The way shaders can access these resources (Buffers, Images and Samplers) is through descriptors. Descriptors don't exist on their own, but are always grouped in descriptor sets. But before you create a descriptor set, its layout must be specified by creating a **DescriptorSetLayout**, which behaves like a template for a descriptor set. For example, your shaders used by a rendering pass for drawing 3D geometry may expect:

Binding slot	Resource
0	One uniform buffer (cbuffer in D3D) available to the vertex shader stage.
1	Another uniform buffer available to the fragment shader stage.
2	A sampled image.
3	A sampler, also available to the fragment shader stage.

You also need to create a **DescriptorPool**. It's a simple object used to allocate descriptor sets. When creating a descriptor pool, you must specify the maximum number of descriptor sets and descriptors of different types that you are going to allocate from it.

Finally, you allocate a **DescriptorSet**. You need both DescriptorPool and DescriptorSetLayout to be able to do it. The DescriptorSet represents memory that holds actual descriptors, and it can be configured so that a descriptor points to specific Buffer, BufferView, Image or Sampler. You can do it by using function vkUpdateDescriptorSets .

Several DescriptorSets can be bound as active sets in a CommandBuffer to be used by rendering commands. To do this, use the function vkCmdBindDescriptorSets . This function requires another object as well – **PipelineLayout**, because there may be multiple DescriptorSets bound and Vulkan wants to know in advance how many and what types of them it should expect. PipelineLayout represents a configuration of the rendering pipeline in terms of what types of descriptor sets will be bound to the CommandBuffer. You create it from an array of DescriptorSetLayouts.

In other graphics APIs you can take the immediate mode approach and just render whatever comes next on your list. This is not possible in Vulkan. Instead, you need to plan

the rendering of your frame in advance and organize it into passes and subpasses. Subpasses are not separate objects, so we won't talk about them here, but they're an important part of the rendering system in Vulkan. Fortunately, you don't need to know all the details when preparing your workload. For example, you can specify the number of triangles to render on submission. The crucial part when defining a **RenderPass** in Vulkan is the number and formats of attachments that will be used in that pass.

Attachment is Vulkan's name for what you might otherwise know as a render target – an Image to be used as output from rendering. You don't point to specific Images here – you just describe their formats. For example, a simple rendering pass may use a color attachment with format R8G8B8A8_UNORM and a depth-stencil attachment with format D16_UNORM . You also specify whether your attachment should have its content preserved, discarded or cleared at the beginning of the pass.

Framebuffer (not to be confused with SwapchainKHR) represents a link to actual Images that can be used as attachments (render targets). You create a Framebuffer object by specifying the RenderPass and a set of ImageViews. Of course, their number and formats must match the specification of the RenderPass. Framebuffer is another layer on top of Images and basically groups these ImageViews together to be bound as attachments during rendering of a specific RenderPass. Whenever you begin rendering of a RenderPass, you call the function vkCmdBeginRenderPass and you also pass the Framebuffer to it.

Pipeline is the big one, as it composes most of the objects listed before. It represents the configuration of the whole pipeline and has a lot of parameters. One of them is PipelineLayout – it defines the layout of descriptors and push constants. There are two types of Pipelines – ComputePipeline and GraphicsPipeline. ComputePipeline is the simpler one, because all it supports is compute-only programs (sometimes called compute shaders). GraphicsPipeline is much more complex, because it encompasses all the parameters like vertex, fragment, geometry, compute and tessellation where applicable, plus things like vertex attributes, primitive topology, backface culling, and blending mode, to name just a few. All those parameters that used to be separate settings in much older graphics APIs (DirectX 9, OpenGL), were later grouped into a smaller number of state objects as the APIs progressed (DirectX 10 and 11) and must now be baked into a single big, immutable object with today's modern APIs like Vulkan. For each different set of parameters needed during rendering you must create a new Pipeline. You can then set it as the current active Pipeline in a CommandBuffer by calling the function vkCmdBindPipeline .

Shader compilation is a multi-stage process in Vulkan. First, Vulkan doesn't support any high-level shading language like GLSL or HLSL. Instead, Vulkan accepts an intermediate format called SPIR-V which any higher-level language can emit. A buffer filled with data in SPIR-V is used to create a **ShaderModule**. This object represents a piece of shader code, possibly in some partially compiled form, but it's not anything the GPU can execute yet. Only when creating the Pipeline for each shader stage you are going to use (vertex, tessellation control, tessellation evaluation, geometry, fragment, or compute) do you specify the ShaderModule plus the name of the entry point function (like "main").

There is also a helper object called **PipelineCache**, that can be used to speed up pipeline creation. It is a simple object that you can optionally pass in during Pipeline creation, but that really helps to improve performance via reduced memory usage, and the compilation time of your pipelines. The driver can use it internally to store some intermediate data, so that the creation of similar Pipelines could potentially be faster. You can also save and load the state of a PipelineCache object to a buffer of binary data, to save it on disk and use it the next time your application is executed. We recommend you use them!

Query is another type of object in Vulkan. It can be used to read back certain numeric values written by the GPU. There are different kinds of queries like Occlusion (telling you whether some pixels were rendered, i.e. they passed all of the pre- and post-shading tests and made it through to the frame) or Timestamp (a timestamp value from some GPU hardware counter). Query doesn't have its own type, because it always resides inside a QueryPool and is just represented by a uint32_t index. **QueryPool** can be created by specifying type and number of queries to contain. They can then be used to issue commands to CommandBuffer, like vkCmdBeginQuery , vkCmdEndQuery , or vkCmdWriteTimestamp .

Finally, there are objects used for synchronization: **Fence**, **Semaphore** and **Event**.
A **Fence** signals to the host that the execution of a task has completed. It can be waited on, polled, and manually unsignaled on the host. It doesn't have its own command function, but it is passed when calling vkQueueSubmit . Once the submitted queue completes the according fence is signaled.

A **Semaphore** is created without configuration parameters. It can be used to control resource access across multiple queues. It can be signaled or waited on as part of command buffer submission, also with a call to vkQueueSubmit , and it can be signaled on one queue (e.g. compute) and waited on other (e.g. graphics).

An **Event** is also created without parameters. It can be waited on or signaled on the GPU as a separate command submitted to CommandBuffer, using the functions vkCmdSetEvent , vkCmdResetEvent , and vkCmdWaitEvents . It can also be set, reset and waited upon (via polling calls to vkGetEventStatus from one or more CPU threads. vkCmdPipelineBarrier can also be used for a similar purpose if synchronization occurs at a single point on the GPU, or subpass dependencies can be used within a render pass.

After that you just need to learn how to call Vulkan functions (possibly in parallel!) to make the GPU perform actual work every frame. Then you are on your way to fully utilizing the great, flexible computational power offered by modern GPUs.