Vulkan in 30 minutes

30 minutes not actually guaranteed.

I've written this post with a specific target audience in mind, namely those who have a good grounding in existing APIs (e.g. D3D11 and GL) and understand the concepts of multithreading, staging resources, synchronisation and so on but want to know specifically how they are implemented in Vulkan. So we end up with a whirlwind tour of what the main Vulkan concepts look like.

This isn't intended to be comprehensive (for that you should read the spec or a more in-depth tutorial), nor is it heavy in background or justification. Hopefully by the end of this you should be able to read specs or headers and have a sketched idea of how a simple Vulkan application is implemented, but you will need to do additional reading.

Mostly, this is the document I wish had already been written when I first encountered Vulkan - so for the most part it is tuned to what I would have wanted to know. I'll reference the spec whenever you should do more reading to get a precise understanding, but you'll at least know what to look for.

- baldurk (https://twitter.com/baldurk)

General

At the end of the post I've included a heavily abbreviated pseudocode program showing the rough steps to a hello world triangle, to match up to the explanations.

A few simple things that don't fit any of the other sections:

- Vulkan is a C API, i.e. free function entry points. This is the same as GL.
- The API is quite heavily typed - unlike GL. Each enum is separate, handles that are returned are opaque 64-bit handles so they are typed on 64-bit (not typed on 32-bit, although you can make them typed if you use C++).
- A lot of functions (most, even) take extensible structures as parameters instead of basic types.
- `VkAllocationCallbacks *` is passed into creation/destruction functions that lets you pass custom malloc/free functions for CPU memory. For more details read the spec, in simple applications you can just pass `NULL` and let the implementation do its own CPU-side allocation.

Warning: I'm not considering any error handling, nor do I talk much about querying for implementation limits and respecting them. While I'm not intentionally getting anything outright wrong, I am skipping over *many* details that a real application needs to respect. This post is just to get a grasp of the API, it's not a tutorial!

First steps

You initialise Vulkan by creating an instance (`VkInstance`). The instance is an entirely isolated silo of Vulkan - instances do not know about each other in any way. At this point you specify some simple information including which layers and extensions you want to activate - there are query functions that let you enumerate what layers and extensions are available.

With a `VkInstance` , you can now examine the GPUs available. A given Vulkan implementation might not be running on a GPU, but let's keep things simple. Each GPU gives you a handle - `VkPhysicalDevice` . You can query the GPUs names, properties, capabilities, etc. For example see `vkGetPhysicalDeviceProperties` and `vkGetPhysicalDeviceFeatures` .

With a VkPhysicalDevice, you can create a `VkDevice`. The `VkDevice` is your main handle and it represents a logical connection - i.e. 'I am running Vulkan on this GPU'. `VkDevice` is used for pretty much everything else. This is the equivalent of a GL context or D3D11 device.

> N.B. Each of these is a 1:many relationship. A `VkInstance` can have many `VkPhysicalDevices`, a `VkPhysicalDevice` can have many `VkDevices`. In Vulkan 1.0, there is no cross-GPU activity, but you can bet this will come in the future though.

I'm hand waving some book-keeping details, Vulkan in general is quite lengthy in setup due to its explicit nature and this is a summary not an implementation guide. The overall picture is that your initialisation mostly looks like `vkCreateInstance() → vkEnumeratePhysicalDevices() → vkCreateDevice()`. For a quick and dirty hello world triangle program, you can do just that and pick the first physical device, then come back to it once you want error reporting & validation, enabling optional device features, etc.

Images and Buffers

Now that we have a `VkDevice` we can start creating pretty much every other resource type (a few have further dependencies on other objects), for example `VkImage` and `VkBuffer`.

For GL people, one kind of new concept is that you must declare at creation time how an image will be used. You provide a bit field, with each bit indicating a certain type of usage - color attachment, or sampled image in shader, or image load/store, etc.

You also specify the tiling for the image - `LINEAR` or `OPTIMAL`. This specifies the tiling/swizzling layout for the image data in memory. `OPTIMAL` tiled images are opaquely tiled, `LINEAR` are laid out just as you expect. This affects whether the image data is directly readable/writable, as well as format support - drivers report image support in terms of 'what image types are supported in `OPTIMAL` tiling, and what image types are supported in `LINEAR`'. Be prepared for *very* limited `LINEAR` support.

Buffers are similar and more straightforward, you give them a size and a usage and that's about it.

> Images aren't used directly, so you will have to create a `VkImageView` - this is familiar to D3D11 people. Unlike GL texture views, image views are mandatory but are the same idea - a description of what array slices or mip levels are visible to wherever the image view is used, and optionally a different (but compatible) format (like aliasing a `UNORM` texture as `UINT`).
>
> Buffers are usually used directly as they're just a block of memory, but if you want to use them as a texel buffer in a shader, you need to provide a `VkBufferView`.

Allocating GPU Memory

Those buffers and images can't be used immediately after creation as no memory has been allocated for them. This step is up to you.

Available memory is exposed to applications by the `vkGetPhysicalDeviceMemoryProperties()`. It reports one or more memory *heaps* of given sizes, and one or more memory *types* with given properties. Each memory type comes from one heap - so a typical example for a discrete GPU on a PC would be two heaps - one for system RAM, and one for GPU RAM, and multiple memory types from each.

The memory types have different properties. Some will be CPU visible or not, coherent between GPU and CPU access, cached or uncached, etc. You can find out all of these properties by querying from the physical device. This allows you to choose the memory type you want. E.g. staging resources will need to be in host visible memory, but your images you render to will want to be in device local memory for optimal use. However there is an additional restriction on memory selection that we'll get to in the next section.

To allocate memory you call `vkAllocateMemory()` which requires your `VkDevice` handle and a description structure. The structure dictates which type of memory to allocate from which heap and how much to allocate, and returns a `VkDeviceMemory` handle.

Host visible memory can be mapped for update - `vkMapMemory()` / `vkUnmapMemory()` are familiar functions. All maps are by definition persistent, and as long as you synchronise it's legal to have memory mapped while in use by the GPU.

GL people will be familiar with the concept, but to explain for D3D11 people - the pointers returned by `vkMapMemory()` can be held and even written to by the CPU while the GPU is using them. These 'persistent' maps are perfectly valid as long as you obey the rules and make sure to synchronise access so that the CPU isn't writing to parts of the memory allocation that the GPU is using (see later).

> This is a little outside the scope of this guide but I'm going to mention it any chance I get - for the purposes of debugging, persistent maps of *non-coherent* memory with explicit region flushes will be much more efficient/fast than coherent memory. The reason being that for coherent memory the debugger must jump through hoops to detect and track changes, but the explicit flushes of non-coherent memory provide nice markup of modifications.
>
> In RenderDoc to help out with this, if you flush a memory region then the tool assumes you will flush for every write, and turns off the expensive hoop-jumping to track coherent memory. That way even if the only memory available is coherent, then you can get efficient debugging.

Binding Memory

Each `VkBuffer` or `VkImage` , depending on its properties like usage flags and tiling mode (remember that one?) will report their memory requirements to you via `vkGetBufferMemoryRequirements` or `vkGetImageMemoryRequirements` .

The reported size requirement will account for padding for alignment between mips, hidden meta-data, and anything else needed for the total allocation. The requirements also include a bitmask of the memory types that are compatible with this particular resource. The obvious restrictions kick in here: that `OPTIMAL` tiling color attachment image will report that only `DEVICE_LOCAL` memory types are compatible, and it will be invalid to try to bind some `HOST_VISIBLE` memory.

The memory type requirements generally won't vary if you have the same kind of image or buffer. For example if you know that optimally tiled images can go in memory type 3, you can allocate all of them from the same place. You will only have to check the size and alignment requirements per-image. Read the spec for the exact guarantee here!

> Note the memory allocation is by no means 1:1. You can allocate a large amount of memory and as long as you obey the above restrictions you can place several images or buffers in it at different offsets. The requirements include an alignment if you are placing the resource at a non-zero offset. In fact you will definitely want to do this in any real application, as there are limits on the total number of allocations allowed.
>
> There is an additional alignment requirement `bufferImageGranularity` - a minimum separation required between memory used for a `VkImage` and memory used for a `VkBuffer` in the same `VkDeviceMemory` . Read the spec for more details, but this mostly boils down to an effective page size, and requirement that each page is only used for one type of resource.

Once you have the right memory type and size and alignment, you can bind it with `vkBindBufferMemory` or `vkBindImageMemory` . This binding is **immutable**, and must happen before you start using the buffer or image.

Command buffers and submission

Work is explicitly recorded to and submitted from a `VkCommandBuffer` .

A `VkCommandBuffer` isn't created directly, it is allocated from a `VkCommandPool` . This allows for better threading behaviour since command buffers and command pools must be externally synchronised (see later). You can have a pool per thread and `vkAllocateCommandBuffers()` / `vkFreeCommandBuffers()` command buffers from it without heavy locking.

Once you have a `VkCommandBuffer` you begin recording, issue all your GPU commands into it **hand waving goes here** and end recording.

Command buffers are submitted to a `VkQueue` . The notion of queues are how work becomes serialised to be passed to the GPU. A `VkPhysicalDevice` (remember way back? The GPU handle) can report a number of *queue families* with different capabilities. e.g. a graphics queue family and a compute-only queue family. When you create your device you ask for a certain number of queues from each family, and then you can enumerate them from the device after creation with `vkGetDeviceQueue()` .

I'm going to focus on having just a single do-everything `VkQueue` as the simple case, since multiple queues must be synchronised against each other as they can run out of order or in parallel to each other. Be aware that some implementations might require you to use a separate queue for swapchain presentation - I think chances are that most won't, but you have to account for this. Again, read the spec for details!

You can `vkQueueSubmit()` several command buffers at once to the queue and they will be executed in turn. Nominally this defines the order of execution but remember that Vulkan has very specific ordering guarantees - mostly about what work can overlap rather than wholesale rearrangement - so take care to read the spec to make sure you synchronise everything correctly.

Shaders and Pipeline State Objects

The reasoning behind moving to monolithic PSOs is well trodden by now so I won't go over it.

A Vulkan `VkPipeline` bakes in a lot of state, but allows specific parts of the fixed function pipeline to be set dynamically: Things like viewport, stencil masks and refs, blend constants, etc. A full list as ever is in the spec. When you call `vkCreateGraphicsPipelines()` , you choose which states will be dynamic, and the others are taken from values specified in the PSO creation info.

You can optionally specify a `VkPipelineCache` at creation time. This allows you to compile a whole bunch of pipelines and then call `vkGetPipelineCacheData()` to save the blob of data to disk. Next time you can prepopulate the cache to save on PSO creation time. The expected caveats apply - there is versioning to be aware of so you can't load out of date or incorrect caches.

Shaders are specified as SPIR-V. This has already been discussed much better elsewhere, so I will just say that you create a `VkShaderModule` from a SPIR-V module, which could contain several entry points, and at pipeline creation time you chose one particular entry point.

The easiest way to get some SPIR-V for testing is with the reference compiler glslang (https://github.com/KhronosGroup/glslang), but other front-ends are available, as well as LLVM → SPIR-V support.

Binding Model

To establish a point of reference, let's roughly outline D3D11's binding model. GL's is quite similar.

- Each shader stage has its own namespace, so pixel shader texture binding 0 is not vertex shader texture binding 0.
- Each resource *type* is namespaced apart, so constant buffer binding 0 is definitely not the same as texture binding 0.

- Resources are individually bound and unbound to slots (or at best in contiguous batches).

In Vulkan, the base binding unit is a *descriptor*. A descriptor is an opaque representation that stores 'one bind'. This could be an image, a sampler, a uniform/constant buffer, etc. It could also be arrayed - so you can have an array of images that can be different sizes etc, as long as they are all 2D floating point images.

Descriptors aren't bound individually, they are bound in blocks in a `VkDescriptorSet` which each have a particular `VkDescriptorSetLayout`. The `VkDescriptorSetLayout` describes the types of the individual bindings in each `VkDescriptorSet`.

The easiest way I find to think about this is consider `VkDescriptorSetLayout` as being like a C struct type - it describes some members, each member having an opaque type (constant buffer, load/store image, etc). The `VkDescriptorSet` is a specific instance of that type - and each member in the `VkDescriptorSet` is a binding you can update with whichever resource you want it to contain.

This is roughly how you create the objects too. You pass a list of the types, array sizes and bindings to Vulkan to create a `VkDescriptorSetLayout`, then you can allocate `VkDescriptorSets` with that layout from a `VkDescriptorPool`. The pool acts the same way as `VkCommandPool`, to let you allocate descriptors on different threads more efficiently by having a pool per thread.

```
VkDescriptorSetLayoutBinding bindings[] = {
    // binding 0 is a UBO, array size 1, visible to all stages
    { 0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 1, VK_SHADER_STAGE_ALL_GRAPHICS, NULL },
    // binding 1 is a sampler, array size 1, visible to all stages
    { 1, VK_DESCRIPTOR_TYPE_SAMPLER,        1, VK_SHADER_STAGE_ALL_GRAPHICS, NULL },
    // binding 5 is an image, array size 10, visible only to fragment shader
    { 5, VK_DESCRIPTOR_TYPE_SAMPLED_IMAGE, 10, VK_SHADER_STAGE_FRAGMENT_BIT, NULL },
};
```

Example C++ outlining creation of a descriptor set layout

Once you have a descriptor set, you can update it directly to put specific values in the bindings, and also copy between different descriptor sets.

When creating a pipeline, you specify N `VkDescriptorSetLayouts` for use in a `VkPipelineLayout`. Then when binding, you have to bind matching `VkDescriptorSets` of those layouts. The sets can update and be bound at different frequencies, which allows grouping all resources by frequency of update.

To extend the above analogy, this defines the pipeline as something like a function, and it can take some number of structs as arguments. When creating the pipeline you declare the types (`VkDescriptorSetLayouts`) of each argument, and when binding the pipeline you pass specific instances of those types (`VkDescriptorSets`).

The other side of the equation is fairly simple - instead of having shader or type namespaced bindings in your shader code, each resource in the shader simply says which descriptor set and binding it pulls from. This matches the descriptor set layout you created.

```
#version 430

layout(set = 0, binding = 0) uniform MyUniformBufferType {
// ...
} MyUniformBufferInstance;

// note in the C++ sample above, this is just a sampler - not a combined image+sampler
// as is typical in GL.
layout(set = 0, binding = 1) sampler MySampler;

layout(set = 0, binding = 5) uniform image2D MyImages[10];
```

Example GLSL showing bindings

Synchronisation

I'm going to hand wave a lot in this section because the specific things you need to synchronise get complicated and long-winded fast, and I'm just going to focus on what synchronisation is available and leave the details of *what* you need to synchronise to reading of specs or more in-depth documents.

This is probably the hardest part of Vulkan to get right, especially since missing synchronisation might not necessarily break anything when you run it!

Several types of objects must be 'externally synchronised'. In fact I've used that phrase before in this post. The meaning is basically that if you try to use the same `VkQueue` on two different threads, there's no internal locking so it will crash - it's up to you to 'externally synchronise' access to that `VkQueue` .

For the exact requirements of what objects must be externally synchronised when you should check the spec, but as a rule you can use `VkDevice` for creation functions freely - it is locked for allocation sake - but things like recording and submitting commands must be synchronised.

N.B. There is no explicit or implicit ref counting of any object - you can't destroy anything until you are sure it is never going to be used again by either the CPU or the GPU.

Vulkan has `VkEvent` , `VkSemaphore` and `VkFence` which can be used for efficient CPU-GPU and GPU-GPU synchronisation. They work as you expect so you can look up the precise use etc yourself, but there are no surprises here. Be careful that you do use synchronisation though, as there are few ordering guarantees in the spec itself.

Pipeline barriers are a new concept, that are used in general terms for ensuring ordering of GPU-side operations where necessary, for example ensuring that results from one operation are complete before another operation starts, or that all work of one type finishes on a resource before it's used for work of another type.

There are three types of barrier - `VkMemoryBarrier` , `VkBufferMemoryBarrier` and `VkImageMemoryBarrier` . A `VkMemoryBarrier` applies to memory globally, and the other two apply to specific resources (and subsections of those resources).

The barrier takes a bit field of different memory access types to specify what operations on each side of the barrier should be synchronised against the other. A simple example of this would be "this `VkImageMemoryBarrier` has `srcAccessMask = ACCESS_COLOR_ATTACHMENT_WRITE` and `dstAccessMask = ACCESS_SHADER_READ` ", which indicates that all color writes should finish before any shader reads begin - without this barrier in place, you could read stale data.

Image layouts

Image barriers have one additional property - images exist in states called *image layouts*. `VkImageMemoryBarrier` can specify a transition from one layout to another. The layout must match how the image is used at any time. There is a `GENERAL` layout which is legal to use for anything but might not be optimal, and there are optimal layouts for color attachment, depth attachment, shader sampling, etc.

Images begin in either the `UNDEFINED` or `PREINITIALIZED` state (you can choose). The latter is useful for populating an image with data before use, as the `UNDEFINED` layout has undefined contents - a transition from `UNDEFINED` to `GENERAL` may lose the contents, but `PREINITIALIZED` to `GENERAL` won't. Neither initial layout is valid for use by the GPU, so at minimum after creation an image needs to be transitioned into some appropriate state.

Usually you have to specify the previous and new layouts accurately, but it is always valid to transition from `UNDEFINED` to another layout. This basically means 'I don't care what the image was like before, throw it away and use it like this'.

Render passes

A `VkRenderpass` is Vulkan's way of more explicitly denoting how your rendering happens, rather than letting you render into then sample images at will. More information about how the frame is structured will aid everyone, but primarily this is to aid tile based renderers so that they have a direct notion of where rendering on a given target happens and what dependencies there are between passes, to avoid leaving tile memory as much as possible.

N.B. Because I primarily work on desktops (and for brevity & simplicity) I'm not mentioning a couple of optional things you can do that aren't commonly suited to desktop GPUs like input and transient attachments. As always, read the spec :).

The first building block is a `VkFramebuffer` , which is a set of `VkImageViews` . This is **not** necessarily the same as the classic idea of a framebuffer as the particular images you are rendering to at any given point, as it can contain potentially more images than you ever render to at once.

A `VkRenderPass` consists of a series of *subpasses*. In your simple triangle case and possibly in many other cases, this will just be one subpass. For now, let's just consider that case. The subpass selects some of the framebuffer attachments as color attachments and maybe one as a depth-stencil attachment. If you have multiple subpasses, this is where you might have different subsets used in each subpass - sometimes as output and sometimes as input.

Drawing commands can only happen inside a `VkRenderPass` , and some commands such as copies clears can only happen **outside** a `VkRenderPass` . Some commands such as state binding can happen inside or outside at will. Consult the spec to see which commands are which.

Subpasses do not inherit state at all, so each time you start a `VkRenderPass` or move to a new subpass you have to bind/set all of the state. Subpasses also specify an action both for loading and storing each attachment. This allows you to say 'the depth should be cleared to 1.0, but the color can be initialised to garbage for all I care - I'm going to fully overwrite the screen in this pass'. Again, this can provide useful optimisation information that the driver no longer has to guess.

The last consideration is compatibility between these different objects. When you create a `VkRenderPass` (and all of its subpasses) you don't reference anything else, but you do specify both the format and use of all attachments. Then when you create a `VkFramebuffer` you must choose a `VkRenderPass` that it will be used with. This doesn't have to be the exact instance that you will later use, but it does have to be compatible - the same number and format of attachments. Similarly when creating a `VkPipeline` you have to specify the `VkRenderPass` and subpass that it will be used with, again not having to be identical but required to be compatible.

There are more complexities to consider if you have multiple subpasses within your render pass, as you have to declare barriers and dependencies between them, and annotate which attachments must be used for what. Again, if you're looking into that read the spec.

Backbuffers and presentation

I'm only going to talk about this fairly briefly because not only is it platform-specific but it's fairly straightforward.

> Note that Vulkan exposes native window system integration via extensions, so you will have to request them explicitly when you create your `VkInstance` and `VkDevice`.

To start with, you create a `VkSurfaceKHR` from whatever native windowing information is needed.

Once you have a surface you can create a `VkSwapchainKHR` for that surface. You'll need to query for things like what formats are supported on that surface, how many backbuffers you can have in the chain, etc.

You can then obtain the actual images in the `VkSwapchainKHR` via `vkGetSwapchainImagesKHR()`. These are normal `VkImage` handles, but you don't control their creation or memory binding - that's all done for you. You will have to create an `VkImageView` each though.

When you want to render to one of the images in the swapchain, you can call `vkAcquireNextImageKHR()` that will return to you the index of the next image in the chain. You can render to it and then call `vkQueuePresentKHR()` with the same index to have it presented to the display.

There are many more subtleties and details if you want to get really optimal use out of the swapchain, but for the dead-simple hello world case, the above suffices.

Conclusion

Hopefully you're still with me after that rather break-neck pace.

As promised I've skipped a lot of details and skimmed over some complexities, for example I have completely failed to mention sparse resources support, primary and secondary command buffers, and I've probably missed some other cool things.

With any luck though you have the broad-strokes impression of how a simple Vulkan applications is put together, and you're in a better place to go look at some documentation and figure the rest out for yourself.

Any questions or comments, let me know on twitter (http://twitter.com/baldurk) or email (mailto:baldurk@baldurk.org). In particular if anything is actually wrong I will correct it, as I don't want to mislead with this document - just set up a basic understanding that can be expanded on with further reading.

Also just to plug myself a little, if you need a graphics debugger for Vulkan consider giving RenderDoc (https://github.com/baldurk/renderdoc) a try, and let me know if you have any problems.

Happy hacking!

Appendix: Sample Pseudocode

```
#include <vulkan/vulkan.h>

// Pseudocode of what an application looks like. I've omitted most creation structures,
// almost all synchronisation and all error checking. This is not a copy-paste guide!
void DoVulkanRendering()
{
  const char *extensionNames[] = { "VK_KHR_surface", "VK_KHR_win32_surface" };

  // future structs will not be detailed, but this one is for illustration.
  // Application info is optional (you can specify application/engine name and version)
  // Note we activate the WSI instance extensions, provided by the ICD to
  // allow us to create a surface (win32 is an example, there's also xcb/xlib/etc)
  VkInstanceCreateInfo instanceCreateInfo = {
    VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO, // VkStructureType sType;
    NULL,                                   // const void* pNext;

    0,                                      // VkInstanceCreateFlags flags;

    NULL,                                   // const VkApplicationInfo* pApplicationInfo;

    0,                                      // uint32_t enabledLayerNameCount;
    NULL,                                   // const char* const* ppEnabledLayerNames;

    2,                                      // uint32_t enabledExtensionNameCount;
    extensionNames,                         // const char* const* ppEnabledExtensionNames;
  };

  VkInstance inst;
  vkCreateInstance(&instanceCreateInfo, NULL, &inst);

  // The enumeration pattern SHOULD be to call with last parameter NULL to
  // get the count, then call again to get the handles. For brevity, omitted
  VkPhysicalDevice phys[4]; uint32_t physCount = 4;
  vkEnumeratePhysicalDevices(inst, &physCount, phys);

  VkDeviceCreateInfo deviceCreateInfo = {
    // I said I was going to start omitting things!
  };

  VkDevice dev;
  vkCreateDevice(phys[0], &deviceCreateInfo, NULL, &dev);

  // fetch vkCreateWin32SurfaceKHR extension function pointer via vkGetInstanceProcAddr
    VkWin32SurfaceCreateInfoKHR surfaceCreateInfo = {
        // HINSTANCE, HWND, etc
    };
  VkSurfaceKHR surf;
  vkCreateWin32SurfaceKHR(inst, &surfaceCreateInfo, NULL, &surf);

  VkSwapchainCreateInfoKHR swapCreateInfo = {
    // surf goes in here
  };
```

```
VkSwapchainKHR swap;
vkCreateSwapchainKHR(dev, &swapCreateInfo, NULL, &swap);

// Again this should be properly enumerated
VkImage images[4]; uint32_t swapCount;
vkGetSwapchainImagesKHR(dev, swap, &swapCount, images);

// Synchronisation is needed here!
uint32_t currentSwapImage;
vkAcquireNextImageKHR(dev, swap, UINT64_MAX, presentCompleteSemaphore, NULL, &currentSwapImage);

// pass appropriate creation info to create view of image
VkImageView backbufferView;
vkCreateImageView(dev, &backbufferViewCreateInfo, NULL, &backbufferView);

VkQueue queue;
vkGetDeviceQueue(dev, 0, 0, &queue);

VkRenderPassCreateInfo renderpassCreateInfo = {
  // here you will specify the total list of attachments
  // (which in this case is just one, that's e.g. R8G8B8A8_UNORM)
  // as well as describe a single subpass, using that attachment
  // for color and with no depth-stencil attachment
};

VkRenderPass renderpass;
vkCreateRenderPass(dev, &renderpassCreateInfo, NULL, &renderpass);

VkFramebufferCreateInfo framebufferCreateInfo = {
  // include backbufferView here to render to, and renderpass to be
  // compatible with.
};

VkFramebuffer framebuffer;
vkCreateFramebuffer(dev, &framebufferCreateInfo, NULL, &framebuffer);

VkDescriptorSetLayoutCreateInfo descSetLayoutCreateInfo = {
  // whatever we want to match our shader. e.g. Binding 0 = UBO for a simple
  // case with just a vertex shader UBO with transform data.
};

VkDescriptorSetLayout descSetLayout;
vkCreateDescriptorSetLayout(dev, &descSetLayoutCreateInfo, NULL, &descSetLayout);

VkPipelineCreateInfo pipeLayoutCreateInfo = {
  // one descriptor set, with layout descSetLayout
};

VkPipelineLayout pipeLayout;
vkCreatePipelineLayout(dev, &pipeLayoutCreateInfo, NULL, &pipeLayout);

// upload the SPIR-V shaders
VkShaderModule vertModule, fragModule;
```

```
vkCreateShaderModule(dev, &vertModuleInfoWithSPIRV, NULL, &vertModule);
vkCreateShaderModule(dev, &fragModuleInfoWithSPIRV, NULL, &fragModule);

VkGraphicsPipelineCreateInfo pipeCreateInfo = {
  // there are a LOT of sub-structures under here to fully specify
  // the PSO state. It will reference vertModule, fragModule and pipeLayout
  // as well as renderpass for compatibility
};

VkPipeline pipeline;
vkCreateGraphicsPipelines(dev, NULL, 1, &pipeCreateInfo, NULL, &pipeline);

VkDescriptorPoolCreateInfo descPoolCreateInfo = {
  // the creation info states how many descriptor sets are in this pool
};

VkDescriptorPool descPool;
vkCreateDescriptorPool(dev, &descPoolCreateInfo, NULL, &descPool);

VkDescriptorSetAllocateInfo descAllocInfo = {
  // from pool descPool, with layout descSetLayout
};

VkDescriptorSet descSet;
vkAllocateDescriptorSets(dev, &descAllocInfo, &descSet);

VkBufferCreateInfo bufferCreateInfo = {
  // buffer for uniform usage, of appropriate size
};

VkMemoryAllocateInfo memAllocInfo = {
  // skipping querying for memory requirements. Let's assume the buffer
  // can be placed in host visible memory.
};
VkBuffer buffer;
VkDeviceMemory memory;
vkCreateBuffer(dev, &bufferCreateInfo, NULL, &buffer);
vkAllocateMemory(dev, &memAllocInfo, NULL, &memory);
vkBindBufferMemory(dev, buffer, memory, 0);

void *data = NULL;
vkMapMemory(dev, memory, 0, VK_WHOLE_SIZE, 0, &data);
// fill data pointer with lovely transform goodness
vkUnmapMemory(dev, memory);

VkWriteDescriptorSet descriptorWrite = {
  // write the details of our UBO buffer into binding 0
};

vkUpdateDescriptorSets(dev, 1, &descriptorWrite, 0, NULL);

// finally we can render something!
// ...
```

```
  // Almost.

  VkCommandPoolCreateInfo commandPoolCreateInfo = {
    // nothing interesting
  };

  VkCommandPool commandPool;
  vkCreateCommandPool(dev, &commandPoolCreateInfo, NULL, &commandPool);

  VkCommandBufferAllocateInfo commandAllocInfo = {
    // allocate from commandPool
  };
  VkCommandBuffer cmd;
  vkAllocateCommandBuffers(dev, &commandAllocInfo, &cmd);

  // Now we can render!

  vkBeginCommandBuffer(cmd, &cmdBeginInfo);
  vkCmdBeginRenderPass(cmd, &renderpassBeginInfo, VK_SUBPASS_CONTENTS_INLINE);
  // bind the pipeline
  vkCmdBindPipeline(cmd, VK_PIPELINE_BIND_POINT_GRAPHICS, pipeline);
  // bind the descriptor set
  vkCmdBindDescriptorSets(cmd, VK_PIPELINE_BIND_POINT_GRAPHICS,
                          descSetLayout, 1, &descSet, 0, NULL);
  // set the viewport
  vkCmdSetViewport(cmd, 1, &viewport);
  // draw the triangle
  vkCmdDraw(cmd, 3, 1, 0, 0);
  vkCmdEndRenderPass(cmd);
  vkEndCommandBuffer(cmd);

  VkSubmitInfo submitInfo = {
    // this contains a reference to the above cmd to submit
  };

  vkQueueSubmit(queue, 1, &submitInfo, NULL);

  // now we can present
  VkPresentInfoKHR presentInfo = {
    // swap and currentSwapImage are used here
  };
  vkQueuePresentKHR(queue, &presentInfo);

  // Wait for everything to be done, and destroy objects
}
```