Vulkan Notes

Joseph Grimsic

November 2025

Vulkan Questions

- what are the things that specifically separates OpenGL and Vulkan?
- What is sType?
- What is a command pool?
- How can we abstract commands/ should we?
- Do we need to work with commands regularly to parallelize our game engine?

What If I Use Sascha as a starting point

- How would I work with or change this program to write a game engine?
- What abstractions would I want to make?
- What if I am a solo dev—does that change how I should abstract things?
- How can I black box the lesser parts of Vulkan I don't need to know that deeply?
- What levels of abstraction will I usually be using?

Vulkan Resources

- Vulkan Triangle Example
- Vulkan Helper?
- Interesting opinions on Vulkan
- Vulkan Tutorial By Khronos Group
- vkguide.dev

Sascha Willems Helpers

`*_CreateInfo and *_Info`

I don't know what Sascha's functions are entirely for—except that they attempt to reduce Vulkan boilerplate. However, it seems that every function returns two kinds of pointers:

```
*_CreateInfo
*_Info
```

We are using some kind of struct—but I'm not sure what the struct is for. The struct probably holds whatever these two kinds of pointers point to?

So actually, the `*` in `*_CreateInfo` and `*_Info` represent the wildcard operator, and is just to show the naming convention in the documentation. These names refer to structs that have state information for Vulkan.

Commands

Commands are the main part of Vulkan. Commands are how we talk directly to the GPU.

Commands are normally stored in a **command buffer**, so we can batch them together. We usually call the act of intializing a command buffer **recording** the command buffer. In the Vulkan workflow, we will have record a command buffer using a **command pool**. We will have one command pool for every **queue family**.

Sascha Repository Notes

Git does not automatically download submodules

To update submodules:

```
git submodule update --init --recursive
```

Then you can run the binaries, for example:

```
./build/bin/bloom
```

or:

```
cd build/bin/
./bloom
```

Reading Through The Vulkan Tutorial

Here is a high level of how I would currently describe the Vulkan setup:

1. create instance
2. device selection
3. create window
4. then some other stuff

here is my second attempt at that

1. create instance + device selection
2. queue families stuff
3. create window + swapchain
4. framebuffer stuff
5. probably something else

we are getting closer, but honestly maybe it's not that important. i am considering reading throught the vulkan tutorial but i am unsure as when to just jump in. i could probably save some time in the future by reading a little bit now but i suspect i will not retain that much information. i guess i will take some notes.

first i will skip to the hello triangle in their example and think of some questions i have..

some things that stand out to me:

- item (will i finish?)

Going Through Sascha's Repository

Idea

I am thinking this will be faster to learn as Sascha provides a standard hello triangle program (meant to be as vanilla as possible) and shows his evolution (you can see comments in his READ.md) of abstractions. This will likely make the most sense to study because we know that Sascha has a very popular repo so we can assume others think it is good and that it is a valid idea for me to pursue.

Notes On Sascha

Let's start with is Sascha's claimed vanilla hello triangle similar to the one in the khronos vulkan tutorial?

Questions From Sascha's Repo:

- what is a descriptor set in vulkan?
- what is the graphics queue?
- what is a vk semaphore?
- what is vk device?
- what is vk null handle?
- what is vk descriptor set?
- what is vk null handle?
- what is vk pipeline layout?
- how to read/write to a glm::mat4?
- what is vk descriptor set layout?

- how do we interact with vk command pool?
- what does this do?

```
VulkanExample() : VulkanExampleBase() {
```

Theory

Descriptor Set

Descriptor sets are sets of pointers to resources that shaders want access to. In OpenGL, you might 'bind' one buffer at a time. Vulkan expects you to bind a descriptor set all together.

Similar to **command pools** there are **descriptor pools**.

Descriptor Pools

To allocate a descriptor set, we first intialize a descriptor pool. When setting up a descriptor pool, we tell vulkan there will be 'x' number of sets with 'y' number of descriptors

Fences

A fence is used to sync the CPU and GPU. I don't know how it does this. A fence can be **signaled** by the GPU, so that the CPU knows that the GPU has finished drawing the frame.

Semaphore

Similarly, a **semaphore** is used to sync different parts of the GPU. Some examples of semaphores are

- **renderSemaphore**
- **swapchainSemaphore**

There is also a **complete semaphore**, but I don't know what that is for yet.

Projection Matrix

This has something to do with how objects look when they are further away, maybe it does some multiplication on the location of the vertices on screen coordinates?

This is close. It has to do with a matrix multiplication on the 3d vertices of an object, altering how it looks from far away. The projection matrix is part of the pipeline to get objects in 3d space to screen space coordinates.

Note: The lack of a projection matrix is called an orthographic projection.

View Model Matrix

This has something to do with the view frustum? Perhaps this is the matrix that we perform operations on to change the camera position and angle?

Depth Stencil Image

What is the depth stencil image? I mean stencils are like outlines of stuff so maybe depth stencil image draws the outlines of objects in the depth frame buffer? Is the Z buffer a separate frame buffer?

So a depth stencil image is actually a combination of buffers (both the depth and stencil buffers).

Stencil Buffer

Let's review what a stencil buffer is. It is true that a stencil buffer can be used for outlines, but more generally—stencil buffers can be thought of as masks. The concept of masking is where you can think of separating layers. So you may 'cutout' a layer by masking, so that you can see to the next layer.

Another way to think of this as a subtractive process. Here is an excerpt from a conversation with Gemini 2.5 Pro:

```
Pass 1: Draw the Object (to create the "cutout" mask).
        Turn off writing to the color and depth buffers.
        Turn on writing to the stencil buffer.
        Set the stencil operation to: "If a pixel passes the depth test,
        write the value 1 into the stencil buffer."
        Draw your mesh.

        Result: The screen looks unchanged, but the stencil buffer now
        contains a perfect silhouette (a "mask") of your object, filled
        with the number 1. Everything else is 0.

    Pass 2: Draw the Outline (the "big yellow" version).
        Turn on writing to the color buffer (so we can see the yellow).
        Turn off writing to the depth buffer (this prevents the outline
        from messing up future depth tests).
        Set the stencil test to: "Only draw a pixel if the stencil buffer value
        at that location is NOT 1 (i.e., it's 0)."
        Draw a slightly larger version of your mesh, all in yellow.

    The "Cutout" Happens:

        The GPU starts to draw the big yellow mesh.
        Where the original object was (the "inner layer"), the stencil buffer
        has a 1. The test (stencil_value != 1) fails, and the yellow pixel is
        discarded. This is your "cutout"!

        On the larger edges of the yellow mesh (the "thickness" you mentioned), the pixel falls out

Final Result: You are left with only the yellow outline.
```

Struct vs. Class

One thing that vkguide.dev has brought to my attention is when to use struct versus when to use a class. From my understanding, a struct is used for simple instances when we just need to store a group of data. A class can hold lots of things for example state, functionality, and data too.

Vulkan 'Views' + Reading Code Snippet

When reading through Sascha's examples I came across this segment from triangle.cpp:

```
// Create a view for the depth stencil image
// Images aren't directly accessed in Vulkan, but rather through views
// described by a subresource range This allows for multiple views of one
// image with differing ranges (e.g. for different layers)
VkImageViewCreateInfo depthStencilViewCI{};
depthStencilViewCI.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
depthStencilViewCI.viewType = VK_IMAGE_VIEW_TYPE_2D;
depthStencilViewCI.format = depthFormat;
depthStencilViewCI.subresourceRange = {};
depthStencilViewCI.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;
```

What is a view? It seems that we first initialize the view and then there are several instance variables we set for it. This code snippet reveals many different questions:

- What is a **subresource**?
- What is an **aspect**?

- What what does ’CI’ mean in ’depthStencilViewCI’?

I also have some more questions as a result from this snippet:

- What kinds of things do we need a view for?
- Does Sascha abstract some of this ’state stuff’ away when intializing a view?
- What does an all caps variable like ’VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO’— mean again? Is it a global constant?
- —and finally, how much of this is important to me as an engine developer?

VK_NULL_HANDLE

Weirdly, the VK_NULL_HANDLE is an integer, and not a null pointer or something like that. We use VK_NULL_HANDLE— when we are declaring some Vulkan object without giving it a value.

Ticket Number

What is a **ticket number**? What does it mean if there is an **empty ticket number**? What is a **ticket slot**, and what does it mean for it to be empty (0)?

How is initialization abstracted with Sascha’s Framework?

The following is a snippet from a conversation with Gemini 2.5 Pro:

```
Before: Raw Vulkan (What we discussed)

You have to manually zero-initialize every struct and set every single
member, including the sType.

// 1. Define the binding
VkDescriptorSetLayoutBinding uboBinding{};
uboBinding.sType = ... // Easy to forget! Oh wait, this one doesn’t have sType.
uboBinding.binding = 0;
uboBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
uboBinding.descriptorCount = 1;
uboBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
uboBinding.pImmutableSamplers = nullptr;

// 2. Define the create info
VkDescriptorSetLayoutCreateInfo layoutInfo{};
```

```
layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
layoutInfo.pNext = nullptr;
layoutInfo.flags = 0;
layoutInfo.bindingCount = 1;
layoutInfo.pBindings = &uboBinding;

After: With Sascha Willems' vks::initializers

He provides helper functions that act as "constructors" for these C structs.
They take the important parameters and set all the other members (sType, pNext,
flags, etc.) to correct default values.

// 1. Define the binding (one line)
VkDescriptorSetLayoutBinding uboBinding =
    vks::initializers::descriptorSetLayoutBinding(
        VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,
        VK_SHADER_STAGE_VERTEX_BIT,
        0); // type, stage, binding

// 2. Define the create info (one line)
VkDescriptorSetLayoutCreateInfo layoutInfo =
    vks::initializers::descriptorSetLayoutCreateInfo(
        &uboBinding,
        1); // pBindings, bindingCount
```

This example may prove useful when learning Sascha's framework.

Appendix

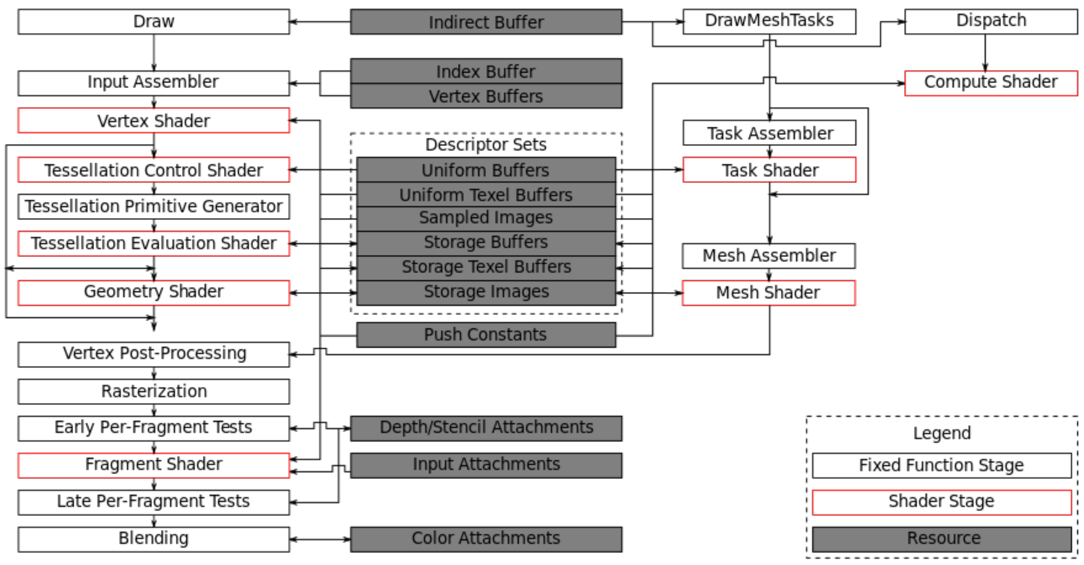

Figure 1: Vulkan Pipeline (Why is it so low res :())

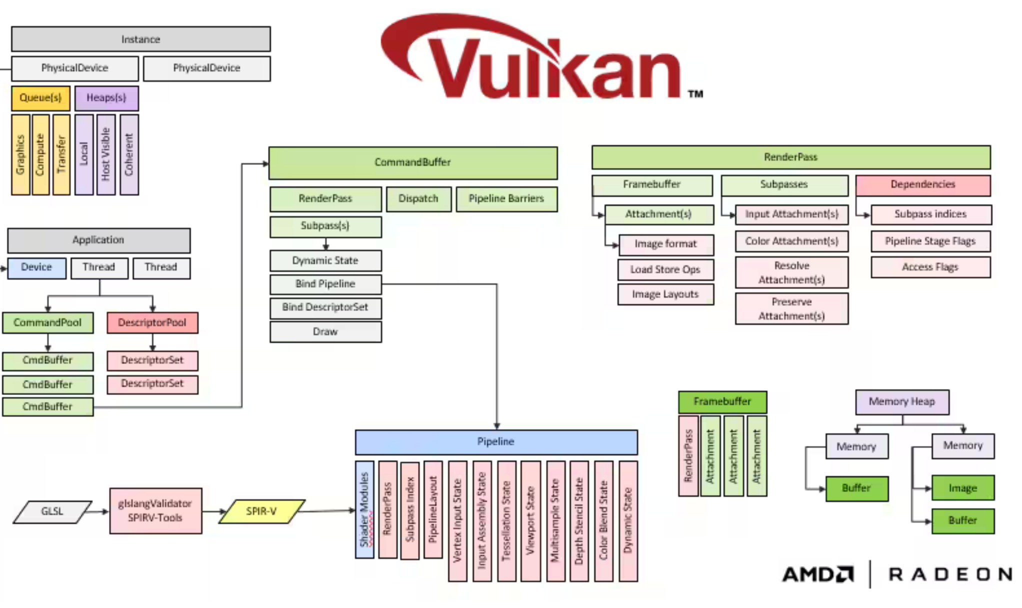

Figure 2: Vulkan Pipeline (Why is it so low res :())