Vulkan Notes

Joseph Grimsic

November 2025

Contents

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
```

The following is an example clean build script specific to Wayland:

```
#!/usr/bin/env bash
set -e

BUILD_DIR="build"

echo ">>> Removing old build directory..."
rm -rf "$BUILD_DIR"

echo ">>> Creating new build directory..."
mkdir "$BUILD_DIR"
cd "$BUILD_DIR"

echo ">>> Running CMake configure..."
cmake .. \
  -DCMAKE_EXPORT_COMPILE_COMMANDS=ON \
  -DUSE_WAYLAND_WSI=OFF
```

```
echo ">>> Building..."
cmake --build . -- -j"$(nproc)"

# Optionally copy compile_commands.json to project root
if [ -f compile_commands.json ]; then
    echo ">>> Copying compile_commands.json to project root..."
    cp compile_commands.json ..
fi

echo ">>> Done."
```

Then you can run the binaries, for example:

```
./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 the graphics queue?
- what is vk pipeline layout?
- how to read/write to a glm::mat4?
- 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.
```

Reading Code Snippet And Some Resulting Questions

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?

What Does 'CI' Suffix Mean In Variable Names?

CI is actually shorthand for **Create Info**. This is because this variable is a **CreateInfo struct**, same as we mentioned earlier.

VK View

Images cannot be directly accessed in Vulkan, but instead with a View object. Hence the two Vulkan objects:

- VkImage
- VkImageView

How Does A View Relate To A Frame Buffer?

A frame buffer has views as attachments?

A **VkImage** is a collection of the image's raw data and overall properties such as:

- Dimension
- Format
- Mip Levels

Subresource

What is a subresource and how can it have a range?

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.

What is multisampling (MSAA?)

Multisampling (MSAA) is a method of anti aliasing. Aliasing happens when a single sample inside of the pixel does not detect the subpixel fragment, so the result of the pixel can become an artifact. Subpixel artifacting are called **jaggies**, which are a result of aliasing. To address this issue, we have multiple subpixel samples—hence the name.

Device vs. Physical Device

In Vulkan, there are two kinds of devices:

- The **physical device (VkPhysicalDevice)**
- And the **logical device (VkDevice)**

Most always in the context of Vulkan, we are talking about the logical device. The logical device is the object that you use to interface with the hardware. You initialize VkDevice with the tools/features that you are going to be using from the GPU.

High-level pseudocode: Basic indexed triangle (Vulkan 1.3)

This example initializes a minimal Vulkan application that renders a single indexed triangle using Vulkan 1.3 dynamic rendering and per-frame uniform buffers.

1. Create the application object and parse command-line arguments. Configure a simple look-at camera and request Vulkan API version `VK_API_VERSION_1_3`, enabling the `dynamicRendering` and `synchronization2` features.
2. Initialize Vulkan: create an instance, select a compatible physical device, create a logical device and graphics queue, and cache device properties and memory capabilities for later allocations.
3. Prepare global state and resources used across frames: determine swapchain color/depth formats and reserve per-frame arrays sized by `MAX_CONCURRENT_FRAMES` for in-flight resources.
4. Create synchronization primitives: one fence per in-flight frame (initially signaled) and semaphores to coordinate image acquisition and render completion.
5. Create one command pool and allocate one primary command buffer per in-flight frame.
6. Build vertex and index data for a triangle; compute buffer sizes and the index count.
7. Create a host-visible staging buffer, map it, and copy both vertex and index data into that single mapped region.
8. Create device-local vertex and index buffers (with transfer destination usage), allocate device-local memory for each, and bind them.
9. Record and submit a short command buffer that copies the ranges from the staging buffer into the device-local buffers; use a temporary fence to wait for completion, then free the staging buffer.
10. Define a uniform block type containing projection, view and model matrices and create one host-visible, coherent uniform buffer per in-flight frame; map them persistently for fast updates.

11. Create a descriptor pool and a descriptor set layout for a single uniform buffer binding; allocate and update one descriptor set per in-flight frame to point at the corresponding uniform buffer.
12. Create a pipeline layout using the descriptor set layout, then build the graphics pipeline: load SPIR-V shaders, configure vertex input bindings/attributes, input assembly, rasterization, depth/stencil, multisampling, color blend and dynamic viewport/scissor, and attach dynamic rendering formats.
13. Allocate and bind a device-local depth image and create an image view for depth/stencil usage.
14. In the render loop: wait on the current frame fence, acquire the next swapchain image (handling out-of-date/suboptimal by resizing), update the mapped uniform buffer for the current frame, reset and begin the command buffer.
15. Insert image memory barriers to transition color and depth images to attachment-optimal layouts, begin a dynamic rendering section, set viewport and scissor dynamically, bind pipeline and the current frame's descriptor set, bind vertex and index buffers, and issue a single indexed draw call; end rendering and transition the color image to present layout.
16. Submit the command buffer waiting on the present semaphore and signaling a render-complete semaphore; pass the per-frame fence for GPU completion tracking. Present the image using `vkQueuePresentKHR`, handling resizing if necessary, and advance the current frame index modulo `MAX_CONCURRENT_FRAMES`.
17. On shutdown, wait for device idle and clean up: destroy pipelines, pipeline layout, descriptor set layout and pool, buffers and their memories, image views and images, semaphores, fences, command pool and swapchain resources; delete the application object.

One thing to note:

After looking at the example **Raytracing Reflections**, I noticed that there are only 500 lines of code, and the output is a fully rendered scene with some reflections. This sort of thing gives me hope because the complexity was seeming cut in half (compared to 900 lines for a triangle). This seems like a decent starting point for a scene. There are still lots of points of confusion—even some that are purely about C++ for example the **::** operator not only being used for namespaces.

Families and Queues

When a Vulkan instance is created, one of the things it does before anything is rendered to the screen, is query the system's GPU (vkPhysicalDevice). The physical device will tell us important information; for example, the index of the compute family. This is because different GPU's have different index values for families.

Families A family is a group of **queues**. For example, there is the **graphics family**, which has one or more **graphics queue** objects that interface with the rendering hardware.

Graphics Queue The **graphics queue** is an object that you would send commands to. The graphics queue is responsible for the rendering, as opposed to the **compute queue**, which you might use for gameplay math.

Compute Queue But actually it is not just for 'gameplay math', but instead highly parallelizable math. Some examples are the folllwing:

- Physics Simulations
- Post Processing (Not apart of the graphics queue!?!)
- Culling!?

Transfer Queue

Transfer Queue The **transfer queue** is specialized for high-speed DMA (Direct Memory Access) transfers. We use this queue to move data from CPU-visible **staging buffers** into high-performance GPU memory. This includes bulk data such as textures, vertex buffers, and index buffers. By offloading these copy operations to the transfer queue, the graphics queue remains free to focus entirely on rendering, allowing for asynchronous execution.

What Are Queues For? Queues are objects that receive **commands**. The central idea behind Vulkan is that the GPU receives **command buffers** from the CPU program, and the GPU (physical device) executes said commands.

Why Not Submit All Tasks To One Of The Families? We seperate graphics family tasks and compute family tasks because say we were to put everything on the graphics queue; then, we would not be able to compute our graphics tasks in parallel with our compute tasks.

Slang

After looking at a Slang shader file, I notice things like there are global functions for dot() and sqrt(). What other global functions are there? What are some main concepts behind writing shaders?

Here is an example of collision detection in Slang:

```
// collide with boundary
if ((vPos.x < -1.0) || (vPos.x > 1.0) || (vPos.y < -1.0) || (vPos.y > 1.0)) {
    vVel = (-vVel * 0.1) + attraction(vPos, destPos) * 12;
} else {
    particlesOut[index].pos.xy = vPos;
}
```

What is this following segment for?

```
    // Write back
    particlesOut[index].vel.xy = vVel;
    particlesOut[index].gradientPos.x = gPos.x + 0.02 * ubo.deltaT;
    if (particlesOut[index].gradientPos.x > 1.0) {
        particlesOut[index].gradientPos.x -= 1.0;
    }
```

sType + pNext

sType and pNext are both core concepts in Vulkan. Vulkan is close to the hardware, and thus it is directly talking to your GPU's drivers. Your GPU drivers do not know how to get to the memory that you want them to.

Enter: the Vulkan linked-list structure. Vulkan keeps an important linked list, which you can add to by setting pNext. However, your driver also needs to know what pNext is pointing to (it is a void* after all). **sType** (short for struct type) tells the drivers what the struct is.

This Vulkan linked-list structure we are talking about is the **extension chain**. Why is the extension chain important?

So the extension chain (the idea that requires pNext and sType) is used once at setup, and then becomes obsolete. In other words: it sets parameters at the beginning of the program. This is called the **object creation** phase.

However, there is an *exception* to this. Every time we record a command buffer (every frame) we require pNext and sType. Why?

sType + pNext (as described by gemini 3)

sType and pNext are both core concepts in Vulkan. Vulkan is close to the hardware, and thus it is directly talking to your GPU's drivers.

The Problem: When you pass a generic pointer to the driver, the driver can access that memory, but it doesn't know how to interpret it. It doesn't know if the data at that address is a request to create a buffer, an image, or a device.

The Solution: The Vulkan extension chain. Vulkan allows you to build a linked list by setting **pNext**. To help the driver understand what it is looking at, **sType** (Structure Type) acts as an ID tag. It tells the driver exactly what struct type is sitting at that memory address.

This structure is primarily used in the Object Creation phase. You build the chain to set parameters, pass it to the driver to create an object (like a Device), and then the chain is obsolete.

However, we also encounter sType and pNext in the Render Loop. When recording command buffers each frame, we fill out temporary structures to describe the current frame's work. Just like in setup, the driver reads these structures once to translate them into GPU commands.

vkPipelineLayout

C++ Notes

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.

Constructor Definition Syntax

The following is a way to declare a new constructor:

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

More generally, it is written as:

```
constructorFromDerivedClass?() : NewConstructor?()
```

I am very confused—but maybe it is unimportant

How To Make A Class

In C++, to make a class we usually do the following:

1. Declare the class in a header (e.g. class.h)
2. Implement the class functionality in a source file (e.g. class.cpp)

How To Make A Struct

Weirdly, in C++ structs do not need keyword **typedef** when declared..

How To Make A Template Class?

A template has to do with implementing functionality across different primitive types? Like overloading function sum so that sum(int a, int b) and sum(double a, double b)? What makes templates different than classes? Do we use headers for them too?

:: Operator

I was previously aware that the **::** operator could be used to access members of a namespace—but now I am finding out that it can be used to call specific functions in an inheritance hierarchy. How does this work?

Update: The **::** operator is called the **scope resolution modifier**. The scope resolution modifier is not only used for namespace members, but another example is an enum class. Say you have an enum class defined with the following:

```
enum class ShadowQuality{
kNone,
kLow,
kMedium,
kHigh,
}
```

Const Correctness

I don't fully understand the idea of **const correctness**—but so far it seems like the idea of a read-only member variable. The current working example I am considering is the following:

We want our game engine to allow the user to choose settings based on the power of their machine. For example, the user may want to lower quality of shadows, so we can set shadows to the following states:

- High Quality
- Medium Quality
- Low Quality
- None

We can store this data in a struct:

```
struct {
int shadow_quality
} typedef GameConfig
```

What Is A Valid Starting Point For My Game Engine?

I am currently considering the example from Sascha Willems called **Raytracing Reflections**.

Glossary

Aliasing When **jaggies** misrepresent a pixel and cause discoloring.

Anti Aliasing Technique to combat **aliasing**, usually by some special way of sampling.

Application Object idk.

Attachment idk, i think they are apart of frame buffers?.

Attributes I think attributes are when we store data in vertices.

Binary Semaphore (Mutex) Has two states(0 or 1), used to lock a resource, and managed by an owner.

Blending **Blending** refers to creating a transparency effect. For example, you might use blending so that a UI is transparent so you can see the game scene behind it.

Color Attachment idk.

Command Pools Object used to allocate command buffers.

Compute Queue For example, could do math.

Const Correctness **const correctness** is the idea of making a variable read-only in certain program scopes, not allowing for unwanted writes. This is a key feature of C++ keyword **const**.

Counting Semaphore Allows up to N threads to access a resource simultanously.

Damped Dot idk.

Dependancy Injection idk.

Depth Image idk.

Descriptor Pool Obviously some kind of object to allocate descriptors or descriptor sets?.

Descriptor Set idk.

Device Like the object that keeps track of the rendering state of the Vulkan program?.

Device Idle Command to pause the CPU until the GPU is finished.

Device-Local Memory on the VRAM.

Dispatch Thread idk.

Dynamic Rendering This is the idea of removing the 'render pass' and 'framebuffer' objects. In other words, we move the configuration from initialization to recording time.

extension chain idk what this is + relates to stype and pnext??.

Fence Object responsible for keeping CPU and GPU synced.

Global Invocation ID idk.

Gradient Position idk.

Graphics Queue Draws stuff.

In-Flight Frame Idea of rendering current frame on GPU, and recording commands for next frame to be rendered, using CPU.

Instanced Mesh Rendering Taking an instance/object and rendering it multiple times. For example a tree with many instances of leaves that have different rotations and size (Tell the GPU: "Draw this mesh 500 times").

KHR Khronos or Khronos Group—can relate to offical extension (what is extension).

label Description.

Max Concurrent Frames Number of frames to be 'processed' simultaneously.

Memory Barriers Used to make sure that one GPU core is finished writing to memory before another core tries to read.

Mip idk.

Mip Map idk, can relate to level of distance optimization? (LOD).

Multisampling Antialiasing (MSAA) An implementation of **anti aliasing**.

Object Slicing idk, it is an idea with inheritance in cpp and copying objects by value.

pNext Chain Vulkan has a chain of extensions that require this? Also this is related to pNext?.

Point Rendering idk.

Push Constants idk.

Queue Family Can refer to Graphics Queue, Compute Queue, or Transfer Queue. These are objects that receive and execute commands.

Queue Family Index How we access a GPU queue family.

Rasterization Part in graphics pipeline when we draw?.

Render The final output of the graphics pipeline?.

Render Target Views idk.

Resource Barriers idk.

RW Structured Buffer idk.

Sampler2D idk.

Sampling Query a subpixel value.

Semaphore Object responsible for keeping different GPU cores synced.

Slang A shading language that benefits from having generics and interfaces that glsl doesn't have.

SPIR-V GLSL, HLSL, and Slang can all be compiled to SPIR-V, which is passed to Vulkan program.

Staging Buffer Memory from CPU to GPU must go to a **staging buffer** first.

Stencil Attachment idk.

sType **sType** is short for structure type, but why is it important? I guess this relates to how the driver (the GPU driver?) needs to understand the Vulkan program. So sType says to the driver: "The next struct (pNext?) is an image, buffer, or pipeline".

Subresource A part of an image.

Super Sampling Antialiasing (SSAA) An implementation of **anti aliasing**, considered to be 'brute force'.

SV Dispatch Thread ID idk.

SV Point Size idk.

SV Position idk.

Swap Chain A queue of rendered images waiting to be displayed.

Synchronization2 idk.

Ticket Number A timeline semaphore tracks completed tasks in strictly increasing number/sequence. A ticket number (a positive integer) is a certain complete task that we might wait for until we do something else.

Timeline Semaphore Normal semaphore tracks complete or not complete—a **timeline semaphore** tracks more than 1 complete steps in increasing order.

Transfer Queue Good for copying data.

Viewport The rectangular part of the window that is rendered to. For example you could have two viewports (top and bottom), for a split-screen multiplayer effect.

Virtual Function A C++ feature relating to polymorphism and inheritance of classes.

Vk Queue Present KHR The command that tells the OS to display the final image.

vk::binding idk.

VkPipelineColorBlendStateCreateInfo Description.

VkQueueFamilyProperties What properties can a queue family have?.

vkRenderingAttachmentInfoKHR idk, but KHR has to do with talking to OS to display output image.

VRAM Stores memory for GPU.

Z Buffer/ Depth Buffer Used to determine if an object should be rendered in-front of another?.

Appendix

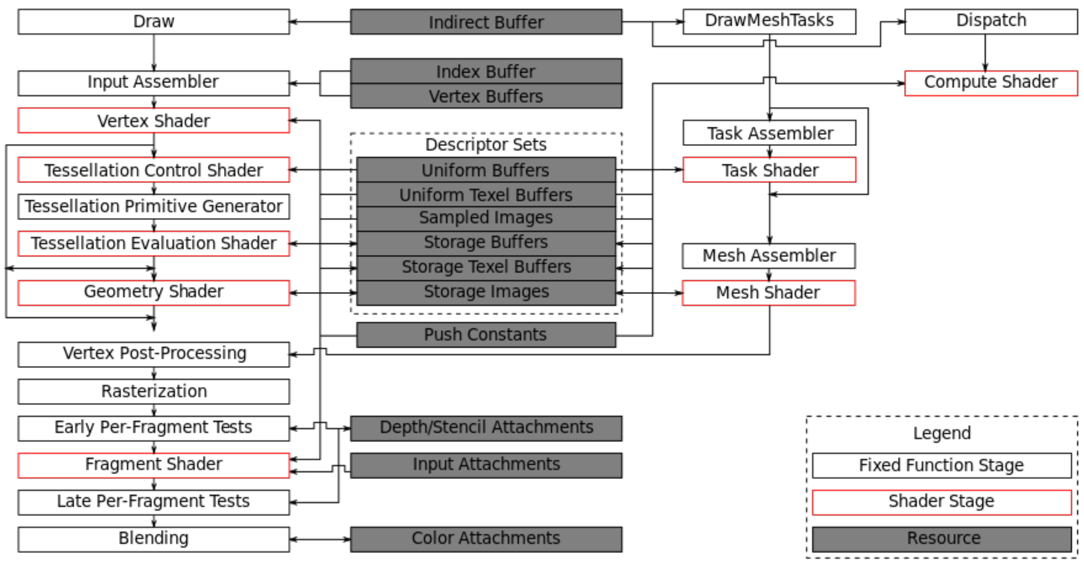

Figure 1: Vulkan Pipeline

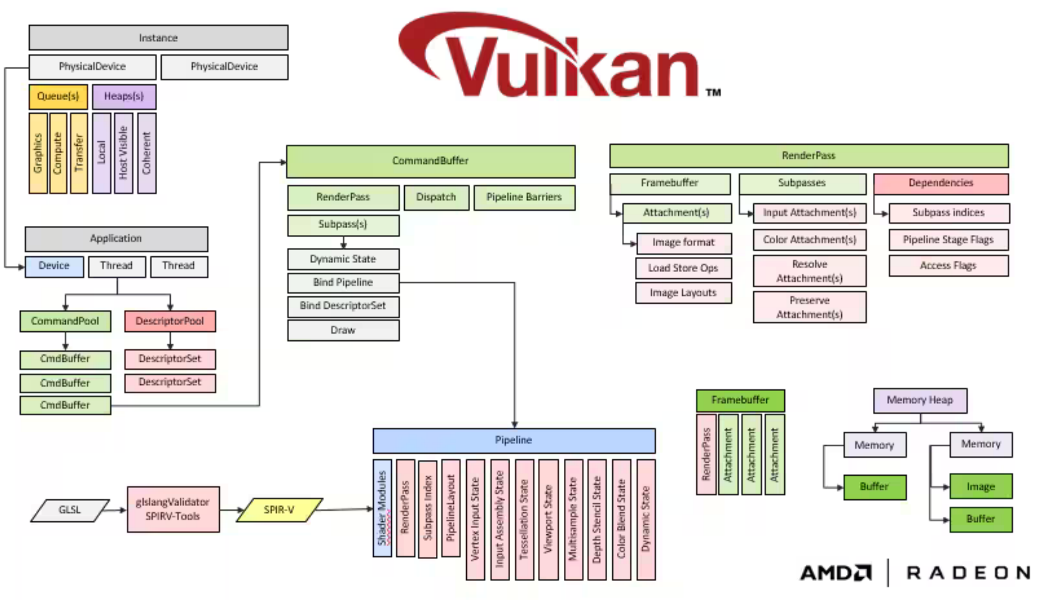

Figure 2: Vulkan Pipeline Again