Vulkan Notes

Joseph Grimsic

November 2025

Contents

Vulkan Questions

- tmp

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
- YouTube Video With Great Diagrams

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.

Sascha Willems Helpers: CreateInfo vs Info

I noticed that Sascha's helper functions—which are designed to reduce Vulkan boilerplate—constantly utilize two specific naming conventions. These are not just random names; they correspond to the specific Vulkan structures we use to pass data to the driver.

The asterisk (*) in the names below acts as a wildcard for the specific object name:

- **_CreateInfo** (e.g., `VkDeviceCreateInfo`): These structs are used during the **Object Creation** phase. They are the "forms" we fill out to tell the driver how to build a persistent object. They always contain `sType` and `pNext`.
- **_Info** (e.g., `VkSubmitInfo`): These structs are generally used inside the **Render Loop**. They pass temporary state or parameters for a specific action, like submitting a draw call. These also utilize `sType` and `pNext`.

Sascha's helper functions simply automate the process of filling out these structs so we don't have to manually set every field (like `sType`) every time.

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() {
```

Vulkan 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 Set Revised

Coming back to the **descriptor set**, I am considering constraints. For example, why not just have one big descriptor set, to solve the problem of not understanding how to separate data into different descriptor sets.

Then of course, if that doesn't work, why not generate a new descriptor set each frame if duplicating the massive single descriptor set is expensive.

I don't know yet, but the following is a table showing how we typically use 4 descriptor sets simultaneously.

Table 1: Descriptor Set Layout Configuration

Set	**Frequency**	**Data Name**	**Description**
Set 0	Global	Frame Data	Camera View/Projection matrices, Global Time, Skybox. (Bound once per frame)
Set 1	Per-Pass	Pass Data	Shadow maps, Light lists, Post-processing buffers. (Bound once per render pass)
Set 2	Per-Material	Material Data	Albedo texture, Normal map, Roughness value. (Bound when material changes)
Set 3	Per-Object	Object Data	Model Matrix (transform), skinning bones. (Bound for every draw call)

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.

The View 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? *Actually, the view matrix is not to be confused with the projection matrix, and the projection matrix relates to the view frustum?*

Usually, this is referred to in two ways:

- The Model-View Matrix: (Legacy OpenGL) A combination of the Model matrix (object position) and View matrix (camera position).
- The View Matrix: (Modern pipelines) A standalone matrix strictly for the camera.

So *really* the **View Matrix** moves all of the meshes in the scene to simulate the camera looking around. But we still don't know what the frustum relates to.

The Projection Matrix

The **projection matrix** can be imagined by considering the concept of *field of view (FOV)*.

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

Okay, but how do attachments play a role in this? How does a render target view (RTV) relate to this? How are attachments different than views? Isn't a view a render target?

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

What is a vkPipelineLayout?

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**.

Note: I wrote this a while ago but just recently move it to this section of the doc

How Do We Allocate A Staging Buffer?

So you want to send memory to the GPU yeah? Well you're going to need a **staging buffer**.

To allocate a staging buffer, we will use the vk object **vks::vkBuffer**. Is the vks namespace special to Sascha Willems' helper functionality?

Does a staging buffer go straight to the GPU? How is the transfer queue involved? Are there more than one staging buffer?

Below is a snippet of buffer staging logic:

```
for (auto &storageBuffer : storageBuffers) {
  // The SSBO will be used as a storage buffer for the compute pipeline and as a vertex buffer in the graphics pipeline
  vulkanDevice->createBuffer(VK_BUFFER_USAGE_VERTEX_BUFFER_BIT | VK_BUFFER_USAGE_STORAGE_BUFFER_BIT |
                                 VK_BUFFER_USAGE_TRANSFER_DST_BIT,
                             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, &storageBuffer, storageBufferSize);
```

Event Queue

Do I want to implement this? Would a physics-based platformer want an event queue? What are they used for? I feel like they would be used for stories or something, but that case would require some check before the next event can happen.

Actually an event queue is used to **decouple** an initial event and a reactionary event (What is decoupling?). You can think of it as storing a message for something that has happened, so that we can use that information later.

UBO and SSBO

So when do we use **UBO** vs. **SSBO**? So SSBO can hold large amounts of memory (gigabytes), and UBO's can hold 16/64 kilobytes and are read only (as opposed to read and write that SSBO's have).

So we know the techincal differences, but which situations suggest which tool? Gemini claims that SSBO's do not replace UBO's

This is true, since UBO is smaller, they are stored in the **constant cache**, which is much faster (and smaller) than L1 and L2 cache hierarchy. Likewise, SSBO's go through L1 and L2 cache.

Some examples of when to use a UBO are the following:

- Camera Matrices: The View and Projection matrices are the same for every single pixel on the screen.
- Global Light Settings: The sun's direction or color doesn't change from pixel to pixel.
- Time: Frame delta time.

Why Do We OR Bits?

Reading through some Vulkan code, you will come across code used to create a buffer. This usually involves ORing bits together—why? Why are we doing bitwise operations when we create a buffer?

Also, why do we use the arrow operator (idk how to type it in latex) when creating a buffer?

Compute Shaders

So you want to massively parallelize your compute on the GPU? You are going to need a **compute shader**.

Dispatch Threads/Command ? Part of compute shaders is creating all of the smaller tasks to be spread across GPU cores/queues(*Aside: can a queue in the compute queue family do more than one thread/process/dispatch command at a time? Is it separated into queues or cores or processes?*)? .

Vulkan Memory Allocater

Apparently the industry standard is in-fact, not to manually allocate all of your Vulkan memory. Instead, you use somethind called the *Vulkan Memory Allocater*, a header-only library developedby AMD to make allocation easier.

Currently, the triangle example does not use the Vulkan Memory Allocater.

Initializing Buffers

So you want to start pushing memory to the GPU eh? This is what buffers are for, and it is usually a 5 step process to initialize them:

1. Create the buffer with vkCreateBuffer() call
2. Get memory requirements vkGetBufferMemoryRequirements() to determine how much memory you actually need
3. Allocate memory (manually) with vkAllocateMemory()
4. Bind memory (VkDeviceMemory) to a buffer object (vkBuffer) with a vkBindBufferMemory() call
5. Upload data so CPU can see it with a vkMapMemory(), memcpy the data into a mapped pointer, and unmap so GPU can take over.

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
```

Ternary Operator

This one is pretty simple, the idea is that if we want to make an if/else statement into one line, we can use the **ternary operator**.

Here is an example:

```
std::default_random_engine rndEngine(benchmark.active ? 0 : (unsigned)time(nullptr));
```

The above can be read as "If benchmark.active, then 0, else (unsigned)time(nullptr))".

Side note: This code is initializing a random number generator (RNG), and setting the seed to 0 if we are benchmarking (for determinism)

Destructors

This one is going to be short for now, but it's probably important. So obviously there are constructors, what do they do? Well they initialize an instance of a class (does it work for objects too?). Well what if the class allocated dynamic memory? Then this would mean that we need to call free at some point right? Do destructors exist so that we don't have to do this?

Auto

So the main idea of **auto** is that it will *automatically* determine the type of a value?

I'm not sure when you would want to use this, but one of the popular ways to use this is with a *for-each loop*. It helps because if we change the tpye of the values we are iterating over, normally we would have to go find the for-each loop in question and change the type there too.

Asynchronous Code

What is the main purpose of asynchronous code? Asynchronous code is important when you want events that happen at the same time, not just from top to bottom as a program is normally executed.

Is this where callbacks are from? Does asynchronous code require callbacks?

The old way of doing things is using **std::async** which gives a **std::future**, which you can call .get() on to retrieve the result. The Problem: It is often naive. Calling .get() blocks the thread until the result is ready. It doesn't easily allow you to chain tasks (continuation) or handle events without blocking.

The new way to implement asynchronous events is using **coroutines**. A coroutine is?

What Is A Valid Starting Point For My Game Engine?

I am currently considering the example from Sascha Willems called **Raytracing Reflections**. I am also considering **Compute Particles**, beause Gemini said that was the closes thing to a vertex-based physics engine—but perhaps it is silly to try to change a code snippet that much.

Some part of me thinks it is a good idea because I don't know how to program the Vulkan pipeline so it would be nice to follow some guideline/snippet.

I suppose I could just line-by-line reference the example code and make changes I think should occur to get me closer to what I want.

Aside: there has to be some sort of asynchronous functionality in the Compute Particles example right? If not, do I need to include a special library for asynchronous code? I should probably dedicate an entry to asynchronous code in the C++ section of this document.

High Level Pitch From Gemini

The following is pseudo code pitch from gemini for the system architecture

Here is the server suggestion:

```
[fontsize=\footnotesize]
class AuthoritativeGameServer:
    def __init__(self):
        self.tick_rate = 60  # Server runs at 60Hz
        self.current_tick = 0
        self.connected_players = {} # Map of ID -> PlayerData
        self.snapshot_history = []  # Buffer of past world states (needed for Lag Compensation)

    def start_server_loop(self):
        while True:
            start_time = get_time()

            # 1. NETWORK RECEIVE
            # We don't process packets immediately upon arrival (interrupts).
            # We buffer them and process them at the start of the tick.
            incoming_packets = network.get_buffered_packets()

            # 2. PROCESS INPUTS (The critical part)
            for packet in incoming_packets:
                player = self.connected_players[packet.player_id]

                # CRITICAL: We don't just set the player's position.
                # We apply the INPUT they sent for a specific Tick.
                # If a player sends inputs for Ticks 100, 101, 102 in one bundle,
                # we must apply them sequentially.
                for input_command in packet.inputs:
                    # Sanity Check: Speed hacks, teleport hacks, etc.
                    if self.validate_input(player, input_command):
                        player.pending_inputs.push(input_command)
```

```
        # 3. PHYSICS SIMULATION STEP
        self.simulate_physics_step()

        # 4. BROADCAST STATE (Snapshot)
        # We don't send every tick (bandwidth). Maybe every 2nd or 3rd tick.
        if self.current_tick % 2 == 0:
            self.send_world_snapshot()

        self.current_tick += 1

        # Sleep to maintain 60Hz
        wait_for_next_tick(start_time)

def simulate_physics_step(self):
    # This is where the standard physics engine runs (Box2D, PhysX)
    for player in self.connected_players:
        # Apply the oldest unprocessed input for this player
        if player.pending_inputs.has_inputs():
            input = player.pending_inputs.pop()

            # Apply forces based on that input
            player.rigidbody.add_force(input.direction * SPEED)
            player.current_simulated_tick = input.tick_number

    # Step the actual physics world
    physics_engine.step(1 / self.tick_rate)

    # Save state for Lag Compensation (Rewind) logic
    self.save_snapshot_to_history()

# COMPLEXITY: LAG COMPENSATION
# This runs when a player shoots. We must see what THEY saw.
def handle_shooting_request(self, attacker_id, target_id, timestamp_when_shot_fired):
    attacker = self.connected_players[attacker_id]

    # 1. Rewind the world to the time the attacker fired
    # Clients are always in the past (interpolation delay).
    # So the server must look back in its history buffer.
    past_state = self.find_snapshot_at_time(timestamp_when_shot_fired)

    # 2. Temporarily move hitboxes to where they were back then
    current_positions = self.cache_current_positions()
    self.apply_snapshot_positions(past_state)
```

```
        # 3. Perform the Raycast / Hitbox check
        did_hit = physics.raycast(attacker.position, attacker.aim_dir)

        # 4. Restore the world to the present
        self.restore_positions(current_positions)

        if did_hit:
            self.apply_damage(target_id)
            self.send_hit_confirmation(attacker_id)

    def send_world_snapshot(self):
        # Create a packet containing the position/velocity of ALL moving objects
        # We include the 'Last Processed Input Number' for each player.
        # This tells the client: "I have processed your inputs up to #50."
        snapshot = {
            "tick": self.current_tick,
            "players": [p.get_state_data() for p in self.connected_players],
            "acks": {p.id: p.last_processed_input_id for p in self.connected_players}
        }
        network.broadcast(snapshot)
```

Here is the client/game suggestion:

```
[fontsize=\footnotesize]
// This pseudo-code illustrates the split between "Prediction" (Local Player)
// and "Interpolation" (Remote Players).

public class MultiplayerPlatformerManager : MonoBehaviour
{
    // STORAGE
    List<InputFrame> localInputHistory; // Stores inputs we've sent but server hasn't ACKed yet
    List<StateSnapshot> localStateHistory; // Stores where we *predicted* we were at each tick
    List<StateSnapshot> serverSnapshots;   // Buffer of states received from server (for other play

    // SETTINGS
    float tickRate = 1f / 60f;
    int currentTick = 0;

    void Update()
    {
        // 1. INPUT COLLECTION
        // We collect input every frame, but we only package it for the next Physics Tick
        InputFrame currentInput = CaptureInput(); // (Jump, Left, Right)
```

```
    // 2. RENDER INTERPOLATION (Visuals only)
    // Physics runs at fixed timestamps. Update runs as fast as possible.
    // We smooth the visual transform between the previous physics tick and the current one.
    RenderSmoothLocalPlayer();
    RenderSmoothRemotePlayers();
}

void FixedUpdate()
{
    // This loop runs exactly 60 times per second

    // 1. PROCESS SERVER DATA (Reconciliation)
    if (network.HasNewServerSnapshot())
    {
        var serverState = network.GetLatestSnapshot();

        // Check if our prediction was wrong
        // We look at our history: Where did WE think we were at Tick X?
        // And compare it to where the Server says we were at Tick X.
        var predictedState = GetStateFromHistory(serverState.tick);

        if (Vector3.Distance(predictedState.position, serverState.myPosition) > 0.05f)
        {
            // ERROR DETECTED!
            // The server is the boss. We must snap to the server's truth.
            Debug.Log("Reconciling: Prediction Error");

            // A. Snap Physics Body to Server Position
            rb.position = serverState.myPosition;
            rb.velocity = serverState.myVelocity;

            // B. Re-Simulate (The expensive part)
            // We have to replay all inputs that happened AFTER this snapshot
            // up to the present moment.
            int ticksToReplay = currentTick - serverState.tick;

            foreach(var pastInput in localInputHistory)
            {
                if (pastInput.tick > serverState.tick)
                {
                    ApplyPhysics(pastInput);
                    Physics.Simulate(tickRate); // Manually step physics engine
                }
```

```
            }
        }

        // Clear inputs that the server has definitely processed
        RemoveOldInputs(serverState.lastAckedInput);
    }

    // 2. CLIENT SIDE PREDICTION (The standard loop)
    // Even if we haven't heard from the server, we assume we are moving.

    InputFrame input = CaptureInput();
    input.tick = currentTick;

    // Apply to local physics immediately (Responsive!)
    ApplyPhysics(input);

    // Save to history so we can check it later against server
    localInputHistory.Add(input);
    localStateHistory.Add(new StateSnapshot(currentTick, rb.position, rb.velocity));

    // Send to server
    network.SendInputPacket(input);

    currentTick++;
}

// -- HELPER LOGIC --

void RenderSmoothRemotePlayers()
{
    // Remote players are NOT predicted. We see them in the past.
    // We take the two most recent snapshots from the server.
    // Snapshot A (Time: 1000ms), Snapshot B (Time: 1100ms)
    // If current time is 1050ms, we show the player 50% between A and B.

    foreach(var player in remotePlayers)
    {
        float interpolationFactor = CalculateRenderTime(player);
        player.visualMesh.position = Vector3.Lerp(player.prevPos, player.currPos, interpolation
    }
}

void ApplyPhysics(InputFrame input)
{
```

```
        // Standard platformer logic
        if (input.jumpPressed && isGrounded)
            rb.AddForce(Vector3.up * jumpForce);

        rb.velocity = new Vector3(input.horizontal * moveSpeed, rb.velocity.y, 0);
    }
}
```

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.

Decouple When two systems can communicate without knowing the state of each other.

Dependancy Injection idk, is this related to dynamically linked libraries(DLL)?.

Depth Image idk, does this relate to the depth buffer?.

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

Descriptor Set This has to do with .

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, relates to Global Invocation ID. I guess just the idea of creating a new thread to do something? But actually gemini says this is too vague as a dispatch thread relates to compute shader, not just a generic CPU thread. You may want to make a section on this...

dstStageMask 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.

Execution Barrier Idk.

Extension Khronos uses **extensions** to officially add functionality to the Vulkan specification?.

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

Fence Object responsible for keeping CPU and GPU synced.

Fragment Shader Determines color of pixels?.

Geometry Shader idk.

Global Invocation ID idk, I think this relates to thread managment. Yes you are right, here is an example (ft Gemini). Global Invocation ID / SV_DispatchThreadID: Imagine you have a grid of 1,000,000 items to process. You launch 1,000,000 GPU threads. This ID is the "Nametag" for the thread. Thread #500 knows it is #500 because this variable tells it so. It uses this ID to look up index #500 in your data array..

GPU Hang idk.

Gradient I know a gradient is a math idea/term but what really is it and why is it important? Is it common to use this in shaders?.

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.

inline What does keyword **inline** do in C++? Is it common? Is it important for performant code?.

Input Assembly Some part of the Vulkan pipeline, but what does it do? Is it at the beginning?.

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").

interrupts idk.

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

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 Map A **mip map** is something that we generate and give to Vulkan. A mip map is how we store a texture with different sizes (factors of 2), so that when the texture is far away Vulkan will sample a lower-resolution texture.

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

newImageLayout idk.

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

oldImageLayout idk.

Pipeline State Object PSO idk, but replaced with shader objects.

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

Point Rendering idk.

prepareStorageBuffers() This is the step where we prepare the buffers to be sent from the CPU to the GPU.

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 When we draw to the screen?? idk.

Rasterization Part in graphics pipeline when we draw?.

Render The final output of the graphics pipeline?.

Render Target Views idk, maybe this is the view we are currently using to write to the image?.

Resource A **resource** can be for example a texture (I don't know what else), and it can hold many **subresources**. So it is usually thought of as a collection of subresources. An example is how a mesh in a game has many 'texures'. Like a roughness map, depth texture etc. The roughness map would be considered a subresource..

Resource Acquisition Is Initialization idk.

Resource Barriers We can think of **resource barriers** as traffic lights that tell other GPU cores when the resource is done being written to.

RW Structured Buffer idk.

Sampler2D idk.

Sampling Query a subpixel value.

Semaphore Object responsible for keeping different GPU cores synced.

Shader Object idk.

Shader Storage Buffer Object (SSBO) idk, this is involved with staging buffers? do we create these on the CPU and send over with the transfer queue? but actually the data goes between shaders, not from CPU to GPU??.

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.

srcStageMask idk.

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

stagingBuffer where we put memory to be sent to the GPU.

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—but actually there are more applications of subresource (like what?).

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, but SV means **system values**, which the GPU gives you? Gemini says that is the output of the vertex shader (*vertex shader's purpose is to map point in 3D space to your 2D screen, based on the view frustum*).

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

Synchronization2 idk, I think it's just a feature added to Vulkan that we can ignore.

Tessellation Shader Can subdivide geometry output from vertex shader—why would we want to do this?.

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.

Uniform A global variable (global to shaders) you pass from your CPU to your GPU shader.

Uniform Buffer Object(UBO) So this is what came before we had Shader Storage Buffer Objects (SSBO). **UBO**'s can store up to 64 or 16 kilobytes depending on the GPU. Other major difference is that UBO's are read only and SSBO's are read and write (at what stage are we reading and writing to SSBO's?).

Vertex Shader Determines where vertices should be drawn on the screen.

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.

vkAccessFlags idk.

vkBuffer idk, I mean I guess it is the primitive buffer object that we allocate in Vulkan? Can we initialize a vkBuffer with the vks::vkBuffer type? how do we populate a buffer?.

vkCmdBeginRenderPass idk.

vkCmdBindPipeline idk.

vkCmdDraw Description.

vkMapMemory idk.

vkMemAllocInfo idk.

vkPipelineCache idk, why are we caching the pipeline?.

VkPipelineColorBlendStateCreateInfo Description.

VkQueueFamilyProperties What properties can a queue family have?.

vkQueueSubmit idk, submit commands to queue? or submit queue (but this doesn't make sense).

vkRenderingAttachmentInfoKHR (Gemini) The Concept: Think of this as a "Work Order" you hand the GPU right before it starts drawing. It tells the GPU: "I am about to draw some geometry. Please output the resulting colors to this specific image and the depth info to that specific image." It simplifies older Vulkan code by letting you define these targets dynamically..

vkShaderModule idk.

vkStructure What is a **vkStructure**? How does Sascha Willems save us from assignments with the **vks** namespace?.

VkSurfaceKHR idk, it has something to do with the image pipeline. So there is something called a surface? How does this relate to Vulkan pipeline?.

vkUnmapMemory This is basically a cleanup function that we should call after we use vkMapMemory.

VRAM Stores memory for GPU.

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

Networking

So there are many networking solutions—which is the best for me? Here are some of the libraries I have come across:

- ENet
- RakNet
- BoostASIO

Are any of these options specific to Windows? If so maybe we should avoid it. And if they aren't operating system specific do we need to interface with the OS ourselves?

So we know that different CPU architectures have different representation of floats, which is a problem for a multiplayer physics engine. How do you plan to reconcile? Well there has to be some networking idea that lets us get away with this.

Server Authority (State Sync)

So what's the idea here? We do the physics on the client in between server ticks?

The first thing to realize is that a normal server tick might be 20hz. A good refresh rate nowadays might be 240hz. So there's more than 10 frames to be **interpolated** between server ticks.

Recall that our biggest issue is that floats across hardware are represented differently. Because of this, our physics is not synced across systems. Handling this is called the **Authority Check**.

The Authority Check (Reconciliation)

There are three parts of the authority check:

1. Snapshot Arrival
2. Comparison
3. Reconciliation

Snapshot Arrival Snapshot simply means the current state (server state in this case). When it arrives, we must adjust our client such that it becomes synced with the server state/snapshot.

Here is a table from Gemini that might help:

Feature	Client Role	Server Role
Input	Sends inputs (WASD, Mouse) to server immediately.	Receives inputs, queues them for the next tick.
Physics	**Predicts** results instantly to keep gameplay smooth.	**Calculates** the "true" result based on game logic.
State Sync	Receives server state; corrects (reconciles) if prediction was wrong.	Broadcasts the "true" state to all connected clients.

The Standard Approach

Client Side Prediction This is the idea that the client's inputs happen on screen without verification from server (so they are immediate/responsive). Then the action is sent to the server which will have a list of checks so that we know that the client's inputs are valid (no funny business). If all the checks pass, then the client's inputs reach the server and become real for all players. Then perhaps there is some memory passed back to client of the real state of everything, including their own actions?

But this is only a small example of the idea that is client-side prediction. What if we have a networked physics object, say a bouncing ball. If the authoritative server just sends the position of the ball, we can only create a new frame when we receive a packet from the server with a position update. Instead, if we sent some extra information like the velocity and accelaration, we could interpolate on the client to generate frames that we would have otherwise had to wait for.

Aside: I am learning some of this from this video

But How Do We Make Other Players Smooth? As we've discussed, we know our own inputs with no delay, so we can predict our server state (real state). But what about other players? If we were to just update them whenever we get a tick of packets from the server, then the would appear to teleport *(recall that we have more than 10 frames to interpolate if we are on 240hz client with 20hz server tick).*

To solve this, we can buffer other players. In other words: we will render them *slightly in the past.* This way, we will have a continuous stream of inputs as if they were giving inputs locally. *One problem this might raise is: what happens when we interact with other players? For example if we grab the other player like in LittleBigPlanet, we then have to do physics that rely on the other player—so our client-side prediction may be laggy. But maybe this is acceptable.*

Packets

So you want to network? You're going to need to understand packets. Packets are the unit that we use to send data on our network. Normally, packets have a max size of

Debugging

So it seems there are not that many easy debugging setups. For now though, I should just go for a prototype.

Some setups to consider are the following:

- GDB in separate terminal
- RenderDoc
- Validation Layers
- :TermDebug
- Switching to IDE?

When using a debugger, you need to generate debug symbols. Because of this it makes sense to have two directories, one for debug build and one for release build.

Modifying Compute Particle

So my plan to learn Vulkan is to work from this base program: *compute particle* from the Sascha Willems Vulkan repo, and to turn it into a minecraft demo? That seems like a fun project. That would teach me the graphics pipeline after all, and some basic structure of making a game prototype and stuff.

Joseph from the future here. Modifying computeparticles.cpp is really hard. It is a 500 line cpp file with many Vulkan abstractions I don't understand. There must be a better way to learn Vulkan. The problem is, I don't know where to start. It seems logical to start from a template like compute particles and try to change it, because that way I would still be using the 'Sascha' setup that I want to adopt. The reason it is so difficult is because I don't know where all the changes need to be made. I mean, if I wanted to change the program then realistically I would scrap the whole thing. For example, if I am going to draw a triangle to the screen, I don't need to use a compute shader—which I believe compute particles is using. Okay, so then maybe I could modify the program in a different way that changes the functionality of the compute shader? So far, I have been taking the approach of looking through the source code first, then finding anything I didn't understand (usually a variable name), and then prompting gemini some questions so that I could make a note about it in this document. So I wonder: what is the amount of working knowledge I need to start prototyping? If only I could successfully transform the code into something else, then I could build confidence

Okay, so my new approach will be starting with some triangles.

Currently, we are compiling with the glsl shader, but I want to use slang instead. So I need to find where we are requiring the glsl shader.

It could be worth trying to work from the *vulkan scene* example too, it looks like a good starting point for a game engine.

Analysis Of Triangle Example

This section will simply be me writing what I think is happening in triangle.cpp from top down. Preferably I touch on every line, and every question I have on every line. I think I will copy each segment of code into this doc and then below write my thoughts.

```
class VulkanExample : public VulkanExampleBase {
```

I am not entriely sure what this syntax means. I believe it is inheritance. Also, I don't know which direction the inheritance goes—maybe from right to left like '=' operator? In that case we are initializing a new class, VulkanExample, from the base class VulkanExampleBase. So all of the inherited functionality would probably be from functions that we can call from VulkanExample.

Yes, this line is inheritance from what is called a 'base class', and the direction of the inheritance is the same as the '=' operator as you guessed. Sascha has already implemented the functionality for initializing the GPU and creating a window, so we inherit that functionality to our class: *VulkanExample*, note the pascal case for the class. Also, *VulkanExampleBase* has functions that are meant to be overridden—one being render(). These functions are virtual functions.

- What is a virtual function?
- How do I use a virtual function?
- What are some other virtual functions that I will have to override?

```
public:
  // Vertex layout used in this example
  struct Vertex {
    float position[3];
    float color[3];
  };

  struct VulkanBuffer {
    VkDeviceMemory memory{VK_NULL_HANDLE};
    VkBuffer handle{VK_NULL_HANDLE};
  };
```

So we are defining a struct Vertex that holds position and color floats. Note that the struct initializations in cpp seem pretty easy: keyword 'struct', then its identifier, its members, and then a semi colon after the closing brace.
What is confusing though, is the VulkanBuffer struct. We are initializing a VkDeviceMemory variable called memory to VK_NULL_HANDLE and the VkBuffer called handle, also initialized to VK_NULL_HANDLE. Do we just initialize things to VK_NULL_HANDLE by default or before we have something else initialized to pass to it? Where else are these structs used?
Most importantly, what are these Vulklan types used for? I know that VkDevice probably means something related to the GPU—but what is VkDeviceMemory? Is a VkBuffer just some generic block of memory? When do we use these types? What are their applications?

So there are some important concepts to take away from this code segment. In Vulkan, objects like VkDeviceMemory and VkMemory are 'opaque pointers' to internal objects managed by the GPU driver. VK_NULL_HANDLE is used as a default value that we can initialize to. There are two good reasons for this default value. If they accidentally never point to an object, they can still be de-allocated at cleanup time. The second reason is that we can track the state of the object with a check like:

```
if (handle != VK_NULL_HANDLE){
```

We want to treat an object differently if it has not been allocated yet.

Both of these datatypes are fundamental to Vulkan. A VkBuffer object is a handle to memory. A VkDeviceMemory object is a large block of memory that we allocate at once. With a VkDeviceMemory object, the best practice is to throw a bunch of memory together and have more than one handle into it (many VkBuffer objects into different sub-blocks of memory in VkDeviceMemory).

However, this raises another point. To allocate different sub-blocks of memory, there are two different approaches:

1. Allocate manually with VkAllocateMemory()
2. Allocate with *Vulkan Memory Allocater*

- Is VkAllocate() the only function we use to manually allocate memory?
- Should I rewrite the triangle example program to use the VMA?
- How will we use the VulkanBuffer struct?

```
VulkanBuffer vertexBuffer;
VulkanBuffer indexBuffer;
uint32_t indexCount{0};
```

So I guess we previously made the VulkanBuffer struct to store our vertex and index buffers. But I don't understand how we will initialize these buffers, and how the shaders will unpack the buffers to draw to the screen.

Also, I don't understand why we have a variable for index count?

So we are now learning how to initialize this VulkanBuffer struct. Recall this struct has two parts: VkDeviceMemory and VkBuffer. See the subsection on initializing buffers to learn more.

- What is the difference between object types that start with vk and Vk? Does it relate to Sascha abstractions vs Vulkan API?

Answer to blue box above:

- vk: represents a standard vulkan function
- Vk: represents a standard vulkan datatype
- vks:: : represents a Sascha abstraction

```
// Uniform buffer block object
struct UniformBuffer : VulkanBuffer {
  // The descriptor set stores the resources bound to the binding points in a shader
  // It connects the binding points of the different shaders with the buffers and images used for
  VkDescriptorSet descriptorSet{VK_NULL_HANDLE};
  // We keep a pointer to the mapped buffer, so we can easily update it's contents via a memcpy
  // joseph: what is the variable called map, and why is it an 8-bit unsigned int?
  uint8_t *mapped{nullptr};
};
```

Now we are getting into this mysterious mapped pointer, and we are creating another struct. But wait, we have not seen a struct get defined this way before. Is this struct inheriting the members from the Vulkan Buffer struct? I know that descriptor sets are like pointers that we give to shaders. Why do we need a *mapped pointer? What is its purpose?

We are getting to the idea of how memory on the GPU (VRAM) is a black box from the CPU's perspective. Mapping is the process of asking the GPU driver to make a window for the CPU to look into its memory.

Firstly, this code segment *does* show struct inheritance.

It may be helpful to recall how we use descriptor sets.

Deciding If I Should Stick With Sascha

Apparently, there are some outdate aspects of Sascha's Vulkan Repo, so I think it is important to discuss whether or not I should stick with it.

Here is an article about some modern Vulkan code: blog post on modern Vulkan

Here is a new resource for Vulkan Samples that I am considering: Vulkan-Samples

So what is different about these two resources? Well, the Vulkan-Samples official repo use the newest Vulkan features, such as synchronization 2.

Though, there are some trade-offs. Vulkan-Samples does not have a lot of the built in functionality that Sascha's repo gives. I wonder how hard it would be to port some of the functionality? I wonder if it is possible. If there are any base classes that are not cleanly separated from the Vulkan API, then they are probably a lost cause.

I'm also not sure if Vulkan-Samples has a lot of the struct groupings that Sascha's repo has.

To-Do (For This Document)

I think this might be a nice place to keep my thoughts about this document in an organized manner.

Appendix

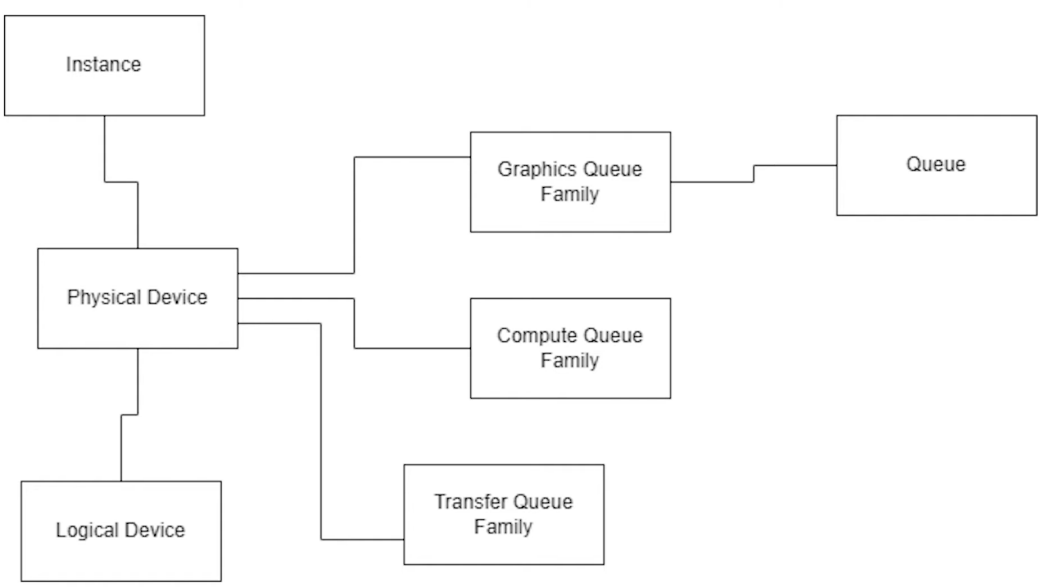

Figure 1: Vulkan Pipeline

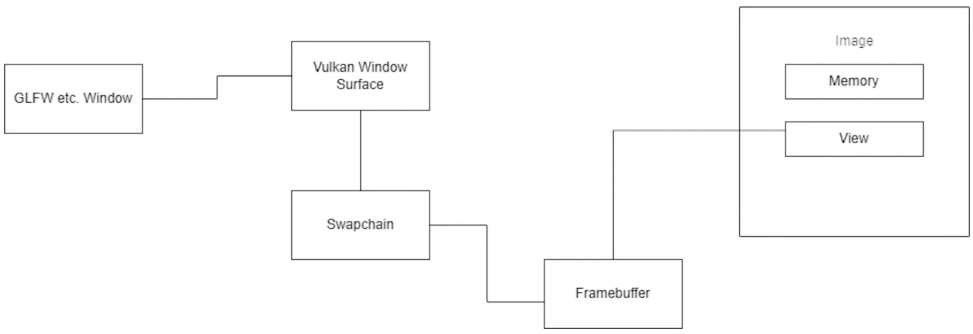

Figure 2: Vulkan Pipeline

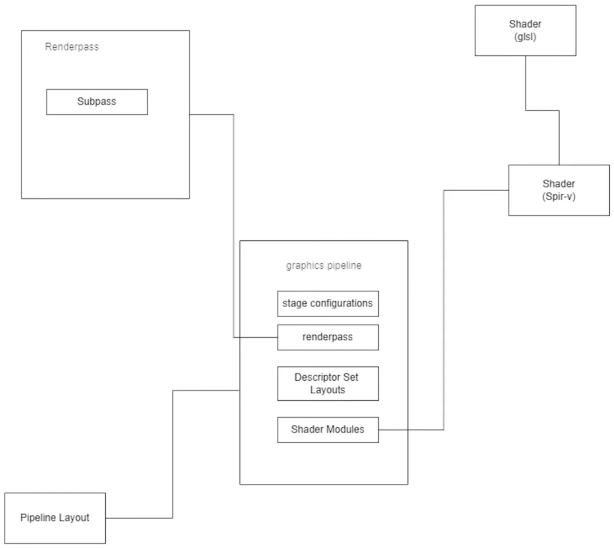

Figure 3: Vulkan Pipeline

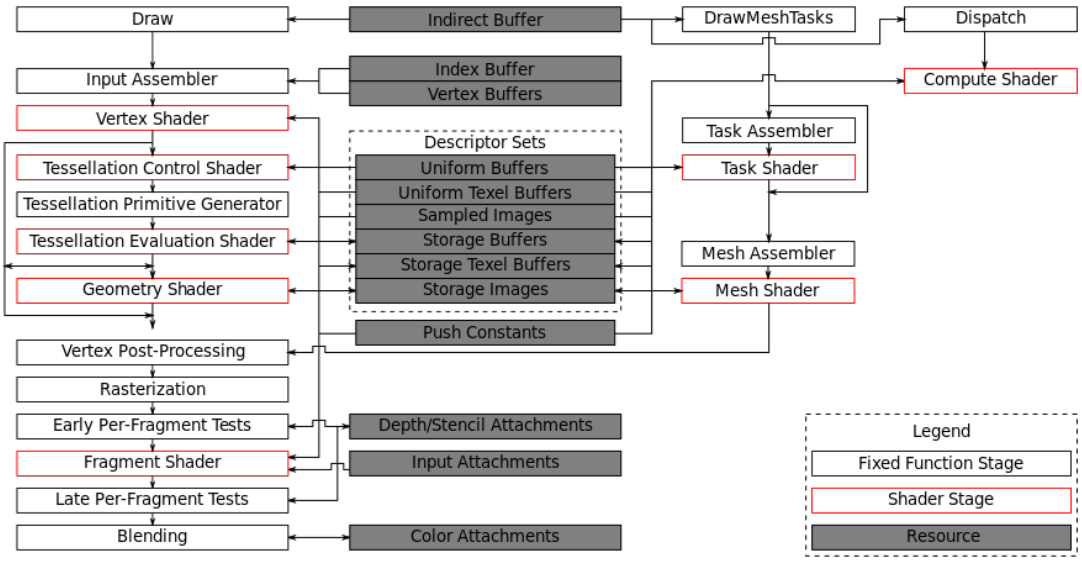

Figure 4: Vulkan Pipeline

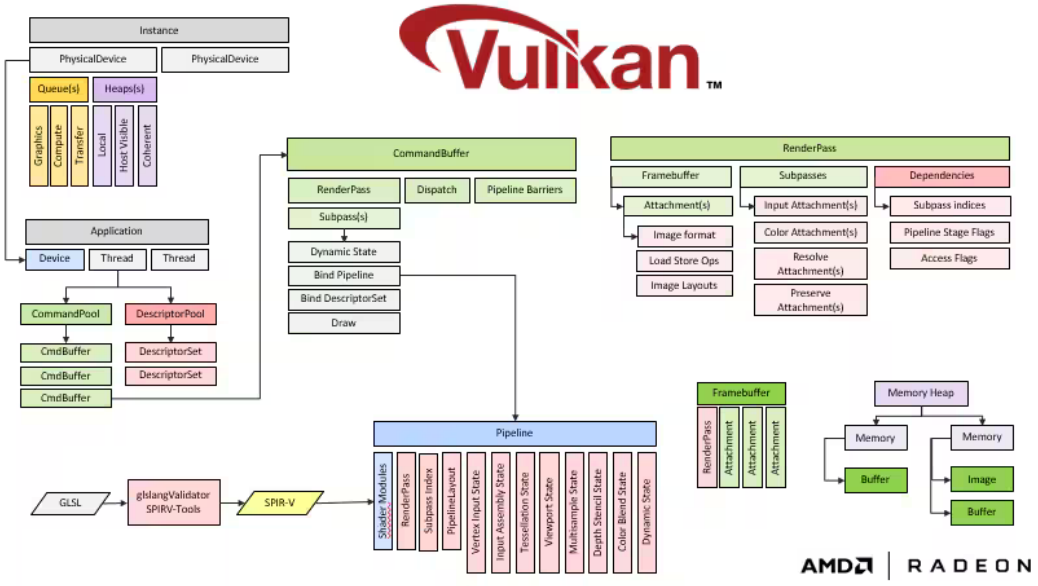

Figure 5: Vulkan Pipeline