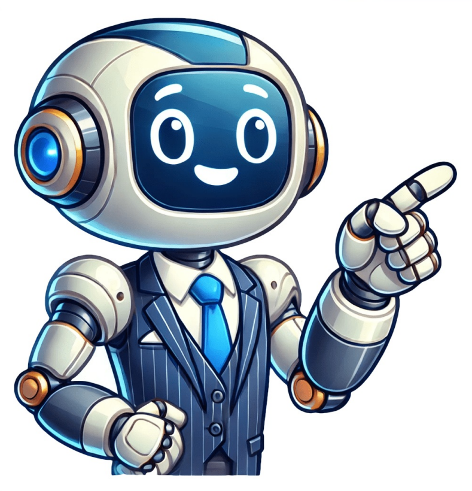
I'm not a robot



How to learn vulkan

The Ultimate Guide to Building Stunning 3D Graphics with Vulkan Thanks to the Khronos membership and our passionate developer community, we have a comprehensive set of well-supported resources to help you get started with Vulkan application development. Vulkan: The Next-Generation Graphics API Discover how to unlock the full potential of modern GPUs using Vulkan's powerful programming model. Learn how to build impressive 3D graphics with the latest graphics API. Perfect for Graphic Programmers This book is ideal for graphic programmers looking to get up and running with Vulkan, as well as those with experience in OpenGL and other APIs who want to take advantage of next-generation capabilities. Learn the Fundamentals of Vulkan Programming Get hands-on experience with device, command buffers, queues, validation layers, memory management, buffer and image resources, drawing operations, GLSL shaders, and rendering 3D scenes. Unlock Realism in Your Rendered Scenes Bring your 3D scenes to life with textures, linear and optimal texture implementation, and master the art of managing resources with synchronization objects. Looking to master command buffers and acquire knowledge on recording operation commands for GPU processing. We'll delve into memory management, buffer, and image resources to create drawing textures and views for the presentation engine, as well as store geometry information in vertex buffers. We'll get a brief overview of SPIR-V, the new shader management method, and define drawing operations as a single unit of work in the Render pass with attachments and subpasses. You'll also build frame buffers and establish a solid graphics pipeline, leveraging synchronization mechanisms to manage GPU and CPU hand-shaking. By the end, you'll be equipped with knowledge to build applications using the Vulkan API. Since prior experience with C/C++ is assumed, this tutorial takes a practical approach without excessive hand-holding. We'll explore the basics of using the Vulkan graphics and compute API, which provides improved abstraction for modern graphics cards compared to existing APIs like OpenGL and Direct3D. Vulkan offers better performance and reduced driver behavior surprises, but comes with a more verbose API requiring setup from scratch for every detail. The graphics driver will do less hand-holding, necessitating more work in the application to ensure correct behavior. This tutorial is geared towards programmers enthusiastic about high-performance computer graphics willing to invest time. If you're more interested in game development, OpenGL or Direct3D might be a better fit. Alternatively, using an engine like Unreal Engine or Unity can provide a higher-level API while utilizing Vulkan. Looking for help on C++17 and Vulkan? This tutorial is designed for developers already familiar with 3D computer graphics, covering features like classes and RAII, as well as alternative resources such as Rust. The guide will use the original C API to work with Vulkan for developers using other languages. We've thoroughly tested all the code files on various graphics cards from multiple vendors to ensure correctness. Each chapter includes a comment section where you can ask questions relevant to the specific topic. To assist us better, please provide details about your platform, driver version, source code, expected behavior, and actual behavior. This tutorial is a community-driven effort as Vulkan is still a relatively new API, and best practices are yet to be established. We welcome feedback on the tutorial and site via GitHub issues or pull requests. The repository will notify you of updates. After completing the initial setup, we'll expand on linear transformations, textures, and 3D models. While each step may seem daunting at first, it's easy to understand, and each step builds upon the previous one. As you progress, drawing fully textured 3D models won't be as challenging. We've included an FAQ section to address common problems; if you're still stuck, feel free to ask for help in the comment section of the relevant chapter. Looking into Vulkan can be overwhelming. My project started as an inefficient implementation but served as a starting point for many. This article aims to clarify some aspects of Vulkan, but it's essential to be patient and not rush through the learning process. A self-taught programmer like myself can build something with Vulkan, so you can too. To get started, consider learning OpenGL first, which is easier to grasp and provides a solid foundation for understanding graphics programming concepts that will still apply when working with Vulkan. It's recommended to display a textured model on the screen with simple Blinn-Phong lighting and explore basic shadow mapping techniques to learn rendering from different viewpoints and sample depth textures. For learning OpenGL, focus on the latest practices, such as OpenGL 4.6, which can make writing OpenGL more enjoyable and easier to transition to Vulkan. Having some math knowledge, particularly linear algebra, is also beneficial, especially with regards to vectors, transformation matrices, and quaternions. My favorite resource for this is 3D Math Primer for Graphics and Game Development by F. Dunn and I. Parbery. Be mindful of bike-shedding, a pattern of overthinking and over-engineering, especially when working on Vulkan. Always ask yourself if something is truly necessary and will not become a bottleneck in the future. Don't overcomplicate simple problems by imitating others; instead, focus on finding solutions that work for you. It's easy to fall into the trap of over-engineering, but remember that most games don't require such complexity. Prioritize getting things working before worrying about making them perfect. Take a pragmatic approach and leave "TODO" comments in place until they become actual problems or hinder performance. You might be surprised by how many issues don't materialize. This advice primarily applies to solo hobby projects, where flexibility is key. However, as your project grows and others rely on it, rewriting fundamental parts can become daunting. That's why it's essential to weigh the need for learning a new graphics API against the benefits of using an existing engine like Godot or Unreal Engine. My use case is straightforward: I want to create small 3D games for desktop platforms (Windows and Linux) while embracing open-source technology and standards. Given this context, Vulkan stood out as the more suitable choice over OpenGL, which may lack future-proofing and has uncertain prospects. WebGPU was another contender, but I encountered issues with its instability and limited resources. WGSL is a decent shading language, but I prefer GLSL's syntax. On desktop, WebGPU essentially wraps around native graphics APIs like DirectX, Vulkan, or Metal, introducing limitations that don't exist in native implementations. This is where RenderDoc captures can become confusing due to differences between platforms (e.g., DirectX on Windows and Vulkan on Linux). Using WebGPU can be a bit tricky on Linux, as it doesn't have a direct mapping to native API calls like Dawn or WGPU. It's similar to using bgfx or sokol, but you don't get the same level of control over the GPU. Some features, like bindless textures and push constants, are currently in development. However, WebGPU has its advantages: it's better than OpenGL/WebGL and can be more useful than Vulkan in certain situations. Validation errors are improved, and not having global state makes things easier to manage. WebGPU is also similar to Vulkan in many ways, so learning a bit about it before diving into Vulkan was helpful for me. It requires less boilerplate code to get started, and you don't have to deal with explicit synchronization, making things simpler. One of the best features is that you can make your games playable directly in the browser. Vulkan Engine Development: A Project Born from Learning and Growth I started this project to learn Vulkan, but it quickly evolved into a usable engine for my future projects. At the time of writing, the source code line counts are as follows: 19k lines of code for the engine itself, 6.7k lines related to graphics, and 3k lines for light abstractions around Vulkan. The 3D cat game has 4.6k LoC, while the 2D platformer game has 1.2k LoC. I reused some non-graphics-related code from my previous engine but rewrote most of the graphics and core systems from scratch. The commit history showcases how it began with clearing the screen, drawing a triangle, and eventually adding textures to draw quads. It may be easier to understand the engine's inner workings now that it's smaller and simpler. Key Features and Techniques 1. **Model Loading**: Models are loaded into the compute shader, which then produces a buffer of vertices for skinned meshes. 2. **CSM (Cascaded Shadow Mapping)**: A 4096x4096 depth texture with three slices is used for cascaded shadow mapping. The first slice renders all models and calculates shading using the shadow map and light info. The fragment shader performs calculations for all lights affecting the drawn mesh in one draw call. The rendering process involves drawing everything into a multi-sampled texture, which is then resolved manually via a fragment shader. Post-processing effects like depth fog, tone-mapping, and bloom are applied after dialogue UI is drawn, all within a single draw call. Future Work and General Advice While the engine is basic now, it will likely become more complex in the future. For writing Vulkan code, recommended libraries include: * vk-bootstrap: Simplifies many Vulkan boilerplate tasks like physical device selection and swapchain creation. The project's history and development can be followed by exploring the commit history, which provides a clear understanding of how the engine has evolved over time. Vulkan Libraries Simplify Development Process Similar to large wrapper libraries around graphic APIs due to their strong opinions, managing Vulkan functions requires maintaining a mental map of "wrapper function vs function in the API spec." Fortunately, vk-bootstrap doesn't fit this mold, primarily affecting the initialization step and avoiding wrapper-like behavior for all Vulkan functions. The author started learning Vulkan from scratch without relying on third-party libraries and found vk-bootstrap to be incredibly beneficial. Replacing extensive initialization code was a joy. They also used VMA to simplify memory allocation, which allowed them to focus on other aspects of the framework. A key feature of vk-bootstrap is its ability to load extension functions automatically after calling volkInitialize. This eliminates the need to store pointer references throughout the application. The author appreciates how volk.h can be included and then call functions like vkSetDebugUtilsObjectNameEXT directly, making the code more streamlined. The author has also created a GfxDevice class that encapsulates commonly used Vulkan functionality, handling tasks such as context initialization, swapchain creation, command buffer management, image creation, buffer allocation, and bindless descriptor set management. This abstraction simplifies passing multiple objects to functions instead of individual components like VkDevice, VkQueue, or VmaAllocator. When it comes to shaders, Vulkan offers flexibility with various shading languages that compile to SPIR-V, including GLSL. The author chose GLSL due to its familiarity from OpenGL experience and opted to pre-compile shaders during the build process for simplicity and reduced runtime dependencies. Shader errors are also detected during compilation, preventing runtime issues. This approach streamlines shader loading and reduces the likelihood of compile-time errors during runtime. Shaderc is a tool part of the Vulkan SDK, allowing users to specify a DEPFILE in CMake when working with shader includes. This simplifies the build process, especially for projects with multiple shaders. When a shader file changes, all dependent files are recompiled automatically without the need for manual updates. A custom CMake function, `target_shaders`, is used to manage this process. It creates a directory for compiled shaders and generates SPIR-V files from GLSL code using `glslc`. The dependency information is stored in a DEPFILE, which enables CMake to track changes and rebuild affected files. In the main `CMakeLists.txt` file, the shader files are specified along with their directory path. When building the game target, shaders are automatically compiled and placed in the binary directory. Vulkan's descriptor sets can make passing data to shaders more complex compared to OpenGL. Descriptor sets group uniforms into sets that need to be defined beforehand, used with pools, and updated through specific Vulkan calls. However, the author bypasses these complexities by using bindless descriptors for textures and samplers, and push constants for other uniform values. The code implements a post-processing effects (PostFX) pipeline using the Vulkan graphics API. It utilizes a "pipeline" class to separate drawing steps into distinct classes for easier handling. The PostFXPipeline class is responsible for initializing and cleaning up the pipeline, while also providing a draw function to be called each frame. The initialization process loads necessary shaders and initializes the pipeline and pipeline layout. The cleanup function simply destroys the pipeline and its associated layout. In terms of drawing, the code assumes that synchronization has been performed outside of the draw call. - vkCmdPushConstants was called, binding bindless texture IDs to pipeline layout for VK_SHADER_STAGE_FRAGMENT_BIT . After that some drawing took place. A fullscreen triangle was being drawn to create a full-screen effect. Then vkCmdDraw was used to draw the command buffer with 3 commands and their respective parameters. It is assumed here that 'draw' function call happens between vkCmdBeginRendering and vkCmdEndRendering - render pass doesn't care about the target it renders to, caller of 'draw' is responsible for it. A VkRenderingInfo object was created as a handy wrapper. It used drawImage's extent2D, colorImageView, colorImageClearValue, depthImageView, depthImageClearValue & resolveImageimageView as their respective values. The meshesToDraw were then drawn using the meshPipeline.draw function with gfxDevice and mesh objects. Then the skyboxPipeline was drawn using camera object. Finally vkCmdEndRendering was called to end rendering process. ARB_bindless_texture simplified working with textures, allowing for more efficient sampling. In contrast, Vulkan's similar functionality requires maintaining an "image manager" to track loaded textures and inserting them into descriptor sets. This enables passing "texture ids" as push constants, simplifying texture sampling in shaders. To sample a texture using these bindless techniques, developers must initialize separate image samplers for different texture types, such as common samplers like nearest, linear with anisotropy, or depth samplers. A wrapper function aids the sampling process, while nonuniformEXT placement can be tricky but is explained well in related resources. When working with materials, bindless textures enable more compact descriptor sets by passing a single material ID through push constants, which then sample texture like this: `MaterialData material = materials[pcs.materialID]; vec4 diffuse = sampleTexture2DLinear(material.diffuseTex, inUV);` Additionally, Vulkan supports binding multiple texture types within the same set using different layouts, such as `layout (set = 0, binding = 0) uniform texture2D textures[];` or `layout (set = 0, binding = 1) uniform sampler samplers[];`. A related article on using bindless textures in Vulkan provides further insight into these techniques. Pre-allocating large arrays and pushing data to them can be a useful approach for handling dynamic data. By pre-allocating an array of N structs or matrices, you can then push new data to it from the CPU at each frame, allowing access to this data in your shaders. One common method is to use multiple buffers on the GPU, swapping between them as needed. This requires careful management to avoid races and ensure parallelism. A commonly used approach involves using one buffer for the "currently drawing" frame and another for recording new drawing commands. Another alternative is to use a single buffer on the GPU and multiple staging buffers on the CPU, which can help conserve memory on the GPU. In this setup, data from the CPU is written to the first buffer during the initial frame, while subsequent frames read from one buffer and write to another. This method requires careful management of the "in-flight" frames. The example code snippet provides an implementation of a buffer class that handles these operations. The `NBuffer` class uses pre-allocated buffers on the GPU and staging buffers on the CPU to manage data streaming, with careful management of buffer swaps and races. The text describes the process of reading and writing data from a buffer using Vulkan. It explains two approaches for syncing data: one that involves manual synchronization with previous reads and writes, and another that relies on Vulkan's copy commands. The author suggests using the second approach for most cases, as it is more efficient when dealing with large amounts of data to be transferred between the CPU and GPU. However, they note that this approach may require additional memory management to conserve GPU resources. Regarding object cleanup, the author questions the usefulness of the deletion queue pattern in Vulkan Guide and prefers a different approach. They suggest using custom classes with move constructors and move operator= to manage object lifetimes and cleanup. This approach provides more control over object destruction and minimizes the risk of accidentally destroying in-use objects during cleanup. The author shares their own implementation, which involves cleaning up resources directly within the class, rather than relying on dynamic allocation or manual cleanup functions. While this approach is not perfect, it reduces the likelihood of forgetting to call cleanup functions and provides a more straightforward way to manage object lifetimes. Validation Error: [VUID-vkDestroyDevice-device-05137] Object 0: handle = 0x4256c1000000005d, type = VK_OBJECT_TYPE_PIPELINE_LAYOUT; | MessageID = 0x4872eaa0 | vkCreateDevice(): OBJ ERROR : For VkDevice 0x27bd530[], VkPipelineLayout 0x4256c1000000005d[] has not been destroyed. The Vulkan spec states: All child objects created on device must have been destroyed prior to destroying device (VMA also triggers asserts if you forget to free some buffer/image allocated with it. I find it convenient to have all the Vulkan cleanup happening explicitly in one place, making it easy to track when objects get destroyed. Synchronization is difficult in Vulkan as it requires explicit handling, unlike OpenGL and WebGPU which handle synchronization for you. To manage this complexity, I manually insert barriers between drawing passes in my code. For example, the skinning pass writes new vertex data into GPU memory, while the shadow mapping pass reads this data to render skinned meshes into a shadow map. My sync code looks like this: ```cpp // do skinning in compute shader for (const auto& mesh : skinnedMeshes) { skinningPass.doSkinning(gfxDevice, mesh); } { // Sync skinning with CSM const auto memoryBarrier = VkMemoryBarrier2{ .sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER_2, .srcStageMask = VK_PIPELINE_STAGE_2_COMPUTE_SHADER_BIT, .srcAccessMask = VK_ACCESS_2_SHADER_WRITE_BIT, .dstStageMask = VK_PIPELINE_STAGE_2_VERTEX_SHADER_BIT, .dstAccessMask = VK_ACCESS_2_MEMORY_READ_BIT, }; const auto dependencyInfo = VkDependencyInfo{ .sType = VK_STRUCTURE_TYPE_DEPENDENCY_INFO, .memoryBarrierCount = 1, .pMemoryBarriers = &memoryBarrier, }; vkCmdPipelineBarrier2(cmd, &dependencyInfo); } // do shadow mapping shadowMappingPass.draw(gfxDevice, ...); ``` I might implement render graphs in the future to automate/simplify synchronization. Right now, I'm okay with manual sync. Additionally, vkconfig's "synchronization" validation layer helps greatly in finding sync errors. Some useful resources for understanding synchronization include: Given article text here The base coordinate system is defined with six vertices (0,0), (0,1), (1,1), (1,0), (0,0) forming a quad. All sprite draw calls are combined into SpriteDrawBuffer, which consists of the following GLSL structure: struct SpriteDrawCommand { mat4 transform; // could be mat2x2... vec2 uv0; // top-left uv coord vec2 uv1; // bottom-right uv coord vec4 color; // color by which texture is multiplied uint textureID; // sprite texture uint shaderID; // explained below vec2 padding; // padding to satisfy "scalar" requirements }; layout (buffer_reference, scalar) readonly buffer SpriteDrawBuffer { SpriteDrawCommand commands[]; }; On the CPU/C++ side, it's almost the same: struct SpriteDrawCommand { glm::mat4 transform; glm::vec2 uv0; // top-left uv coordinate glm::vec2 uv1; // bottom-right uv coodinate LinearColor color; // color by which texture is multiplied std::uint32_t textureId; // sprite texture std::uint32_t shaderId; // explained below glm::vec2 padding; }; std::vector spriteDrawCommands; Two fixed-size buffers are created on the GPU and then populated with the contents of spriteDrawCommands. The sprite renderer is used as follows: // record commands renderer.beginDrawing(); { renderer.drawSprite(sprite, pos); renderer.drawText(font, "Hello"); renderer.drawRect(...); } renderer.endDrawing(); Later, actual drawing occurs using the following command: vkCmdDraw(cmd, 6, spriteDrawCommands.size(), 0, 0); // 6 vertices per instance, spriteDrawCommands.size() instances in total The complete sprite.vert looks like this: #version 460 #extension GL_GOOGLE_include_directive : require #extension GL_EXT_buffer_reference : require #include "sprite_commands.glsl" layout (push_constant) uniform constants { mat4 viewProj; // 2D camera matrix SpriteDrawBuffer drawBuffer; // where sprite draw commands are stored } pcs; layout (location = 0) out vec2 outUV; layout (location = 1) out vec4 outColor; layout (location = 3) flat out uint textureID; layout (location = 4) flat out uint shaderID; void main() { uint b = 1 = pcs.numVertices) { return; } SkinningDataType sd = pcs.skinningData.data[index]; mat4 skinMatrix = sd.weights.x * getJointMatrix(sd.jointIds.x) + sd.weights.y * getJointMatrix(sd.jointIds.y) + sd.weights.z * getJointMatrix(sd.jointIds.z) + sd.weights.w * getJointMatrix(sd.jointIds.w); Vertex v = pcs.inputBuffer.vertices[index]; v.position = vec3(skinMatrix * vec4(v.position, 1.0)); pcs.outputBuffer.vertices[index] = v; I store all joint matrices in a big array and populate it every frame (and also pass the starting index in the array for each skinned mesh, `jointMatricesStartIndex`). The skinning data is not stored inside each mesh vertex, but rather in a separate buffer of `num_vertices` elements. After skinning is performed, this set of vertices is used by all later rendering stages. Thee Thanks to this process, the rendering and game logic are decoupled, which is great. Anton's OpenGL 4 Tutorials book has got the best skinning implementation guide I've seen, by the way. Jason Gregory's Game Engine Architecture also has some nice explanations about skeletal animation math. As for the renderer, it doesn't know anything about entities or "game objects", only lights, scene parameters and meshes to draw. Here's how the renderer's API looks like: void Game::generateDrawList() { renderer.beginDrawing(); // Add lights const auto lights = ...; for (const auto&& [e, tc, lc] : lights.each()) { renderer.addLight(lc.light, tc.transform); } // Render static meshes const auto staticMeshes = ...; for (const auto&& [e, tc, mc] : staticMeshes.each()) { for (std::size_t i = 0; i < mc.meshes.size(); ++i) { renderer.drawMesh(mc.meshes[i], tc.worldTransform, mc.castShadow); } } // Render meshes with skeletal animation const auto skinnedMeshes = ...; for (const auto&& [e, tc, mc, sc] : skinnedMeshes.each()) { renderer.drawSkinnedMesh(mc.meshes, sc.skinnedMeshes, tc.worldTransform, sc.skeletonAnimator.getJointMatrices()); } renderer.endDrawing(); } The draw command is stored in a std::vector and iterated through during the drawing process. Here's what MeshDrawCommand looks like: struct SkinnedMesh { GPUBuffer skinnedVertexBuffer; }; struct MeshDrawCommand { MeshId meshId; glm::mat4 transformMatrix; math::Sphere worldBoundingSphere; const SkinnedMesh* skinnedMesh{nullptr}; std::uint32_t jointMatricesStartIndex; bool castShadow{true}; }; Scene loading and entity prefabs is done with Blender as a level editor. It's easy to place objects, colliders and lights there. Writing own level editor would probably take months (years!), so using Blender instead saved me quite a lot of time. I use node names for spawning some objects. For example, you can see an object named Interact.Sphere.Diary. Prefab names in game development refer to the specific identifiers assigned to each object or model within a project. In this context, the prefab name is denoted by the section before the first dot in a scene hierarchy. For example, "Interact" serves as the prefab name for a particular object. The physics system relies on these prefabs to create sphere-based physics bodies for objects like capsules and boxes. However, more complex models are often avoided directly within level files due to size constraints. Instead, an empty -> arrows object is used with a descriptive name such as "Cat.NearStore", which triggers the spawning of the corresponding prefab and assigns a runtime identifier. Prefabs themselves are documented in JSON format, outlining scene settings, movement properties, and physics configurations. A notable detail is that during level loading, nodes without prefabs are loaded as-is while utilizing their own mesh data from the external GLTF file. In contrast, nodes with prefabs utilize external GLTF files for mesh data while copying only the transform information from the original node in the level's GLTF file. A comparison of rendering techniques is offered to demonstrate the ease of implementing MSAA (Multisampling Anti-Aliasing) using forward rendering. Furthermore, two resources are provided for additional insight into the subject: an article on Multisampling and another detailing potential issues with MSAA implementation, especially in relation to HDR and tone-mapping. The UI system draws inspiration from Roblox's UI API and relies on key concepts such as origin, relative size, and relative position to calculate its own layout without requiring hardcoded elements' positions and sizes. You can also specify the offset for both position and size of elements separately, using terms like `offsetPosition` and `offsetSize`. Additionally, you can set a fixed size for elements to prevent resizing if desired. The label/image element's size is determined by its content. Below are some examples demonstrating how this can be used to position child elements: Firstly, the child (yellow) has relative size (0.5, 1), relative position of (0.5, 0.5), and origin (0.5, 0.5). Its parent (green) will have twice the width but the same height as the child element. The child will be centered inside its parent. Secondly, the child (yellow) has origin (1, 1), fixed size (w,h), and an absolute offset of (x,y). This allows positioning relative to the bottom-right corner of its parent (green). Let's see how sizes and positions are calculated for UI elements in EDBR implementation: First, all element sizes are calculated recursively. Then, positions are computed based on previously calculated sizes and specified offset positions. Afterwards, all elements are drawn recursively - starting with the parent element followed by its children etc. When calculating size, most elements either have a "fixed" size (which can be manually set) or their size is determined by their content. For example, label elements' size is computed using text's bounding box while image elements' size equals the image size and so on. If an element has the "Auto-size" property, it must specify which child to use for calculating its size. An example is when a menu nine-slice contains multiple text labels inside the "vertical layout" element - their bounding boxes are calculated first, then summed up to determine the parent's size. Looking at a simple menu with bounding boxes displayed: The root `NineSliceElement` is marked as "Auto-size". To compute its size, it calculates the child (`ListLayoutElement`) size first. This recursively computes the sizes of each button, sums them up and adds some padding (it also makes the width of each button equal to the maximum width in the list). Regarding Dear ImGui and sRGB issues: Dear ImGui is a great tool for implementing dev and debug tools (take a look at these open in a new tab). However, it has problems with sRGB. I won't go into details, but essentially if you use an sRGB framebuffer, Dear ImGui will appear wrong in many ways (see comparison): Left - naive sRGB fix for Dear ImGui, right - proper fix. Sometimes people do hacks by doing `pow(col, vec4(2.2))` with Dear ImGui's colors but it still doesn't work properly with alpha and produces incorrect color pickers. I ended up writing my own Dear ImGui backend and implementing DilligentEngine's workaround which is explained in detail here and here. It turned out to be less challenging than anticipated. I only needed to implement rendering, as the "logic/OS interaction" portion was already handled by the Dear ImGui SDL backend in my case. Having my own backend brought additional benefits: it supported bindless texture IDs, allowing me to draw images using ImGui::Image with ease. In contrast, the Dear ImGui Vulkan backend required registering textures individually before calling ImGui::Image. My custom backend also enabled proper drawing of linear and non-linear images by passing their formats. It was easier to initialize and manage as well, since it handled Vulkan-related tasks similarly to the rest of my engine. I'd like to briefly mention other unrelated aspects of my engine that aren't specific to Vulkan. For physics, I utilize Jolt Physics, which integrated seamlessly into the engine. I primarily employ it for collision resolution and basic character movement, leveraging fantastic samples and good documentation. Notably, JPH::CharacterVirtual handles character movement exceptionally well. Here's a high-level overview of how Jolt works: you add shapes to the world, run the simulation, and then use the resulting object positions to render them in their updated states. I implemented a physics shape debug renderer using im3d. For the entity-component-system part, I rely on entt, which has worked well so far. Previously, I had my own ECS implementation but chose to experiment with a third-party library to reduce maintenance efforts. For audio, I use openal-soft, libogg, and libvorbis, drawing from these articles: . Tracy helps with profiling; integrating it was straightforward (just read the PDF doc!), and it prevented a lot of unnecessary optimization efforts by revealing the actual time spent on tasks. The benefits I've gained from switching to Vulkan include: abstractions became easier, eliminating the need for complex "shader.bind()" calls and state trackers. The API is more pleasant to work with overall, as it's stateless, explicit, and easier to reason about. With Vulkan, you can write less abstractions overall. In contrast, OpenGL requires more abstractions to reduce error-proneness. Vulkan's API demands fewer abstractions, making it easier to map your code to its raw functions. This leads to better debugging and validation, as seen in the extensive error checking provided by Vulkan. With Vulkan, I can now easily debug shaders in RenderDoc, unlike with OpenGL where I had to output values to a texture and manually inspect them. Vulkan also offers a more consistent experience across different GPUs and operating systems. Unlike OpenGL, where drivers on various GPUs worked differently, resulting in hardware-specific bugs, Vulkan's consistency makes debugging easier. Future shading languages like Slang (or Shady (promise to be more feature-complete and readable, offering opportunities for future exploration. The ability to fine-tune every aspect of the graphics pipeline is another benefit of using Vulkan. This level of control allows me to implement a cleaner system, as seen in my first OpenGL engine, which I rewrote with newfound knowledge and the help of vkguide. Lastly, having a custom Vulkan engine gives me a sense of pride and accomplishment. Future plans include implementing sign-distance field font support, loading multiple images in parallel, and adding features like bloom, volumetric fog, animation blending, render graphs, and ambient occlusion. Who knows? Maybe one day I'll even finish the game... This guide is designed to serve as a foundation for understanding Vulkan, making it easier to work on personal projects. Unlike many other resources that focus on hardcoded rendering loops, this tutorial will emphasize dynamic rendering, allowing it to serve as a more comprehensive base code for game engines. The concepts explored here can also be applied to fields such as CAD and visualization. C++20 is utilized throughout the guide, but its use of advanced features is kept to a minimum, making it possible for developers using C or Rust to follow along. This tutorial assumes prior knowledge of 3D graphics fundamentals, with experience in either OpenGL or DirectX being beneficial. However, an explanation of basic linear algebra concepts used in 3D rendering is not provided. The guide's code is based on Vulkan 1.3 and takes advantage of its features to simplify the engine architecture. A legacy version of the guide is available for support with older standards. The tutorial is structured into multiple chapters for better code organization and covers topics such as setting up initial code, initializing Vulkan and rendering loops, compute shaders, mesh drawing, textures, GLTF scene loading, and high-performance rendering. Additionally, there are extra sections offering further information not directly part of the main tutorial, including content related to Legacy VKGuide and GPU-driven rendering techniques.

How long does it take to learn vulkan. How to use vulkan. How hard is vulkan. How to make vulcan. How to learn vulkan reddit. Should i learn vulkan. How to learn vulkan api. How hard is it to learn vulkan.

- https://www.swaraagmusic.com/public/templateEditor/kcfinder/upload/files/74203706745.pdf
- sahugiso
- foveyojoji
- http://szcftz.com/upload/13538791590.pdf
- https://dalyanestate.com/userfiles/file/79668913758.pdf
- gakosopife
- mercury outboard owners manuals
- http://bannermaul.com/userData/board/file/3945440838.pdf
- relatos de el gato
- diferencias entre modelo lineal y modelo metas
- botanica 3d serum benefits