



REYNEP's Vulkan "Adventure Guide"

Where, you adventure on your own 😊, I only 'guide', showing you the roadmap

Chapter 0: Prerequisites

1. What is Vulkan? Why Vulkan?
 - i. Read the 1. Introduction part from here only 😊
 - a. <https://paminerva.github.io/docs/LearnVulkan/01.A-Hello-Window>
 - b. [TODO:-] Convert to PDF and add a link to that
 - ii. Alternatively:- you can give this page a try too:- <https://vkdoc.net/chapters/fundamentals>
 - iii. What is This?
 - iv. Why should I learn this?
 - v. Why is this Important?
 - vi. When will I need this?
 - vii. How does this work?
2. grab `vulkan-sdk` , `cmake` , `amGHOST`
 - i. <https://vulkan.lunarg.com/sdk/home>
 - make sure `VULKAN_SDK` & `VK_SDK_PATH` environment variables are set
 - ii. <https://cmake.org/download/>
 - [optional] <https://encs.github.io/intro-cmake/hello-cmake/>
 - [optional] OR: Watch 6/7 videos from this playlist:- <https://www.youtube.com/playlist?list=PLK6MXr8gasrGmliSuVQXpfFuE1uPT615s>
 - iii. `git clone -b win32-intro https://github.com/REYNEP/amGHOST`
 - if you don't have `vscode` & `C++ Compiler` --> see [4.guide.vscode.md](#)
 - Open it with VSCode
 - `F1 --> CMake: Configure`
 - `F1 --> CMake: Build`
 - `F1 --> CMake: Install --> .install dir`
 - check's **amGHOST's Usage Example** inside [README.md](#)
 - Option 1 :- use `cmake` for your project too using `add_subdirectory(amGHOST)`
 - Option 2 :- use `libamGHOST.lib` after installing & `#include amGHOST/<header>`
 - just copy paste **amGHOST's Usage Example** into a `main.cpp` for your program
 - now you shall have a OS-Window 😊

4. `vkCreateInstance(CI, &m_instance)`

- <https://vkdoc.net/man/vkCreateInstance>

5. Error Handling / Checking / Logging

- check out my `amVK_log.hh`
 - uses `REY_LoggerNUtils` inside `amGHOST`
 - has a simple `stackTracer()` that i basically stripped from blender3D codebase 💎

6. The Result

- Check out:- [4.guide.chapter1.hh](#)

Overview



We need to create/get hold of a couple of handles:

Instance	1 <code>VkInstance</code> per program/app	<code>VkInstance</code>
Window Surface	<code>Surface(OS-Window)</code> <i>[for actually linking Vulkan-Renders to Screen/Surface]</i>	<code>VkSurfaceKHR</code>
Physical Device	An Actual <i>HARDWARE-GPU-device</i>	<code>VkPhysicalDevice</code>
Queue	<code>Queue(Commands)</code> <i>to be executed on the GPU</i>	<code>VkQueue</code>
Logical Device	The "Logical" GPU Context/Interface (Software Layer)	<code>VkDevice</code>
Swap Chain	<i>Sends Rendered-Image to the</i> <code>Surface(OS-Window)</code> <i>Keeps a backup image-buffer to</i> <code>Render_onto</code>	<code>VkSwapchainKHR</code>



Take a look into this awesome [slide](#) from slide-26 onwards, to understand what each of steps "feel like"/mean/"how to imagine them".

*slide = [Vulkanised 2023 Tutorial Part 1](#)

Chapter 2: `VkDevice`

1. `vkEnumeratePhysicalDevices(m_instance, &m_deviceCount, nullptr)`

- <https://vkdoc.net/man/vkEnumeratePhysicalDevices>

```
uint32_t m_deviceCount = 0;    // [implicit valid usage]:- must be 0    [if 3rd-param =  
nullptr]  
vkEnumeratePhysicalDevices(m_instance, &m_deviceCount, nullptr);  
// it's kinda like the function is 'output-ing into' m_deviceCount  
  
std::vector<VkPhysicalDevice> HardwareGPU_List(gpuCount);  
VkResult return_code = vkEnumeratePhysicalDevices(m_instance, &m_deviceCount,  
HardwareGPU_List.data());
```

Chapter 3: Common Patterns: if someone missed to catch it yet 😊

1. `Types(Structures/Enums)`
 - prefix:- `Vk` -- [e.g. `VkInstanceCreateInfo`]
2. Functions
 - prefix:- `vk` -- [e.g. `vkCreateInstance()`]
3. Preprocessor definitions and enumerators (enumeration values)
 - prefix:- `VK_` -- [e.g. `VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO`]
4. Extensions has extras
 - e.g. `KHR` :- Khronos authored,
 - e.g. `EXT` :- multi-company authored
5. Say you wanna create an `vkZZZ` object.
 - i. First, you take a `VkZZZCreateInfo` --> fill it up
 - ii. Second, you call `vkCreateZZZ()`
 - iii. Lastly, `vkDestroyZZZ()` before closing your app
6. Some objects get '**allocated**' rather than '**created**'
 - `VkZZZAllocateInfo` --> `vkAllocateZZZ` --> `vkFreeZZZ`
7. -- | -- | -- | -----
8. `sType` & `pNext`
 - Many Vulkan structures include these two common fields
9. `sType` :-
 - It may seem somewhat redundant, but this information can be useful for the `vulkan-loader` and actual `gpu-driver-implementations` to know what type of structure was passed in through `pNext` .
10. `pNext` :-
 - *allows to create a linked list between structures.*
 - It is mostly used when dealing with extensions that expose new structures to provide additional information to the `vulkan-loader` , `debugging-validation-layers` , and `gpu-driver-implementations` .
 - *i.e. they can use the `pNext->sType` field to know what's ahead in the linked list*
11. -- | -- | -- | -----
12. Do remember to check the `Valid Usage` section within each manual-page

Two Questions I keep on pondering 🤔

- a) Would this make sense to someone else?
- b) Would this make sense to a 5 year old?