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`?

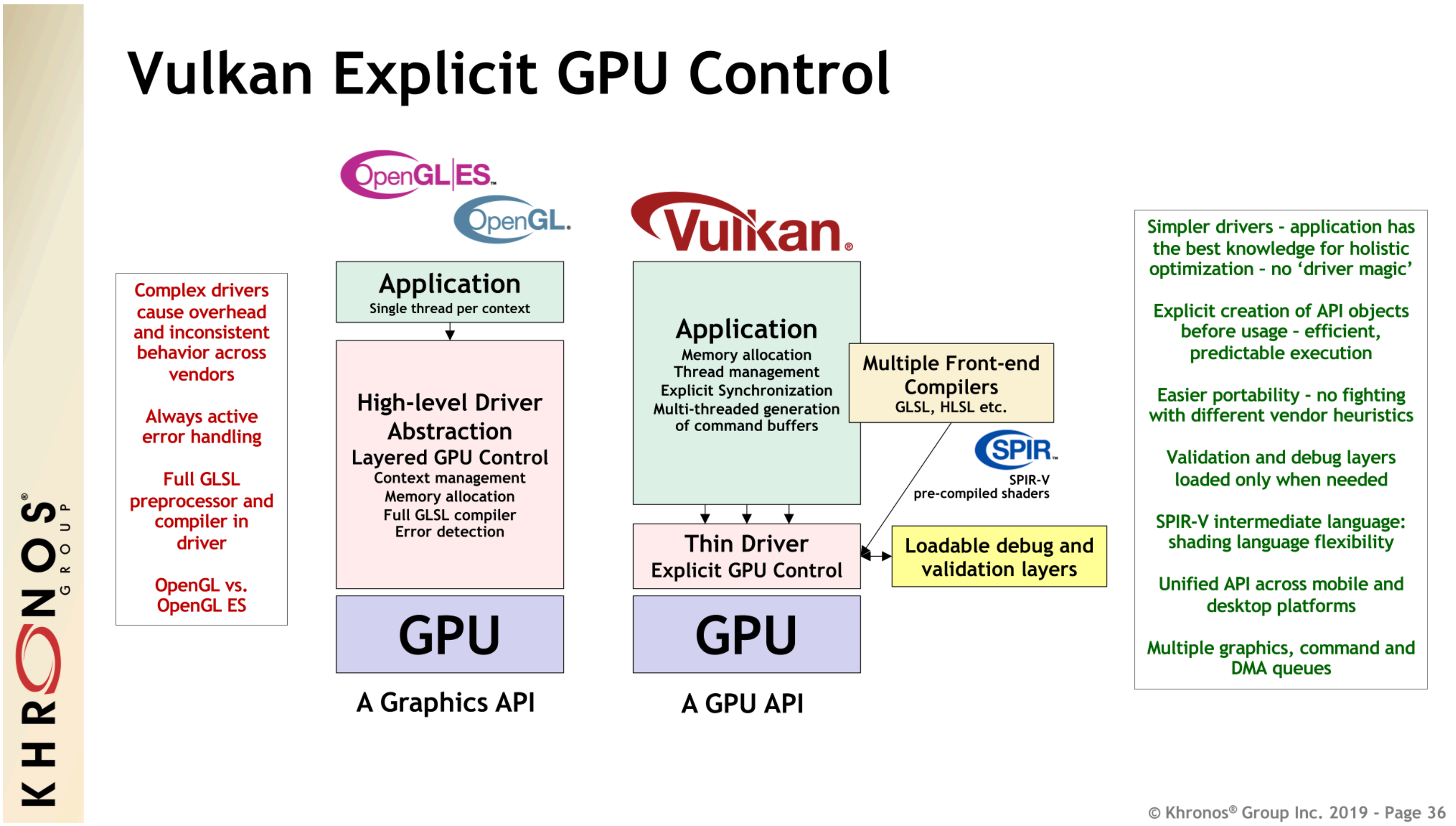
1. Read the `1. Introduction` part from here only 😌
   i. https://paminerva.github.io/docs/LearnVulkan/01.A-Hello-Window
   ii. [TODO:-] Convert (above page) to PDF and add a link to that
2. Alternatively:- you can give this page a try too:- https://vkdoc.net/chapters/fundamentals
3. Why should *'you'* learn/use `Vulkan`?
   i. Faster
   ii. More Control
   iii. Lower Level API
4. Why is this Important?
   i. Well if you are planning on becoming a game dev, then yeah. Otherwise OpenGL is kinda enough.
5. When will I need `vulkan`?
   i. kind of never, unless you've grown tired of OpenGL
6. How does `vulkan` work?
   - Rest of the document is dedicated to answer this question 😌

2. grab `vulkan-sdk`, `cmake`, `amGHOST`

1. https://vulkan.lunarg.com/sdk/home
   - make sure `VULKAN_SDK` & `VK_SDK_PATH` environment variables are set
   - restart vscode after installing
2. https://cmake.org/download/
   - [optional] https://enccs.github.io/intro-cmake/hello-cmake/
   - [optional] OR: Watch 6/7 videos from this playlist:- https://www.youtube.com/playlist?list=PLK6MXr8gasrGmIiSuVQXpfFuE1uPT615s
   - restart vscode after installing
3. if you don't have `vscode` & `C++ Compiler` --> see 4.guide.vscode.md
4. `git clone -b win32-intro https://github.com/REYNEP/amGHOST`
   - Open it with VSCode
   - `F1` --> `CMake: Configure`
   - `F1` --> `CMake: Build`
   - `F1` --> `CMake: Install` --> `.insall` dir
   - check's *amGHOST's Usage Example* inside `amGHOST/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 😌

The Real "Adventure" begins here!

[well, not really. I believe the real adventure is it SHADERs and Algorithms!]



Vulkan Explicit GPU Control

Complex drivers cause overhead and inconsistent behavior across vendors

Always active error handling

Full GLSL preprocessor and compiler in driver

OpenGL vs. OpenGL ES

Simpler drivers - application has the best knowledge for holistic optimization - no 'driver magic'

Explicit creation of API objects before usage - efficient, predictable execution

Easier portability - no fighting with different vendor heuristics

Validation and debug layers loaded only when needed

SPIR-V intermediate language: shading language flexibility

Unified API across mobile and desktop platforms

Multiple graphics, command and DMA queues

© Khronos® Group Inc. 2019 - Page 36

Chapter 1: `VkInstance`

1. `VkApplicationInfo`

- https://vkdoc.net/man/VkApplicationInfo
  - do remember to check the `Valid Usage` section 😉
- yes, what are you waiting for, go go, shooo....
  i. `#include <vulkan/vulkan.h>`
  ii. take an instance of that `Struct` -> Fill it up [😉][have the vkdoc.net as assist]
- REY Docs
  - `VkApplicationInfo` -> holds `name` and `version`, also the `lowest Vulkan API version` Your APP ***"can run"*** on. [*clarification needed:- lowest or highest]
  - Also, we can set the `name` and `version` of the `engine` (if any) used to create Your APP. This can help `vulkan driver implementations` to perform ad-hoc optimizations.
    - e.g. like if a Triple-A [AAA] game used, for say, `Unreal Engine Version 4.1.smth` idk 🤷
  - REFs:- 1. minerva

2. `VkInstanceCreateInfo`

- https://vkdoc.net/man/VkInstanceCreateInfo
    - yeah, do remember to check the `Valid Usage` section 😉
    - Don't hesitate about `EnabledLayer` & `EnabledExtensions` right now
        - come back and add them when you need to
- REY Docs
    - Nothing that I need to add
    - Tho if this section gets big, I will create a separate `.md` file for that thingy

3. `VkInstance m_instance = nullptr;`

- https://vkdoc.net/man/VkInstance
    - again.... yeah, do remember to check the `Valid Usage` section 😉

4. `vkCreateInstance(CI, &m_instance)`

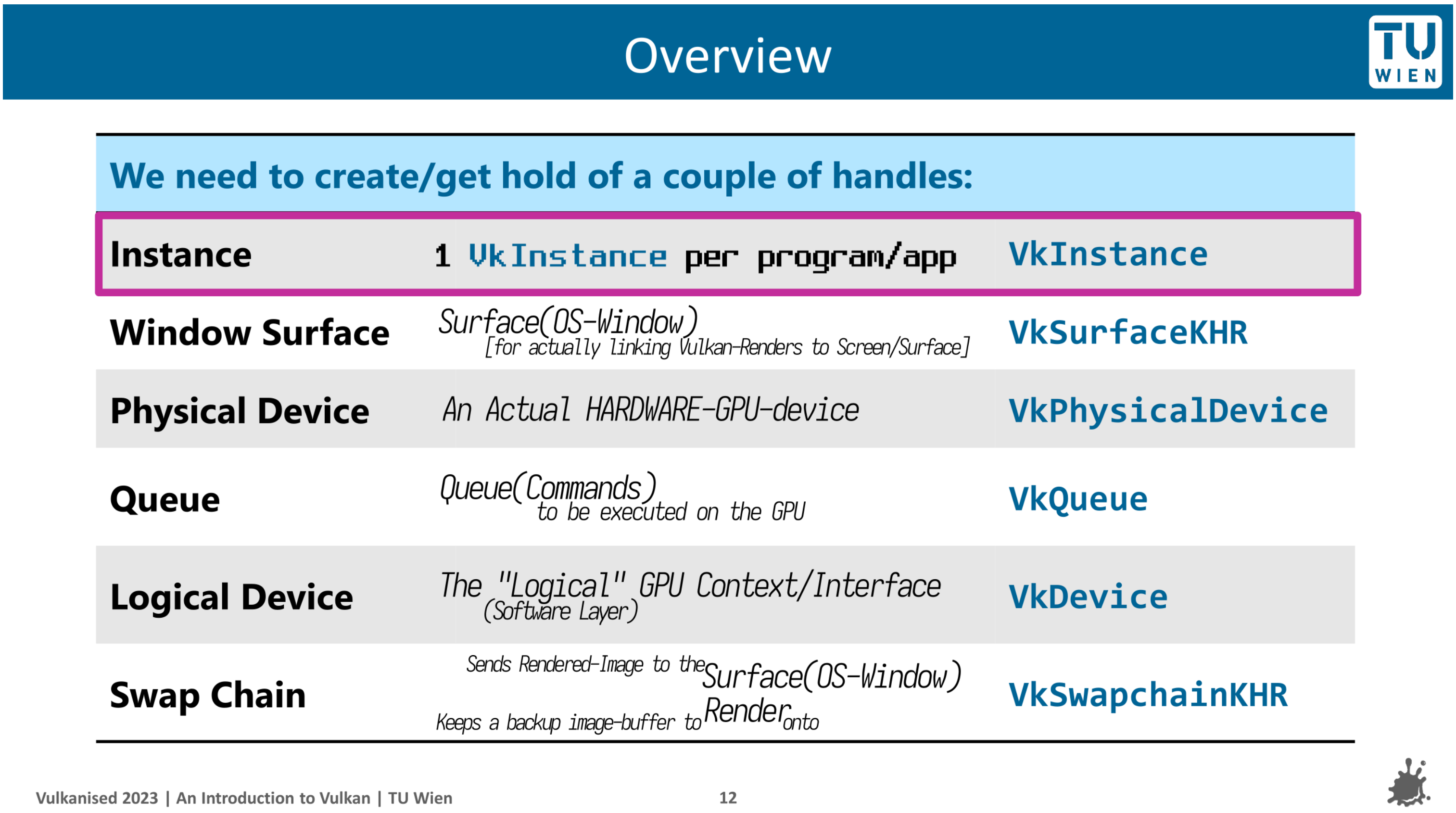
- https://vkdoc.net/man/vkCreateInstance
    - `Valid Usage` section.... (yeah, everytime)

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

Vulkanised 2023 | An Introduction to Vulkan | TU Wien 12

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
   - REY Docs

```
uint32_t deviceCount = 0;
    // [implicit valid usage]:- must be 0 [if 3rd-param = nullptr]
    vkEnumeratePhysicalDevices(m_instance, &deviceCount, nullptr);
        // it's kinda like the function is 'output-ing into' deviceCount

std::vector<VkPhysicalDevice> HardwareGPU_List(gpuCount);
    // best to save this as a class member variable
    vkEnumeratePhysicalDevices(m_instance, &deviceCount, HardwareGPU_List.data());
        // note: it does return     VkResult return_code
```

   - Visualization / [See it] / JSON Printing:- 4.guide.chapter2.1.json.hh
   - So far, The result:- 4.guide.chapter2.1.midway.hh

2. `vkCreateDevice()`
   - https://vkdoc.net/man/vkCreateDevice
     - `param pAllocator` -> **"Chapter ZZZ"**
   - REY DOCs
     - we are not gonna call the `vkCreateDevice()` yeeeet....
       - but, yes, we've already made the class container around it 😁
       - we'll call this functiion in `Chapter2.9.`

    - but we did need to know first about `vkCreateDevice()`
        - because, the idea is, our sole task is to ***fill it up step by step***

3. `VkDeviceCreateInfo`

- https://vkdoc.net/man/VkDeviceCreateInfo
    - `.LayerInfo` -> Deprecated
    - `.ExtensionInfo` -> **"Chapter ZZZ"**
    - `.pQueueCreateInfos` -> next part
        - So far, The result:- 4.guide.chapter2.3.midway.hh
- REY Docs
    - `.pQueueCreateInfos` -> yes, you 'can' mass multiple 😏
    - Sometimes there will be `.zzzCreateInfoCount` & `.pZZZCreateInfos`
        - So you could like pass in an array/vector
        - You will see this in lots of other places

4. `VkDeviceQueueCreateInfo` - *'The Real Deal'*

- https://vkdoc.net/man/VkDeviceQueueCreateInfo
    - `.queueFamilyIndex` -> next 3 subchapters
        - So far, The result:- 4.guide.chapter2.4.midway.hh
- REY Docs:- Support for multiple QCI
    - `.pQueuePriorities` -> yes, this can be multiple "Priorities" 😵 [idk yet why tho]

```
/* ============ REY_LoggerNUtils::REY_Utils.hh ============ */
REY_ArrayDYN<VkDeviceQueueCreateInfo> Array = REY_ArrayDYN<VkDeviceQueueCreateInfo>(2);
    // allocate enough space for 2 elements
REY_ARRAY_PUSH_BACK(Array) = this->Default_QCI;
REY_ARRAY_PUSH_BACK(Array) =          Your_QCI;

/* ============ std::vector ============ */
std::vector<VkDeviceQueueCreateInfo> Array = std::vector<VkDeviceQueueCreateInfo>(2);
Array.push_back(this->Default_QCI);
Array.push_back(         Your_QCI)
```

- So far, The result:- 4.guide.chapter2.4.TheEnd.hh

5. `vkGetPhysicalDeviceQueueFamilyProperties()`

- https://vkdoc.net/man/vkGetPhysicalDeviceQueueFamilyProperties
- REY DOCS
    - a GPU can have *"multiple QueueFamilies"*
        - a `QueueFamily` might support `VK_QUEUE_GRAPHICS_BIT`
        - another `QueueFamily` might support `VK_QUEUE_COMPUTE_BIT`
        - another `QueueFamily` might support `VK_QUEUE_TRANSFER_BIT`
        - another `QueueFamily` might support `VK_QUEUE_VIDEO_ENCODE_BIT_KHR`
        - another `QueueFamily` might support a-mixture of multiple
        - talking about this in -> the next part [chapter2.6.]

```
static inline REY_Array<REY_Array<VkQueueFamilyProperties>>    s_HardwareGPU_QFamProps_List2D;
#define amVK_2D_QFAM_PROPs                     amVK_Instance::s_HardwareGPU_QFamProps_List2D
    // "REY_LoggerNUtils/REY_Utils.hh" 😄

static inline void GetPhysicalDeviceQueueFamilyProperties(void) {
    amVK_2D_QFAM_PROPs.reserve(amVK_GPU_List.n);               // malloc using "new" keyword
    for ( uint32_t k = 0;  k < amVK_GPU_List.n; k++ )         // for each GPU
    {
        REY_Array<VkQueueFamilyProperties> *k_QFamProps = &amVK_2D_QFAM_PROPs.data[k];
        uint32_t queueFamilyCount = 0;
            vkGetPhysicalDeviceQueueFamilyProperties(amVK_GPU_List[k], &queueFamilyCount, nullptr);

        k_QFamProps->n = queueFamilyCount;
        k_QFamProps->data = new VkQueueFamilyProperties[queueFamilyCount];
            vkGetPhysicalDeviceQueueFamilyProperties(amVK_GPU_List[k], &k_QFamProps->n, k_QFamProps->data);
    }
}
```

- Visualization / [See it] / JSON Printing:- 4.guide.chapter2.5.json.hh
    - Check the 3070 JSON by REY
- So far, The result:- 4.guide.chapter2.5.TheEnd.hh
    - Compare to -> 4.guide.chapter2.1.midway.hh
        - `2DArray_QFAM_Props` part & below were added only compared to `Chapter2.1.`

6. `VkQueueFamilyProperties`

- https://vkdoc.net/man/VkQueueFamilyProperties
- REY DOCs
    - `.queueFlags` -> we are gonna choose a `QCI.queueFamilyIndex` based on these flags
        - primarily, for the least, we wanna choose a `QueueFamily` that supports `VK_QUEUE_GRAPHICS_BIT`
        - all kinds of amazing things can be done using
            - `VK_QUEUE_COMPUTE_BIT`
            - `VK_QUEUE_TRANSFER_BIT`
            - `VK_QUEUE_VIDEO_ENCODE_BIT_KHR`
    - `.queueCount` -> yes there is a limit to 'how many `Queues` we are allowed to work with' 🥴

7. `VkDeviceQCI.queueFamilyIndex`

- `QCI => QueueCreateInfo`
    - [VkDeviceQueueCreateInfo]
- REY DOCs
    - `Task:-` is to choose a `QueueFamily` that supports `VK_QUEUE_GRAPHICS_BIT` 😉
        - (if you've followed on so far -> this should be easy 😉)
    - Resolving all of this into `amVK_Device.hh`

```
void amVK_Device::Select_QFAM_GRAPHICS(void) {
    if (!amVK_Instance::called_GetPhysicalDeviceQueueFamilyProperties) {
        amVK_Instance::EnumeratePhysicalDevices();
    }

    if (!amVK_Instance::called_GetPhysicalDeviceQueueFamilyProperties) {
        amVK_Instance::GetPhysicalDeviceQueueFamilyProperties();
    }

    amVK_Instance::amVK_PhysicalDevice_Index index = amVK_HEART->GetARandomPhysicalDevice_amVK_Index();
    this->QCI.Default.queueFamilyIndex = amVK_Instance::ChooseAQueueFamily(VK_QUEUE_GRAPHICS_BIT,
index);
}
```

8. back to `vkCreateDevice()` [finally calling it 😊]

- REY DOCs

```
amVK_Device* D = new amVK_Device(amVK_HEART->GetARandomPhysicalDevice());
    // VkDeviceCreateInfo CI => Class Member
    // VkDeviceQueueCreateInfo QCI => Class Member
D->Select_QFAM_GRAPHICS();
D->CreateDevice();
```

- Think of this as a PSeudoCode / or / check out my code if you wanna

    - `CreateInfo` => By default has initial values inside `amVK_Device`

9. Organizing stuff into classes....

1. `amVK_Props.hh`
    i. `class amVK_Props`
        - `amVK_Instance::GetPhysicalDeviceQueueFamilyProperties()`
        - `amVK_Instance::EnumeratePhysicalDevices()`
        - & Everything related to those two + The Data + The Properties

10. `vkGetPhysicalDeviceProperties()`

- for now we won't need, we will need in `ChapterXXX`
- https://vkdoc.net/man/vkGetPhysicalDeviceProperties
- `VkPhysicalDeviceProperties` :- https://vkdoc.net/man/VkPhysicalDeviceProperties
    - `.deviceType` :- https://vkdoc.net/man/VkPhysicalDeviceType
    - `.limits` :- save it for later 😉
    - you don't need to read the whole documentation of this page

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

```
Object  Vk      VkInstance
Types   Vk      VkInstanceCreateInfo
Funcs   vk      vkCreateInstance()
Enums   VK_     VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO


Extensions
    KHR:- Khronos authored,
    EXT:- multi-company authored


Creating "VkZZZ" object
    1. take `VkZZZCreateInfo` --> fill it up
    2. call `vkCreateZZZ()`
    3. also `vkDestroyZZZ()` before closing your app

    4. Some objects get "allocated" rather than "created"
        `VkZZZAllocateInfo` --> `vkAllocateZZZ` --> `vkFreeZZZ`

    5. Sometimes there will be `.zzzCreateInfoCount` & `.pZZZCreateInfos`
                        e.g. `.queueCreateInfoCount` & `.pQueueCreateInfos``
            -> So you could like pass in an array/vector
            -> You will see this in lots of other places

Getting List/Properties
    1. vkEnumerateZZZ() --> \see `[Chapter2.1.] vkEnumeratePhysicalDevices()` example
```

-- | -- | -- | --

7. `sType` & `pNext`
    - Many Vulkan structures include these two common fields
8. `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` .
9. `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*

-- | -- | -- | --

```
10. 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?
```