Common Mistakes when Learning Vulkan

Charles Giessen, LunarG

Presentation:
Outline

- Basics Usage
- Tooling Tips
- Abstract Advice
- API shenanigans
- Conceptual Conundrums
- Profiling Pro-tips
What is my Vulkan experience?

- Started learning Vulkan in 2017
- Joined the Vulkan Discord in 2018
- Began work at LunarG in 2019
  - Maintain the Vulkan-Loader, vulkaninfo, api_dump, & help with lots more
- Talked with everyone and anyone I could about Vulkan
- Helped countless others learning the API
Who is this talk for?

- People new to Vulkan
- People new to Vulkan and programming
- Some content may not be new to you
  - It was new to you at one point so bears repeating for everyone
Basic Usage
Mistake: It's VulKAN, not VulCAN’T!

- Don’t count yourself out!
- Everybody was in your position
- You KAN do it!
Mistake: Ignoring programming best practices

- Enable warnings in compilers: /W4 in MSVC, -Wall in gcc/clang
- Use Address Sanitizer for C & C++
- Use version control, such as Git
Mistake: Optimizing too early

- Feature first, speed second
- A single triangle is not a serious workload
- “Premature optimization is the root of all evil” - Donald Knuth
- Computers are way faster than you think
  - AAA games have thousands of shaders, pipelines, drawcalls, and more
Mistake: Ignoring VkResult return values

- Ignoring return values often results in crashing in subsequent code
- Error checking macro taken from Vulkan-Samples/framework/common/error.h

```c
#define VK_CHECK(x)
    do {
        VkResult err = x;
        if (err)
            {
                LOGE("Detected Vulkan error: {}", vkb::to_string(err));
                abort();
            }
    } while (0)

// Usage
VK_CHECK(vkEnumerateInstanceExtensionProperties(...));
```
Tooling Tips
Mistake: Ignoring the SDK

- Building every tool is a hassle, save yourself the trouble
- Full of useful tools
  - Validation, shader compilers, shader reflection, Vulkan Configurator, gfxreconstruct, and more!
- More info “Everything you need to know about the Vulkan SDK”
  - Talk was held yesterday
- May need to close & reopen IDE after installing the SDK
Mistake: Not using Vulkan Configurator

- Super easy layer configuration
- Makes many validation settings just a click away
- My goto for using validation, api_dump, gfxreconstruct, disabling layers
Mistake: Ignoring validation errors

- They wouldn’t be called validation errors if they weren’t errors
- Undefined behavior (UB) ensues after all invalid usage
  - UB includes working on your current hardware
- Jeremy’s talk “Using Vulkan Validation Effectively” yesterday for more info
Mistake: Not utilizing educational resources

- [https://docs.vulkan.org](https://docs.vulkan.org)
  - Specification, Guide, Tutorial, & Samples all in one place

- People want to help!
  - [https://discord.com/vulkan](https://discord.com/vulkan)
  - [https://khr.io/slack](https://khr.io/slack)
  - [https://reddit.com/r/vulkan](https://reddit.com/r/vulkan)
Mistake: Not using Graphics Debuggers

- Essential tool in every graphics programmers toolbox

- Many to choose from!
  - Renderdoc (cross platform)
  - Nsight (Nvidia)
  - Radeon Developer Tools Suite (AMD)
  - Intel GPA
  - Android GPU Inspector
Mistake: Ignoring helper libraries

- They exist to help you get things done
- Don’t try to reinvent the wheel
- I recommend Vk-bootstrap & Vulkan Memory Allocator & Volk
- Windowing libraries are essential
  - SDL or GLFW is fine
- Lots of other libraries out there
- Even AAA games uses libraries
Abstract advice
Mistake: Too much abstraction

● The Vulkan API is an abstraction siren, resist its call!
  ○ A bad abstraction is worse than no abstraction

● Vulkan is complex, learn how it works first before trying to abstract it!
  ○ Use libraries & frameworks before creating your own

● A overly abstract renderer makes is difficult to maintain & add onto

● KISS - Keep It Super Simple, Keep It Small and Simple, Keep It Stupidly Simple
Tips for abstracting Vulkan

● Focus on exactly what your application requires

● Don’t support every possible Vulkan feature
  ○ Stick to the features & capabilities you make use of
  ○ Combinatorial explosion of code paths is exponentially more difficult to maintain

● Areas where it’s very easy to go overboard:
  ○ Memory allocation
  ○ Automatic barrier placement
  ○ Descriptor set handling
  ○ Initialization/device selection
  ○ Swapchain resizing
  ○ Resource uploading
Mistake: Falling into Analysis Paralysis

- Don’t let PERFECT be the enemy of GOOD ENOUGH
- Best way to find out the answer is to start trying them out
- If you really can’t decide - pick randomly
- No ‘best’ way, only different ways with different tradeoffs
- Example: Data Uploading
  - Push constants, memory mapped buffers, device local buffers, & more
Mistake: Not throwing code away

- Best Vulkan code I’ve written is code I rewrote countless times
- Code that was difficult to write the first time becomes easier to do
- Good solutions come from knowing the exact problem being solved
  - Hard to know what the real problem is at first
- Good use of version control means code is never really gone
- This isn’t an excuse to write bad code!
Mistake: Assuming you are a solo developer

- The “other” person may be you in the future
- Don’t assume the you of the future will know what the you of today was thinking
- Vulkan has plenty of rules that aren’t obvious from just looking at samples
  - So simple, obvious code to make them apparent
API Shenanigans
Mistake: Not using dynamic viewport & scissor

- Greatly simplifies using pipelines
- No performance penalty
- Supported since launch

```c
VkViewport viewport = {0, 0, 800, 600, 0.0f, 1.0f};
vkCmdSetViewport(command_buffer, 0, 1, &viewport);

VkRect2D scissor = {{0,0}, {800, 600}};
vkCmdSetScissor(command_buffer, 0, 1, &scissor);
```
Mistake: Not using Dynamic Rendering

- Removes VkRenderPass & subpasses
  - The most confusing part of Vulkan 1.0
- From VK_KHR_dynamic_rendering, made core in 1.3
- Makes the API more consistent - but not easier
- VK_KHR_dynamic_rendering_local_read just released!
  - Lets multiple passes stay on tile, which previously required subpasses
Mistake: Not knowing your hardware requirements

- If you aren’t actively testing on a platform, you don’t support that platform
  - Focus on the hardware & OS you have
  - Cross-platform is more work
- If a hardware vendor doesn’t support it, neither should you
- Fallback paths for old hardware is often not worth it
  - 1.3, released in 2022 is widely supported on desktop hardware
Mistake: Prioritizing smooth swapchain resizing

- Doesn’t provide enough benefit for the cost
- Resizing is often slow to begin with, increasing complexity
- Difficult to do when generating a frame is expensive, like in games
- Requires VK_EXT_swapchain_maintenance1 to do it properly
- Resizing can happen anytime, complicates multithreaded renderers
Mistake: Not understanding VkPresentModeKHR

- No perfect mode - know the tradeoffs

<table>
<thead>
<tr>
<th>Present Mode</th>
<th>Tearing</th>
<th>Latency</th>
<th>Drops Frames</th>
<th>Support</th>
<th>Consumes Battery</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Yes</td>
<td>Lower</td>
<td>YES</td>
<td>Almost Guaranteed</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Mailbox</td>
<td>No</td>
<td>Low</td>
<td>Yes</td>
<td>In newer drivers</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>FIFO</td>
<td>No</td>
<td>Highest</td>
<td>No</td>
<td>Guaranteed</td>
<td>Not as much</td>
<td>Good default</td>
</tr>
<tr>
<td>FIFO_relaxed</td>
<td>Yes</td>
<td>High</td>
<td>Yes, but rarely</td>
<td>In newer drivers</td>
<td>Not as much</td>
<td></td>
</tr>
</tbody>
</table>
Mistake: Enabling all features and extensions

- Be explicit about which versions, features, and extensions you use
- Some extensions have significant performance penalties
  - For example, the robustness extensions cost performance
- Blindly enabling things greatly limits cross-platform support
  - May accidentally use features not found on other platforms
- Use Vulkan Profiles as baselines
  - VP_KHR_roadmap_2022 & VP_ANDROID_baseline_2022
Mistake: Forgetting portability extensions

- Enable VK_KHR_portability_enumeration on the instance
- Enable VK_KHR_portability_subset on the device
- Validation will complain otherwise
Mistake: Trying to use all available VkQueues

- First queue supports graphics, compute, and transfer operations
  - This queue can do everything you need*

- Multi-queue can offer performance advantages
  - More difficult implementation wise
  - Example: Requires queue family ownership transfers

* Video encode/decode may require separate queues
Mistake: Calling vkQueueWaitIdle every frame

- Causes the CPU to wait for the GPU to finish before continuing
- Significantly reduces pipelining, if not eliminates it
- And especially don’t call vkDeviceWaitIdle!
  - Same as calling vkQueueWaitIdle on all queues
Conceptual Conundrums
Mistake: Modifying/Destroying objects in use

- “In use” refers to objects referenced by command buffers that are executing
- Anything used by a command buffer is subject to this rule
- Vulkan is asynchronous by nature
  - Think of Vulkan like a remote server
  - Must explicitly sync with fences & timeline semaphores
- Anything with “externally synchronized” makes you handle synchronization
Mistake: Assuming Vulkan is Object Oriented

- Vulkan is an API - doesn’t follow OO paradigms
- Vulkan objects aren’t analogous to OOP objects
- RAII wrappers are difficult to get correct
  - The lifetime of CPU objects doesn’t line up with Vulkan objects implicitly
Advice: Use a deletion queue to manage cleanup

● Deletion queue contains objects & their associated “expiration”
  ○ Centralizes cleanup of old objects
● “In-use” objects that need to be deleted are added to the queue
● Every frame, deletion queue checks which objects have expired
  ○ Checks can be a fence, timeline semaphore, or similar
  ○ Calls appropriate cleanup calls on each object, such as vkDestroyImage()
Mistake: Misunderstanding Frames in Flight

- Concept CPU and GPU to work on different frames at the same time
  - Often called “Double buffering” or “multi-buffering”
- It is NOT equal to swapchain image count
- 2 Frames in flight is fine
- Only need to duplicate resources that are written by CPU & read by GPU
  - Depth buffer does not need double buffering
Mistake: Waiting to add a GUI

- Reduces development time of graphics features
- Quickly change what is being shown
  - Rather than have to close & reload application
- Dear ImGUI is a great choice
  - But many other options are available
Profiling Pro-tips
Mistake: Not measuring

- Don’t think, measure
- Deciding things without measuring is like no better than guessing
  - It’s like scientists using “feelings” instead of “facts”
- Don’t waste your time optimizing things that aren’t slow
- Understand Amdahl’s law
  - "The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used"
Mistake: Not using a profiler

- Gives a wealth of information
  - Drawcall execution time, memory pressure, register spilling, occupancy, & more

- Many great profilers exist - for example:
  - AMD Radeon GPU Profiler
  - Android GPU Inspector (AGI)
  - ARM Mobile Studio & PerfDoc
  - Nvidia Nsight Tools
  - Qualcomm Snapdragon Profiler
  - Tracy Profiler - cross vendor

- Bonus Mistake: Renderdoc isn’t a profiler -
Mistake: Not following vendor Best Practices

- Vendors want you to succeed
- Best practices from vendors are available in the validation layer
- Written guides also are available
  - AMD: https://gpuopen.com/performance/
  - Nvidia: https://developer.nvidia.com/blog/vulkan-dos-donts/
Mistake: Using Frames Per Second

- Use milliseconds instead!
- FPS doesn’t measure time
  - Δ FPS is meaningless without context
  - Δ ms means the same thing always
- Milliseconds is linear & more granular
- Profilers give you ms, not FPS
Recap
Recap

- Use the validation layer, debuggers, profilers, & other tools
- Use all available resources - educational, people, and libraries
- Don’t overthink it - KISS again
- Don’t think you need to use every Vulkan feature
- Measure, then act
Help Us Improve the Vulkan SDK and Ecosystem

Share Your Feedback
Take the LunarG annual developer’s survey
https://www.surveymonkey.com/r/KTBZDCM

- Survey results are tabulated
- Shared with the Vulkan Working Group
- Actions are assigned
- Results are reported

Survey closes February 26, 2024

Thank you!

QUESTIONS?

Today’s Presentation:

Get A FREE Tumbler at the LunarG Sponsor Table!