Vulkan notes

August Heining Holst

June 27, 2022

1 What it takes to draw a triangle

1.1 Instance and physical device selection

A Vulkan program starts by setting up the Vulkan API through `VkInstance`.
An instance is made based on which API extensions will be used.
With the instance you can query for Vulkan and select on or more `VkPhysicalDevice`*s* to use.

1.2 Logical Device and queue families

After selecting the hardware device to use, create a `VkDevice` (Logical device) describing which `VkPhysicalDeviceFeatures` will be used. Here a queue family will be specified.
Most Vulkan operation are asynchronously executed from a `VkQueue`. A Queue family pertains to the type of operations, I.E. graphics, memory and computation.

1.3 Window surface and swap chain

Creating a window is done natively or using something like GLFW or SDL.
Rendering to window requires two components: A window surface `VkSurfaceKHR` and a swap chain `VkSwapchainKHR`.
The swap chain is a collection of render targets, whose purpose is to hold and swap the image buffers for smooth realtime rendering. A swap chain has a present mode, common ones are double and triple buffering.

Some platforms allow direct rendering without any window manager through the extensions `VK_KHR_display` and `VK_KHR_display_swapchain`.

1.4 Image views and framebuffers

To draw a frame from the swap chain, it has to be wrapped into a `VkImageView` and `VkFramebuffer`.
Image view reference a specific part of an image to be used.
A framebuffer reference image views that are used for color, depth and stencil targets.

1.5 Render passes

A render pass in Vulkan describe the type of image used during a rendering operation. For example telling Vulkan a which solid color to clear the screen with, before any rendering.
A `VkFramebuffer` binds the specific render pass image.

1.6 Graphics pipeline

The `VkPipeline` object describes the configurable state of the graphics card, like viewport size, depth buffer operation and state using `VkShaderModule` objects.
Vulkan requires almost all configuration of the pipeline in advance, so swtiching shaders requires management of multiple `VkPipeline` objects.
Only some basic confugarations, like viewport and clear color can be changed dynamically.

1.7 Command pools and commnad buffers

Queued operations are recorded into a `VkCommandBuffer`. These command buffers are allocated from a `VkCommandPool`, a command pool is associated with a specific queue familiy.
So drawing a triangle, the command buffer holds the following operations.

- Begin the render pass.
- Bind the graphics pipeline.

- Draw 3 vertices.
- End the render pass.

The correct command buffer need to be selected, before rendering whatever image the swap chain returns.

1.8 Main loop

The main loop first acquires an image from the swap chain using `VkAcquireNextImageKHR`. Then the appropriate coomand buffer and execute with `VkQueueSubmit`. Finally return the image to swap chain for presention with `VkQueuePresentKHR`.

Queue operations are executed asynchronously and therefore requires semaphore to ensure correct order of execution.
Firstly, the image must be acquired, as not to modify the currently rendered frame.
Secondly, the rendered frame must be finished before presenting it.

1.9 Summary

In short, to draw a triangle using Vulkan, one needs to:

1. Create a `VkInstance`
2. Select a supported graphics card (VkPhysicalDevice)
3. Create a `VkDevice` and `VkQueue` for drawing and presentation.
4. Create a window, window surface and swap chain.
5. Wrap the swap chain images into `VkImageview`
6. Create a render pass that specifies the render targets and usage.
7. Create framebuffers for the render pass.
8. Set up the graphics pipeline.
9. Allocaate and record a command buffer with draw commands for every possible swap chain image.
10. Draw frames by acquring images, submitting the right draw command buffer and persenting the images through the swap chain.

2 Coding conventions for Vulkan

All Vulkan function, enums and structs are defined in the header `vulkan.h`, with the following naming convention.

- Functions: Lower case `vk` prefix.
- Enums and structs: Upper case `Vk` prefix.
- Enum values: All caps `VK_` prefix.

The API uses a lot of structs to provide parameters for functions. Two important fields are `sType` and `pNext`. Structure type and extensions structure (`nullptr` if none)
Funcions that create or destroy objects will have a `VkAllocationCallbacks` parameter, allowing for custom driver memory (Can be left as nullptr).

Generally object creation adheres to the following pattern:

```
VkXXXCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_XXX_CREATE_INFO;
createInfo.pNext = nullptr;
createInfo.foo = ...;
createInfo.bar = ...;

VkXXX object;
if (vkCreateXXX(&createInfo, nullptr, &object) != VK_SUCCESS) {
    std::cerr << "failed to create object" << std::endl;
    return false;
}
```

3 Development environment

3.1 Linux

On Arch Linux, you can run `sudo pacman -S vulkan-devel` to install the following tools:

- `vulkaninfo` profiler and such.

- `vkcube` renders a small cube, quick for testing.
- `Vulkan loader` looks up functions in the driver at runtime, similar to OpenGL's GLEW.
- `standard validation layers` debugging stuff
- `SPIR-V tools` idk