Vulkan API Concepts

Krisztián Szabó

June 29, 2024

Contents

0 References

This documentation was written with the help of the following sources:

- Vulkan Specification
- Vulkan Guide

1 Swapchain

A swapchain object (a.k.a. swapchain) provides the ability to present rendering results to a surface. Swapchain objects are represented by `VkSwapchainKHR` handles. A swapchain is an abstraction for an array of presentable images that are associated with a surface. The presentable images are represented by `VkImage` objects created by the platform. One image (which can be an array image for multiview/stereoscopic-3D surfaces) is displayed at a time, but multiple images can be queued for presentation. An application renders to the image, and then queues the image for presentation to the surface.

When creating a swapchain we have to specify a lot of pieces of data. One of the most important option is the present mode. To describe the functionality of these options, first let us introduce some definitions.

- vertical blank: typically refers to the period during which the display device (such as a monitor) refreshes its image. This process is often referred to as Vertical Blank (VBlank) or Vertical Sync (VSync).
- tearing: Tearing is a visual artifact in video display where a frame is displayed on the screen before the previous frame has finished being drawn, causing parts of multiple frames to be shown at once.

There are four options that are important for us.

1. `VK_PRESENT_MODE_IMMEDIATE_KHR`: specifies that the presentation engine does not wait for a vertical blanking period to update the current image, meaning this mode may result in visible tearing. No internal queuing of presentation requests is needed, as the requests are applied immediately.
2. `VK_PRESENT_MODE_MAILBOX_KHR`: specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing cannot be observed. An internal single-entry queue is used to hold pending presentation requests. If the queue is full when a new presentation request is received, the new request replaces the existing entry, and any images associated with the prior entry become available for reuse by the application. One request is removed from the queue and processed during each vertical blanking period in which the queue is non-empty.
3. `VK_PRESENT_MODE_FIFO_KHR`: specifies that the presentation engine waits for the next vertical blanking period to update the current image. Tearing cannot be observed. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during each vertical blanking period in which the queue is non-empty. This is the only value of presentMode that is required to be supported.
4. `VK_PRESENT_MODE_FIFO_RELAXED_KHR`: specifies that the presentation engine generally waits for the next vertical blanking period to update the current image. If a vertical blanking period has already passed since the last update of the current image then the

presentation engine does not wait for another vertical blanking period for the update, meaning this mode may result in visible tearing in this case. This mode is useful for reducing visual stutter with an application that will mostly present a new image before the next vertical blanking period, but may occasionally be late, and present a new image just after the next vertical blanking period. An internal queue is used to hold pending presentation requests. New requests are appended to the end of the queue, and one request is removed from the beginning of the queue and processed during or after each vertical blanking period in which the queue is non-empty.

2 Making abstractions

2.1 Object creations

Vulkan utilizes object creations and deletions to make our engine work. We have to be very explicit when it comes to creating an object. In most of the cases, we do it by specifying an object creation struct and pass it to some functions. This way Vulkan can determine the state of an object creation. There is an example below:

```
void App::CreateInstanceObject()
{
    VkApplicationInfo appCI{};
    appCI.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appCI.pNext = nullptr;
    appCI.pApplicationName = "Vulkan Application";
    appCI.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    appCI.pEngineName = "No Engine";
    appCI.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    appCI.apiVersion = VK_MAKE_VERSION(1, 1, 0);

    VkInstanceCreateInfo instanceCI{};
    instanceCI.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
    instanceCI.pNext = &mDebugMessengerCreateInfo;
    instanceCI.flags = 0;
    instanceCI.pApplicationInfo = &mAppCreateInfo;
    instanceCI.enabledLayerCount = 0;
    instanceCI.ppEnabledLayerNames = nullptr;
    instanceCI.enabledExtensionCount = 0;
    instanceCI.ppEnabledExtensionNames = nullptr;

    VkInstance instance;
    if (vkCreateInstance(&instanceCI, nullptr, &instance))
    {
        throw std::runtime_error("instance creation failure!");
    }
}
```

In most of the cases, we only want to specify just a few options. This can be simplified in a variaty of ways, for instance using pre-written functions returning pre-filled structs.

```
inline VkMemoryBarrier memoryBarrier()
{
    VkMemoryBarrier memoryBarrier {};
    memoryBarrier.sType = VK_STRUCTURE_TYPE_MEMORY_BARRIER;
    return memoryBarrier;
}
```

Another approach is to use the builder design pattern for such object creations.

```cpp
class InstanceBuilder
{
public:
    InstanceBuilder& EnableDebugger();
    InstanceBuilder& SetAppName(const std::string& appName);
    InstanceBuilder& SetAppVersion(uint32_t verson);
    InstanceBuilder& SetEngineName(const std::string& appName);
    InstanceBuilder& SetEngineVersion(uint32_t verson);
    InstanceBuilder& SetAPIVersion(uint32_t version);
    InstanceBuilder& EnableLayers(const char** layerNames);
    InstanceBuilder& EnableExtensions(const char** extensions);
    InstanceBuilder& Build();
    VkInstance GetInstance();

private:
    VkInstance instance;
};

InstanceBuilder builder;
VkInstance instance = builder.EnableDebugger()
                             .SetAppName("Vulkan")
                             .SetAppVersion(VK_MAKE_VERSION(1,0,0))
                             .SetEngineName("No Engine")
                             .SetEngineVersion(VK_MAKE_VERSION(1,0,0))
                             .SetAPIVersion(VK_MAKE_VERSION(1,1,0))
                             .Build()
                             .GetInstance();
```

3 Descriptors

3.1 What are descriptors?

Think of a single descriptor as a handle or pointer into a resource. That resource being a Buffer or a Image, and also holds other information, such as the size of the buffer, or the type of sampler if it's for an image. A `VkDescriptorSet` is a pack of those pointers that are bound together. Vulkan does not allow you to bind individual resources in shaders. They have to be grouped in the sets. If you still insist on being able to bind them individually, then you will need a descriptor set for each resource. This is very inefficient and won't work in many hardware. If you look at this Chart, you will see that some devices will only allow up to 4 descriptor sets to be bound to a given pipeline, on PC. Due to this, we can really only use up to 4 descriptor sets in our pipelines if we want the engine to run on Intel integrated GPUs. A common and performant way of dealing with that limitation of 4 descriptors, is to group them by binding frequency.

3.2 Descriptor allocation

Descriptor sets have to be allocated directly by the engine from a `VkDescriptorPool`. A descriptor set allocation will typically be allocated in a section of GPU VRAM. Once a descriptor set is allocated, you need to write it to make it point into your buffers/textures. Once you bind a descriptor set and use it in a `vkCmdDraw()` function, you can no longer modify it unless you specify the `VK_DESCRIPTOR_POOL_CREATE_UPDATE_AFTER_BIND_BIT` flag. When a descriptor pool is allocated, you have to tell the driver how many descriptors sets, and what number of resources you will be using.

Allocating descriptor sets can be very cheap if you explicitly disallow freeing individual sets by not setting the `VK_DESCRIPTOR_POOL_CREATE_FREE_DESCRIPTOR_SET_BIT` flag. By using that flag, you are telling the driver that you want descriptors to be able to deallocate individually. If you are allocating descriptor sets per frame, you should not be using that, and then you reset the entire pool instead of individual descriptor sets. For your global descriptor sets, it's fine to allocate them once, and reuse them from frame to frame.

3.3 Writing descriptors

A freshly allocated descriptor set is just a bit of GPU memory, you need to make it point to your buffers. For that you use `vkUpdateDescriptorSets()`, which takes an array of `VkWriteDescriptorSet` for each of the resources that a descriptor set points to. If you were using the Update After Bind flag, it is possible to use descriptor sets, and bind them in command buffers, and update it right before submitting the command buffer. This is mostly a niche use case, and not commonly used. You can only update a descriptor set before it's bound for the first time, unless you use that flag, in which case you can only update it before you submit the command buffer into a queue. When a descriptor set is being used, it's immutable, and trying to update it will cause errors. The validation layers catch that. To be able to update the descriptor sets again, you need to wait until the command has finished executing.

3.4 Binding descriptors

Descriptor sets bind into specific "slots" on a Vulkan pipeline. When creating a pipeline, you have to specify the layouts for each of the descriptor sets that can be bound to the pipeline. This is commonly done automatically, generated from reflection on the shader. We will be doing it manually to show how it's done. Once you bind a pipeline in a command buffer, the pipeline has slots for the different descriptor sets, and then you can bind a set into each of the slots. If the descriptor set doesn't match the slot, there will be errors. If you bind a descriptor set to slot 0, and then you switch pipelines by binding another one, the descriptor set will stay bound, IF the slot is the same on the new pipeline. If the slot isn't exactly the same, then the slot will be "unbound", and you need to bind it again. For example, let's say we have 2 pipelines, one of which has a descriptor set 0 that binds to a buffer, and descriptor set 1 that binds to 4 images. Then the other pipeline has descriptor set 0 that binds to a buffer (same as the same slot in the other pipeline), but in descriptor set 1 it has a descriptor set that binds to 3 images, not 4. If you bind the second pipeline, the descriptor set 0 will stay bound, but the descriptor 1 will be unbound because it no longer matches. This is why we assign a frequency to the descriptor slots, to minimize binding.

3.5 Descriptor set layout

Used in both the pipelines and when allocating descriptors, a `VkDescriptorSetLayout` is the shape of the descriptor. For example, a possible layout will be one where it binds 2 buffers and 1 image. When creating pipelines or allocating the descriptor sets themselves, you have to use the layout. Descriptor set layouts can be compatible if they are the same even if they are created in two different places.

3.6 Uniform buffers

Descriptor sets point into buffers, but we didn't explain that. Right now we are creating GPU buffers that hold vertex data, but you can also create buffers that hold arbitrary data for you to use in the shaders. For that type of data, Uniform Buffers are the common thing. They are small size (up to a few kilobytes), but are very fast to read, so they are perfect for shader parameters. By creating a Uniform Buffer and writing to it from the CPU, you can send data to the GPU in a much more efficient way than push constants. We will be using it for the camera information. It is possible to have multiple descriptor sets pointing to one uniform buffer, and it's also possible to have a big uniform buffer, and then each descriptor sets point to a section of the buffer. The shader will not know the difference.