Logical device and queues

Introduction

After selecting a physical device to use we need to set up a *logical device* to interface with it. The logical device creation process is similar to the instance creation process and describes the features we want to use. We also need to specify which queues to create now that we've queried which queue families are available. You can even create multiple logical devices from the same physical device if you have varying requirements.

Start by adding a new class member to store the logical device handle in.

```
VkDevice device;
```

Next, add a `createLogicalDevice` function that is called from `initVulkan`.

C++ |

```
void initVulkan() {
    createInstance();
    setupDebugMessenger();
    pickPhysicalDevice();
    createLogicalDevice();
}

void createLogicalDevice() {

}
```

Specifying the queues to be created

The creation of a logical device involves specifying a bunch of details in structs again, of which the first one will be `VkDeviceQueueCreateInfo`. This structure describes the number of queues we want for a single queue family. Right now we're only interested in a queue with graphics capabilities.

```
QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

VkDeviceQueueCreateInfo queueCreateInfo{};
queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
queueCreateInfo.queueFamilyIndex = indices.graphicsFamily.value();
queueCreateInfo.queueCount = 1;
```

The currently available drivers will only allow you to create a small number of queues for each queue family and you don't really need more than one. That's because you can create all of the command buffers on multiple threads and then submit them all at once on the main thread with a single low-overhead call.

Vulkan lets you assign priorities to queues to influence the scheduling of command buffer execution using floating point numbers between `0.0` and `1.0`. This is required even if there is only a single queue:

```
float queuePriority = 1.0f;
queueCreateInfo.pQueuePriorities = &queuePriority;
```

Specifying used device features

The next information to specify is the set of device features that we'll be using. These are the features that we queried support for with `vkGetPhysicalDeviceFeatures` in the previous chapter, like geometry shaders. Right now we don't need anything special, so we can simply define it and leave everything to `VK_FALSE`. We'll come back to this structure once we're about to start doing more interesting things with Vulkan.

```
VkPhysicalDeviceFeatures deviceFeatures{};
```

Creating the logical device

With the previous two structures in place, we can start filling in the main `VkDeviceCreateInfo` structure.

```
VkDeviceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
```

First add pointers to the queue creation info and device features structs:

```
createInfo.pQueueCreateInfos = &queueCreateInfo;
createInfo.queueCreateInfoCount = 1;

createInfo.pEnabledFeatures = &deviceFeatures;
```

The remainder of the information bears a resemblance to the `VkInstanceCreateInfo` struct and requires you to specify extensions and validation layers. The difference is that these are device specific this time.

An example of a device specific extension is `VK_KHR_swapchain`, which allows you to present rendered images from that device to windows. It is possible that there are Vulkan devices in the system that lack this ability, for example because they only support compute operations. We will come back to this extension in the swap chain chapter.

Previous implementations of Vulkan made a distinction between instance and device specific validation layers, but this is no longer the case. That means that the `enabledLayerCount` and `ppEnabledLayerNames` fields of `VkDeviceCreateInfo` are ignored by up-to-date implementations. However, it is still a good idea to set them anyway to be compatible with older implementations:

```
createInfo.enabledExtensionCount = 0;

if (enableValidationLayers) {
    createInfo.enabledLayerCount = static_cast<uint32_t>
(validationLayers.size());
    createInfo.ppEnabledLayerNames = validationLayers.data();
} else {
    createInfo.enabledLayerCount = 0;
}
```

We won't need any device specific extensions for now.

That's it, we're now ready to instantiate the logical device with a call to the appropriately named `vkCreateDevice` function.

```
if (vkCreateDevice(physicalDevice, &createInfo, nullptr, &device) != VK_SUCCESS)
{
    throw std::runtime_error("failed to create logical device!");
}
```

The parameters are the physical device to interface with, the queue and usage info we just specified, the optional allocation callbacks pointer and a pointer to a variable to store the logical device handle in. Similarly to the instance creation function, this call can return errors based on enabling non-existent extensions or specifying the desired usage of unsupported features.

The device should be destroyed in `cleanup` with the `vkDestroyDevice` function:

```
void cleanup() {
    vkDestroyDevice(device, nullptr);
    ...
}
```

Logical devices don't interact directly with instances, which is why it's not included as a parameter.

Retrieving queue handles

The queues are automatically created along with the logical device, but we don't have a handle to interface with them yet. First add a class member to store a handle to the graphics queue:

```
VkQueue graphicsQueue;
```

Device queues are implicitly cleaned up when the device is destroyed, so we don't need to do anything in `cleanup`.

We can use the `vkGetDeviceQueue` function to retrieve queue handles for each queue family. The parameters are the logical device, queue family, queue index and a pointer to the variable to store the queue handle in. Because we're only creating a single queue from this family, we'll simply use index `0`.

```
vkGetDeviceQueue(device, indices.graphicsFamily.value(), 0, &graphicsQueue);
```

With the logical device and queue handles we can now actually start using the graphics card to do things! In the next few chapters we'll set up the resources to present results to the window system.

C++ code