Physical devices and queue families

Selecting a physical device

After initializing the Vulkan library through a VkInstance we need to look for and select a graphics card in the system that supports the features we need. In fact we can select any number of graphics cards and use them simultaneously, but in this tutorial we'll stick to the first graphics card that suits our needs.

We'll add a function `pickPhysicalDevice` and add a call to it in the `initVulkan` function.

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

void pickPhysicalDevice() {

}
```

The graphics card that we'll end up selecting will be stored in a VkPhysicalDevice handle that is added as a new class member. This object will be implicitly destroyed when the VkInstance is destroyed, so we won't need to do anything new in the `cleanup` function.

```
VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
```

Listing the graphics cards is very similar to listing extensions and starts with querying just the number.

```
uint32_t deviceCount = 0;
vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);
```

If there are 0 devices with Vulkan support then there is no point going further.

```
if (deviceCount == 0) {
    throw std::runtime_error("failed to find GPUs with Vulkan support!");
}
```

Otherwise we can now allocate an array to hold all of the VkPhysicalDevice handles.

```
std::vector<VkPhysicalDevice> devices(deviceCount);
vkEnumeratePhysicalDevices(instance, &deviceCount, devices.data());
```

Now we need to evaluate each of them and check if they are suitable for the operations we want to perform, because not all graphics cards are created equal. For that we'll introduce a new function:

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    return true;
}
```

And we'll check if any of the physical devices meet the requirements that we'll add to that function.

```
for (const auto& device : devices) {
    if (isDeviceSuitable(device)) {
        physicalDevice = device;
        break;
    }
}

if (physicalDevice == VK_NULL_HANDLE) {
    throw std::runtime_error("failed to find a suitable GPU!");
}
```

The next section will introduce the first requirements that we'll check for in the `isDeviceSuitable` function. As we'll start using more Vulkan features in the later chapters we will also extend this function to include more checks.

Base device suitability checks

To evaluate the suitability of a device we can start by querying for some details. Basic device properties like the name, type and supported Vulkan version can be queried using vkGetPhysicalDeviceProperties.

```
VkPhysicalDeviceProperties deviceProperties;
vkGetPhysicalDeviceProperties(device, &deviceProperties);
```

The support for optional features like texture compression, 64 bit floats and multi viewport rendering (useful for VR) can be queried using vkGetPhysicalDeviceFeatures:

```
VkPhysicalDeviceFeatures deviceFeatures;
vkGetPhysicalDeviceFeatures(device, &deviceFeatures);
```

There are more details that can be queried from devices that we'll discuss later concerning device memory and queue families (see the next section).

As an example, let's say we consider our application only usable for dedicated graphics cards that support geometry shaders. Then the `isDeviceSuitable` function would look like this:

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    VkPhysicalDeviceProperties deviceProperties;
    VkPhysicalDeviceFeatures deviceFeatures;
    vkGetPhysicalDeviceProperties(device, &deviceProperties);
    vkGetPhysicalDeviceFeatures(device, &deviceFeatures);

    return deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU
&&
           deviceFeatures.geometryShader;
}
```

Instead of just checking if a device is suitable or not and going with the first one, you could also give each device a score and pick the highest one. That way you could favor a dedicated graphics card by giving it a higher score, but fall back to an integrated GPU if that's the only available one. You could implement something like that as follows:

```
#include <map>

...

void pickPhysicalDevice() {
    ...

    // Use an ordered map to automatically sort candidates by increasing score
    std::multimap<int, VkPhysicalDevice> candidates;

    for (const auto& device : devices) {
        int score = rateDeviceSuitability(device);
        candidates.insert(std::make_pair(score, device));
    }

    // Check if the best candidate is suitable at all
    if (candidates.rbegin()->first > 0) {
        physicalDevice = candidates.rbegin()->second;
    } else {
        throw std::runtime_error("failed to find a suitable GPU!");
    }
}

int rateDeviceSuitability(VkPhysicalDevice device) {
    ...

    int score = 0;

    // Discrete GPUs have a significant performance advantage
    if (deviceProperties.deviceType == VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU) {
```

```
        score += 1000;
    }

    // Maximum possible size of textures affects graphics quality
    score += deviceProperties.limits.maxImageDimension2D;

    // Application can't function without geometry shaders
    if (!deviceFeatures.geometryShader) {
        return 0;
    }

    return score;
}
```

You don't need to implement all that for this tutorial, but it's to give you an idea of how you could design your device selection process. Of course you can also just display the names of the choices and allow the user to select.

Because we're just starting out, Vulkan support is the only thing we need and therefore we'll settle for just any GPU:

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    return true;
}
```

In the next section we'll discuss the first real required feature to check for.

Queue families

It has been briefly touched upon before that almost every operation in Vulkan, anything from drawing to uploading textures, requires commands to be submitted to a queue. There are different types of queues that originate from different *queue families* and each family of queues allows only a subset of commands. For example, there could be a queue family that only allows processing of compute commands or one that only allows memory transfer related commands.

We need to check which queue families are supported by the device and which one of these supports the commands that we want to use. For that purpose we'll add a new function `findQueueFamilies` that looks for all the queue families we need.

Right now we are only going to look for a queue that supports graphics commands, so the function could look like this:

```
uint32_t findQueueFamilies(VkPhysicalDevice device) {
    // Logic to find graphics queue family
}
```

However, in one of the next chapters we're already going to look for yet another queue, so it's better to prepare for that and bundle the indices into a struct:

```
struct QueueFamilyIndices {
    uint32_t graphicsFamily;
};

QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
    QueueFamilyIndices indices;
    // Logic to find queue family indices to populate struct with
    return indices;
}
```

But what if a queue family is not available? We could throw an exception in `findQueueFamilies`, but this function is not really the right place to make decisions about device suitability. For example, we may *prefer* devices with a dedicated transfer queue family, but not require it. Therefore we need some way of indicating whether a particular queue family was found.

It's not really possible to use a magic value to indicate the nonexistence of a queue family, since any value of `uint32_t` could in theory be a valid queue family index including `0`. Luckily C++17 introduced a data structure to distinguish between the case of a value existing or not:

```
#include <optional>

...

std::optional<uint32_t> graphicsFamily;

std::cout << std::boolalpha << graphicsFamily.has_value() << std::endl; // false

graphicsFamily = 0;

std::cout << std::boolalpha << graphicsFamily.has_value() << std::endl; // true
```

`std::optional` is a wrapper that contains no value until you assign something to it. At any point you can query if it contains a value or not by calling its `has_value()` member function. That means that we can change the logic to:

```
#include <optional>

...

struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;
};

QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
```

```
    QueueFamilyIndices indices;
    // Assign index to queue families that could be found
    return indices;
}
```

We can now begin to actually implement `findQueueFamilies`:

```
QueueFamilyIndices findQueueFamilies(VkPhysicalDevice device) {
    QueueFamilyIndices indices;

    ...

    return indices;
}
```

The process of retrieving the list of queue families is exactly what you expect and uses `vkGetPhysicalDeviceQueueFamilyProperties`:

```
uint32_t queueFamilyCount = 0;
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount, nullptr);

std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount);
vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount,
queueFamilies.data());
```

The VkQueueFamilyProperties struct contains some details about the queue family, including the type of operations that are supported and the number of queues that can be created based on that family. We need to find at least one queue family that supports `VK_QUEUE_GRAPHICS_BIT`.

```
int i = 0;
for (const auto& queueFamily : queueFamilies) {
    if (queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT) {
        indices.graphicsFamily = i;
    }

    i++;
}
```

Now that we have this fancy queue family lookup function, we can use it as a check in the `isDeviceSuitable` function to ensure that the device can process the commands we want to use:

```
bool isDeviceSuitable(VkPhysicalDevice device) {
    QueueFamilyIndices indices = findQueueFamilies(device);

    return indices.graphicsFamily.has_value();
}
```

To make this a little bit more convenient, we'll also add a generic check to the struct itself:

```
struct QueueFamilyIndices {
    std::optional<uint32_t> graphicsFamily;

    bool isComplete() {
        return graphicsFamily.has_value();
    }
};

...

bool isDeviceSuitable(VkPhysicalDevice device) {
    QueueFamilyIndices indices = findQueueFamilies(device);

    return indices.isComplete();
}
```

We can now also use this for an early exit from `findQueueFamilies`:

```
for (const auto& queueFamily : queueFamilies) {
    ...

    if (indices.isComplete()) {
        break;
    }

    i++;
}
```

Great, that's all we need for now to find the right physical device! The next step is to create a logical device to interface with it.

C++ code