Instance

Creating an instance

The very first thing you need to do is initialize the Vulkan library by creating an *instance*. The instance is the connection between your application and the Vulkan library and creating it involves specifying some details about your application to the driver.

Start by adding a `createInstance` function and invoking it in the `initVulkan` function.

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

Additionally add a data member to hold the handle to the instance:

```
private:
VkInstance instance;
```

Now, to create an instance we'll first have to fill in a struct with some information about our application. This data is technically optional, but it may provide some useful information to the driver in order to optimize our specific application (e.g. because it uses a well-known graphics engine with certain special behavior). This struct is called `VkApplicationInfo`:

```
void createInstance() {
    VkApplicationInfo appInfo{};
    appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
    appInfo.pApplicationName = "Hello Triangle";
    appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.pEngineName = "No Engine";
    appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
    appInfo.apiVersion = VK_API_VERSION_1_0;
}
```

As mentioned before, many structs in Vulkan require you to explicitly specify the type in the `sType` member. This is also one of the many structs with a `pNext` member that can point to extension information in the future. We're using value initialization here to leave it as `nullptr`.

A lot of information in Vulkan is passed through structs instead of function parameters and we'll have to fill in one more struct to provide sufficient information for creating an instance. This next struct is not optional and tells the Vulkan driver which global extensions and validation layers we

want to use. Global here means that they apply to the entire program and not a specific device, which will become clear in the next few chapters.

```
VkInstanceCreateInfo createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;
```

The first two parameters are straightforward. The next two layers specify the desired global extensions. As mentioned in the overview chapter, Vulkan is a platform agnostic API, which means that you need an extension to interface with the window system. GLFW has a handy built-in function that returns the extension(s) it needs to do that which we can pass to the struct:

```
uint32_t glfwExtensionCount = 0;
const char** glfwExtensions;

glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);

createInfo.enabledExtensionCount = glfwExtensionCount;
createInfo.ppEnabledExtensionNames = glfwExtensions;
```

The last two members of the struct determine the global validation layers to enable. We'll talk about these more in-depth in the next chapter, so just leave these empty for now.

```
createInfo.enabledLayerCount = 0;
```

We've now specified everything Vulkan needs to create an instance and we can finally issue the `vkCreateInstance` call:

```
VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);
```

As you'll see, the general pattern that object creation function parameters in Vulkan follow is:

- Pointer to struct with creation info
- Pointer to custom allocator callbacks, always `nullptr` in this tutorial
- Pointer to the variable that stores the handle to the new object

If everything went well then the handle to the instance was stored in the `VkInstance` class member. Nearly all Vulkan functions return a value of type `VkResult` that is either `VK_SUCCESS` or an error code. To check if the instance was created successfully, we don't need to store the result and can just use a check for the success value instead:

```
if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
    throw std::runtime_error("failed to create instance!");
```

```
}
```

Now run the program to make sure that the instance is created successfully.

Encountered VK_ERROR_INCOMPATIBLE_DRIVER:

If using MacOS with the latest MoltenVK sdk, you may get `VK_ERROR_INCOMPATIBLE_DRIVER` returned from `vkCreateInstance`. According to the Getting Start Notes. Beginning with the 1.3.216 Vulkan SDK, the `VK_KHR_PORTABILITY_subset` extension is mandatory.

To get over this error, first add the `VK_INSTANCE_CREATE_ENUMERATE_PORTABILITY_BIT_KHR` bit to `VkInstanceCreateInfo` struct's flags, then add `VK_KHR_PORTABILITY_ENUMERATION_EXTENSION_NAME` to instance enabled extension list.

Typically the code could be like this:

```
...

std::vector<const char*> requiredExtensions;

for(uint32_t i = 0; i < glfwExtensionCount; i++) {
    requiredExtensions.emplace_back(glfwExtensions[i]);
}

requiredExtensions.emplace_back(VK_KHR_PORTABILITY_ENUMERATION_EXTENSION_NAME);

createInfo.flags |= VK_INSTANCE_CREATE_ENUMERATE_PORTABILITY_BIT_KHR;

createInfo.enabledExtensionCount = (uint32_t) requiredExtensions.size();
createInfo.ppEnabledExtensionNames = requiredExtensions.data();

if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
    throw std::runtime_error("failed to create instance!");
}
```

Checking for extension support

If you look at the `vkCreateInstance` documentation then you'll see that one of the possible error codes is `VK_ERROR_EXTENSION_NOT_PRESENT`. We could simply specify the extensions we require and terminate if that error code comes back. That makes sense for essential extensions like the window system interface, but what if we want to check for optional functionality?

To retrieve a list of supported extensions before creating an instance, there's the `vkEnumerateInstanceExtensionProperties` function. It takes a pointer to a variable that stores the number of extensions and an array of `VkExtensionProperties` to store details of the

extensions. It also takes an optional first parameter that allows us to filter extensions by a specific validation layer, which we'll ignore for now.

To allocate an array to hold the extension details we first need to know how many there are. You can request just the number of extensions by leaving the latter parameter empty:

```
uint32_t extensionCount = 0;
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount, nullptr);
```

Now allocate an array to hold the extension details (`include <vector>`):

```
std::vector<VkExtensionProperties> extensions(extensionCount);
```

Finally we can query the extension details:

```
vkEnumerateInstanceExtensionProperties(nullptr, &extensionCount,
extensions.data());
```

Each `VkExtensionProperties` struct contains the name and version of an extension. We can list them with a simple for loop (`\t` is a tab for indentation):

```
std::cout << "available extensions:\n";

for (const auto& extension : extensions) {
    std::cout << '\t' << extension.extensionName << '\n';
}
```

You can add this code to the `createInstance` function if you'd like to provide some details about the Vulkan support. As a challenge, try to create a function that checks if all of the extensions returned by `glfwGetRequiredInstanceExtensions` are included in the supported extensions list.

Cleaning up

The `VkInstance` should be only destroyed right before the program exits. It can be destroyed in `cleanup` with the `vkDestroyInstance` function:

```
void cleanup() {
    vkDestroyInstance(instance, nullptr);

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

The parameters for the `vkDestroyInstance` function are straightforward. As mentioned in the previous chapter, the allocation and deallocation functions in Vulkan have an optional allocator callback that we'll ignore by passing `nullptr` to it. All of the other Vulkan resources that we'll create in the following chapters should be cleaned up before the instance is destroyed.

Before continuing with the more complex steps after instance creation, it's time to evaluate our debugging options by checking out validation layers.

C++ code
