Window surface

Since Vulkan is a platform agnostic API, it can not interface directly with the window system on its own. To establish the connection between Vulkan and the window system to present results to the screen, we need to use the WSI (Window System Integration) extensions. In this chapter we'll discuss the first one, which is `VK_KHR_surface`. It exposes a `VkSurfaceKHR` object that represents an abstract type of surface to present rendered images to. The surface in our program will be backed by the window that we've already opened with GLFW.

The `VK_KHR_surface` extension is an instance level extension and we've actually already enabled it, because it's included in the list returned by `glfwGetRequiredInstanceExtensions`. The list also includes some other WSI extensions that we'll use in the next couple of chapters.

The window surface needs to be created right after the instance creation, because it can actually influence the physical device selection. The reason we postponed this is because window surfaces are part of the larger topic of render targets and presentation for which the explanation would have cluttered the basic setup. It should also be noted that window surfaces are an entirely optional component in Vulkan, if you just need off-screen rendering. Vulkan allows you to do that without hacks like creating an invisible window (necessary for OpenGL).

Window surface creation

Start by adding a `surface` class member right below the debug callback.

```
VkSurfaceKHR surface;
```

Although the `VkSurfaceKHR` object and its usage is platform agnostic, its creation isn't because it depends on window system details. For example, it needs the `HWND` and `HMODULE` handles on Windows. Therefore there is a platform-specific addition to the extension, which on Windows is called `VK_KHR_win32_surface` and is also automatically included in the list from `glfwGetRequiredInstanceExtensions`.

I will demonstrate how this platform specific extension can be used to create a surface on Windows, but we won't actually use it in this tutorial. It doesn't make any sense to use a library like GLFW and then proceed to use platform-specific code anyway. GLFW actually has `glfwCreateWindowSurface` that handles the platform differences for us. Still, it's good to see what it does behind the scenes before we start relying on it.

To access native platform functions, you need to update the includes at the top:

```
#define VK_USE_PLATFORM_WIN32_KHR
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
#define GLFW_EXPOSE_NATIVE_WIN32
#include <GLFW/glfw3native.h>
```

Because a window surface is a Vulkan object, it comes with a `VkWin32SurfaceCreateInfoKHR` struct that needs to be filled in. It has two important parameters: `hwnd` and `hinstance`. These are the handles to the window and the process.

```
VkWin32SurfaceCreateInfoKHR createInfo{};
createInfo.sType = VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;
createInfo.hwnd = glfwGetWin32Window(window);
createInfo.hinstance = GetModuleHandle(nullptr);
```

The `glfwGetWin32Window` function is used to get the raw `HWND` from the GLFW window object. The `GetModuleHandle` call returns the `HINSTANCE` handle of the current process.

After that the surface can be created with `vkCreateWin32SurfaceKHR`, which includes a parameter for the instance, surface creation details, custom allocators and the variable for the surface handle to be stored in. Technically this is a WSI extension function, but it is so commonly used that the standard Vulkan loader includes it, so unlike other extensions you don't need to explicitly load it.

```
if (vkCreateWin32SurfaceKHR(instance, &createInfo, nullptr, &surface) !=
VK_SUCCESS) {
    throw std::runtime_error("failed to create window surface!");
}
```

The process is similar for other platforms like Linux, where `vkCreateXcbSurfaceKHR` takes an XCB connection and window as creation details with X11.

The `glfwCreateWindowSurface` function performs exactly this operation with a different implementation for each platform. We'll now integrate it into our program. Add a function `createSurface` to be called from `initVulkan` right after instance creation and `setupDebugMessenger`.

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

void createSurface() {
```

```
}
```

The GLFW call takes simple parameters instead of a struct which makes the implementation of the function very straightforward:

```
void createSurface() {
    if (glfwCreateWindowSurface(instance, window, nullptr, &surface) !=
VK_SUCCESS) {
        throw std::runtime_error("failed to create window surface!");
    }
}
```

The parameters are the `VkInstance`, GLFW window pointer, custom allocators and pointer to `VkSurfaceKHR` variable. It simply passes through the `VkResult` from the relevant platform call. GLFW doesn't offer a special function for destroying a surface, but that can easily be done through the original API:

```
void cleanup() {
        ...
        vkDestroySurfaceKHR(instance, surface, nullptr);
        vkDestroyInstance(instance, nullptr);
        ...
    }
```

Make sure that the surface is destroyed before the instance.

Querying for presentation support

Although the Vulkan implementation may support window system integration, that does not mean that every device in the system supports it. Therefore we need to extend `isDeviceSuitable` to ensure that a device can present images to the surface we created. Since the presentation is a queue-specific feature, the problem is actually about finding a queue family that supports presenting to the surface we created.

It's actually possible that the queue families supporting drawing commands and the ones supporting presentation do not overlap. Therefore we have to take into account that there could be a distinct presentation queue by modifying the `QueueFamilyIndices` structure:

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

    bool isComplete() {
        return graphicsFamily.has_value() && presentFamily.has_value();
```

```
    }
};
```

Next, we'll modify the `findQueueFamilies` function to look for a queue family that has the capability of presenting to our window surface. The function to check for that is `vkGetPhysicalDeviceSurfaceSupportKHR`, which takes the physical device, queue family index and surface as parameters. Add a call to it in the same loop as the `VK_QUEUE_GRAPHICS_BIT`:

```
VkBool32 presentSupport = false;
vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface, &presentSupport);
```

Then simply check the value of the boolean and store the presentation family queue index:

```
if (presentSupport) {
    indices.presentFamily = i;
}
```

Note that it's very likely that these end up being the same queue family after all, but throughout the program we will treat them as if they were separate queues for a uniform approach. Nevertheless, you could add logic to explicitly prefer a physical device that supports drawing and presentation in the same queue for improved performance.

Creating the presentation queue

The one thing that remains is modifying the logical device creation procedure to create the presentation queue and retrieve the `VkQueue` handle. Add a member variable for the handle:

```
VkQueue presentQueue;
```

Next, we need to have multiple `VkDeviceQueueCreateInfo` structs to create a queue from both families. An elegant way to do that is to create a set of all unique queue families that are necessary for the required queues:

```
#include <set>

...

QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
std::set<uint32_t> uniqueQueueFamilies = {indices.graphicsFamily.value(),
indices.presentFamily.value()};

float queuePriority = 1.0f;
```

```
for (uint32_t queueFamily : uniqueQueueFamilies) {
    VkDeviceQueueCreateInfo queueCreateInfo{};
    queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
    queueCreateInfo.queueFamilyIndex = queueFamily;
    queueCreateInfo.queueCount = 1;
    queueCreateInfo.pQueuePriorities = &queuePriority;
    queueCreateInfos.push_back(queueCreateInfo);
}
```

And modify `VkDeviceCreateInfo` to point to the vector:

```
createInfo.queueCreateInfoCount = static_cast<uint32_t>
(queueCreateInfos.size());
createInfo.pQueueCreateInfos = queueCreateInfos.data();
```

If the queue families are the same, then we only need to pass its index once. Finally, add a call to retrieve the queue handle:

```
vkGetDeviceQueue(device, indices.presentFamily.value(), 0, &presentQueue);
```

In case the queue families are the same, the two handles will most likely have the same value now. In the next chapter we're going to look at swap chains and how they give us the ability to present images to the surface.

C++ code