Base code

General structure

In the previous chapter you've created a Vulkan project with all of the proper configuration and tested it with the sample code. In this chapter we're starting from scratch with the following code:

```
#include <vulkan/vulkan.h>

#include <iostream>
#include <stdexcept>
#include <cstdlib>

class HelloTriangleApplication {
public:
    void run() {
        initVulkan();
        mainLoop();
        cleanup();
    }

private:
    void initVulkan() {

    }

    void mainLoop() {

    }

    void cleanup() {

    }
};

int main() {
    HelloTriangleApplication app;

    try {
        app.run();
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }
```

```
    return EXIT_SUCCESS;
}
```

We first include the Vulkan header from the LunarG SDK, which provides the functions, structures and enumerations. The `stdexcept` and `iostream` headers are included for reporting and propagating errors. The `cstdlib` header provides the `EXIT_SUCCESS` and `EXIT_FAILURE` macros.

The program itself is wrapped into a class where we'll store the Vulkan objects as private class members and add functions to initiate each of them, which will be called from the `initVulkan` function. Once everything has been prepared, we enter the main loop to start rendering frames. We'll fill in the `mainLoop` function to include a loop that iterates until the window is closed in a moment. Once the window is closed and `mainLoop` returns, we'll make sure to deallocate the resources we've used in the `cleanup` function.

If any kind of fatal error occurs during execution then we'll throw a `std::runtime_error` exception with a descriptive message, which will propagate back to the `main` function and be printed to the command prompt. To handle a variety of standard exception types as well, we catch the more general `std::exception`. One example of an error that we will deal with soon is finding out that a certain required extension is not supported.

Roughly every chapter that follows after this one will add one new function that will be called from `initVulkan` and one or more new Vulkan objects to the private class members that need to be freed at the end in `cleanup`.

Resource management

Just like each chunk of memory allocated with `malloc` requires a call to `free`, every Vulkan object that we create needs to be explicitly destroyed when we no longer need it. In C++ it is possible to perform automatic resource management using RAII or smart pointers provided in the `<memory>` header. However, I've chosen to be explicit about allocation and deallocation of Vulkan objects in this tutorial. After all, Vulkan's niche is to be explicit about every operation to avoid mistakes, so it's good to be explicit about the lifetime of objects to learn how the API works.

After following this tutorial, you could implement automatic resource management by writing C++ classes that acquire Vulkan objects in their constructor and release them in their destructor, or by providing a custom deleter to either `std::unique_ptr` or `std::shared_ptr`, depending on your ownership requirements. RAII is the recommended model for larger Vulkan programs, but for learning purposes it's always good to know what's going on behind the scenes.

Vulkan objects are either created directly with functions like `vkCreateXXX`, or allocated through another object with functions like `vkAllocateXXX`. After making sure that an object is no longer used anywhere, you need to destroy it with the counterparts `vkDestroyXXX` and `vkFreeXXX`. The parameters for these functions generally vary for different types of objects, but there is one parameter that they all share: `pAllocator`. This is an optional parameter that allows you to specify callbacks

for a custom memory allocator. We will ignore this parameter in the tutorial and always pass `nullptr` as argument.

Integrating GLFW

Vulkan works perfectly fine without creating a window if you want to use it for off-screen rendering, but it's a lot more exciting to actually show something! First replace the `#include <vulkan/vulkan.h>` line with

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

That way GLFW will include its own definitions and automatically load the Vulkan header with it. Add a `initWindow` function and add a call to it from the `run` function before the other calls. We'll use that function to initialize GLFW and create a window.

```
void run() {
    initWindow();
    initVulkan();
    mainLoop();
    cleanup();
}

private:
    void initWindow() {

    }
```

The very first call in `initWindow` should be `glfwInit()`, which initializes the GLFW library. Because GLFW was originally designed to create an OpenGL context, we need to tell it to not create an OpenGL context with a subsequent call:

```
glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
```

Because handling resized windows takes special care that we'll look into later, disable it for now with another window hint call:

```
glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);
```

All that's left now is creating the actual window. Add a `GLFWwindow* window;` private class member to store a reference to it and initialize the window with:

```
window = glfwCreateWindow(800, 600, "Vulkan", nullptr, nullptr);
```

The first three parameters specify the width, height and title of the window. The fourth parameter allows you to optionally specify a monitor to open the window on and the last parameter is only relevant to OpenGL.

It's a good idea to use constants instead of hardcoded width and height numbers because we'll be referring to these values a couple of times in the future. I've added the following lines above the `HelloTriangleApplication` class definition:

```
const uint32_t WIDTH = 800;
const uint32_t HEIGHT = 600;
```

and replaced the window creation call with

```
window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
```

You should now have a `initWindow` function that looks like this:

```
void initWindow() {
    glfwInit();

    glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
    glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

    window = glfwCreateWindow(WIDTH, HEIGHT, "Vulkan", nullptr, nullptr);
}
```

To keep the application running until either an error occurs or the window is closed, we need to add an event loop to the `mainLoop` function as follows:

```
void mainLoop() {
    while (!glfwWindowShouldClose(window)) {
        glfwPollEvents();
    }
}
```

This code should be fairly self-explanatory. It loops and checks for events like pressing the X button until the window has been closed by the user. This is also the loop where we'll later call a function to render a single frame.

Once the window is closed, we need to clean up resources by destroying it and terminating GLFW itself. This will be our first `cleanup` code:

```
void cleanup() {
    glfwDestroyWindow(window);
```

```
    glfwTerminate();
}
```

When you run the program now you should see a window titled `Vulkan` show up until the application is terminated by closing the window. Now that we have the skeleton for the Vulkan application, let's create the first Vulkan object!

C++ code
