The Vulkan API Companion Guide

Harry Gould

Contents

Chapter 1

An Introduction

This is going to be a technical book about the newly published Vulkan specification. We'll go over different topics such as:

- Getting started
- Instances
- Physical and logical devices
- Windowing across different platforms
- Surfaces
- Swapchains
- Image layouts
- Image views
- Please see the table-of-contents for more!

Resources

While resources are added everyday, there are a few I'd thoroughly recommend checking out:

- Vulkan Quick Reference v1.0
- Vulkan Specification v1.0.9
- Vulkan Specification v1.0.9 + WSI Extensions
- Sascha Willem's Vulkan Samples
- Vulkan in 30 Minutes

Code

You can find the repository on Github at HarryLovesCode/Vulkan-API-Book. For now, only Linux and Windows are **officially** supported. However, Android support is in the works.

Reading the Book

You can find the latest stable version on the Github releases page. Otherwise, you can read it on Gitbook. If you'd like to build it yourself, you will need `pandoc`, `xetex` (for Linux), and `latex-extras` installed.

Building Code on Linux

To build on Linux, you will need to make sure you have Vulkan headers available. Also, you will need `autotools`. When in doubt, look at your distribution's package repositories. You can use these commands.

```
git clone https://github.com/HarryLovesCode/Vulkan-API-Book
cd Vulkan-API-Book

autoreconf --install
./configure
make -j4
```

Once you've done that, you will find all the binaries located in the `./bin` folder.

Building Code on Windows

To build on Windows, you'll need Visual Studio 2015. You can find the Visual Studio solution in the root directory of the repository. Just open that, choose a startup project, and you're ready to go.

A Little About Me

I'm Harry and I'm a young developer who enjoys 3D graphics. I've worked with WebGL, OpenGL, and DirectX and when I heard about Vulkan, I was excited to get started. The publishing of this book is, in a way, and experiment because I'm publishing as I go. Thus, the book may be rough around the edges. Feel free to submit issues or pull requests on Github and add inline comments through Gitbook.

Chapter 2

Creating an Instance

Before we are able to start using Vulkan, we must first create an instance. A `VkInstance` is an object that contains all the information the implementation needs to work. Unlike OpenGL, Vulkan does not have a global state. Because of this, we must instead store our states in this object. In this chapter, we'll be beginning a class we'll use for the rest of the book. Here are the headers we will need to include:

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>

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

We'll be storing all of our variables in a class we'll call `VulkanSwapchain` for now. The code will start out looking like:

```
class VulkanSwapchain {
 private:
  void exitOnError(const char* msg);
  void initInstance();

  const char* applicationName = "Vulkan Example";
  const char* engineName = "Vulkan Engine";

  VkInstance instance;
 public:
  VulkanSwapchain();
  virtual ~VulkanSwapchain();
};
```

In this chapter, we'll be focusing on the constructor and the `initInstance` method.

Application Information

Before we can use `VkInstanceCreateInfo`, we need to use another type to describe our application. The `VkApplication-Info` contains the name of the application, application version, engine name, engine version, and API version. Like OpenGL, if the driver does not support the version we request, we most likely will not be able to recover.

Definition for `VkApplicationInfo`:

```
typedef struct VkApplicationInfo {
  VkStructureType    sType;
  const void*        pNext;
  const char*        pApplicationName;
  uint32_t           applicationVersion;
  const char*        pEngineName;
  uint32_t           engineVersion;
  uint32_t           apiVersion;
} VkApplicationInfo;
```

Documentation for `VkApplicationInfo`:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `pApplicationName` is a pointer to a null-terminated UTF-8 string containing the name of the application.
- `applicationVersion` is an unsigned integer variable containing the developer-supplied version number of the application.
- `pEngineName` is a pointer to a null-terminated UTF-8 string containing the name of the engine (if any) used to create the application.
- `engineVersion` is an unsigned integer variable containing the developer-supplied version number of the engine used to create the application.
- `apiVersion` is the version of the Vulkan API against which the application expects to run (encoded). If `apiVersion` is `0` the implementation must ignore it, otherwise if the implementation does not support the requested `apiVersion` it must return `VK_ERROR_INCOMPATIBLE_DRIVER`.

Usage for `VkApplicationInfo`:

```
VkApplicationInfo appInfo = {};
appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
appInfo.pNext = NULL;
appInfo.pApplicationName = applicationName;
appInfo.pEngineName = engineName;
appInfo.apiVersion = VK_MAKE_VERSION(1, 0, 3);
```

You'll notice that for `apiVersion`, I am using `VK_MAKE_VERSION`. This allows the developer to specify a targeted Vulkan version. We'll see later that if the version we try to get is unsupported, we'll get an error called `VK_ERROR_INCOMPATIBLE_DRIVER`.

Instance Create Information

`VkInstanceCreateInfo` will be used to inform Vulkan of our application info, layers we'll be using, and extensions we want.

Definition for `VkInstanceCreateInfo`:

```
typedef struct VkInstanceCreateInfo {
  VkStructureType             sType;
  const void*                 pNext;
  VkInstanceCreateFlags       flags;
  const VkApplicationInfo*    pApplicationInfo;
  uint32_t                    enabledLayerCount;
  const char* const*          ppEnabledLayerNames;
  uint32_t                    enabledExtensionCount;
  const char* const*          ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

Documentation for `VkInstanceCreateInfo`:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `pApplicationInfo` is `NULL` or a pointer to an instance of `VkApplicationInfo`. If not `NULL`, this information helps implementations recognize behavior inherent to classes of applications.
- `enabledLayerCount` is the number of global layers to enable.
- `ppEnabledLayerNames` is a pointer to an array of `enabledLayerCount` null-terminated UTF-8 strings containing the names of layers to enable for the created instance.
- `enabledExtensionCount` is the number of global extensions to enable.
- `ppEnabledExtensionNames` is a pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings containing the names of extensions to enable.

Usage for `VkInstanceCreateInfo`:

```
VkInstanceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pNext = NULL;
createInfo.flags = 0;
createInfo.pApplicationInfo = &appInfo;
```

However, we're missing platform specific extensions. We'll need these later when we approach rendering and get started with swap chains. Basically, Vulkan as an API does not make us render to a surface. But if you're rendering a scene, you'll most likely want to display something. Retrieving a surface is platform specific and requires extensions. Let's look at how to get the extension for a surface:

```
std::vector<const char*> enabledExtensions = { VK_KHR_SURFACE_EXTENSION_NAME };

#if defined(_WIN32)
    enabledExtensions.push_back(VK_KHR_WIN32_SURFACE_EXTENSION_NAME);
#elif defined(__ANDROID__)
    enabledExtensions.push_back(VK_KHR_ANDROID_SURFACE_EXTENSION_NAME);
#elif defined(__linux__)
    enabledExtensions.push_back(VK_KHR_XCB_SURFACE_EXTENSION_NAME);
#endif
```

Now we should be able to go back to our `createInfo` variable and add these lines:

```
createInfo.enabledExtensionCount = enabledExtensions.size();
createInfo.ppEnabledExtensionNames = enabledExtensions.data();
```

Creating an Instance

Finally we're ready to create our instance. A Vulkan instance is similar, in concept, to an OpenGL rendering context. Like I alluded to before, we'll store the engine state here including layers and extensions. To create an instance, we'll be using the `vkCreateInstance` function.

Definition for `vkCreateInstance`:

```
VkResult vkCreateInstance(
  const VkInstanceCreateInfo*                 pCreateInfo,
  const VkAllocationCallbacks*                pAllocator,
  VkInstance*                                 pInstance);
```

Documentation for `vkCreateInstance`:

- `pCreateInfo` points to an instance of `VkInstanceCreateInfo` controlling creation of the instance.
- `pAllocator` controls host memory allocation.
- `pInstance` points a `VkInstance` handle in which the resulting instance is returned.

You'll want to note two things. First, we're not going to use `pAllocator`. Instead, we'll pass in `NULL` for this argument. Second, the call to this function returns a `VkResult`. This value is used for many function calls in Vulkan. It will tell us if we were successful, failed, or if something else happened. In this case, the value will tell us if the instance creation was successful or if it failed.

Usage for `vkCreateInstance`:

```
VkResult res = vkCreateInstance(&createInfo, NULL, &instance);
```

After we've made the call, we should first check if our driver compatible. If it is not, we need to exit. We cannot recover. However, we should provide some sort of useful error. This error is extremely common in my experience. Second, we'll check if our call to `vkCreateInstance` successful. Again, if it was not, we need to exit. We cannot continue. These checks can be made with the following code:

```
if (res == VK_ERROR_INCOMPATIBLE_DRIVER) {
  exitOnError(
      "Cannot find a compatible Vulkan installable client "
      "driver (ICD). Please make sure your driver supports "
      "Vulkan before continuing. The call to vkCreateInstance failed.");
} else if (res != VK_SUCCESS) {
  exitOnError(
      "The call to vkCreateInstance failed. Please make sure "
      "you have a Vulkan installable client driver (ICD) before "
      "continuing.");
}
```

Termination on Error

Our `exitOnError` method is simple at the moment. We'll make some minor changes to it when we start working with windows, but for now, this will fulfill our needs:

```
void VulkanSwapchain::exitOnError(const char* msg) {
  fputs(msg, stderr);
  exit(EXIT_FAILURE);
}
```

Cleaning Up (Destructor)

Exiting should be graceful if possible. In the case that our destructor is called, we will destroy the instance we created. Afterwards, the program will exit. The destructor looks like this:

```
VulkanSwapchain::~VulkanSwapchain() {
  vkDestroyInstance(instance, NULL);
}
```

Chapter 3

Physical Devices and Logical Devices

Once we have created a Vulkan instance, we can use two objects to interact with our implementation. These objects are queues and devices. This chapter is going to focus on the two types of devices: physical and logical. A physical device is a single component in the system. It can also be multiple components working in conjunction to function like a single device. A logical device is basically our interface with the physical device.

Physical Devices

A `VkPhysicalDevice` is a data type that we will use to represent each piece of hardware. There's not much to say here other than we will pass a pointer to an array to the implementation. The implementation will then write handles for each physical device in the system to said array. You can find more information on physical devices here.

Enumerating Physical Devices

To get the handles of all the physical devices in the system, we can call use `vkEnumeratePhysicalDevices`. We will call it twice. First, we'll pass in `NULL` as the last parameter. This will allow us to get `pPhysicalDeviceCount` out by passing in the address to a variable for the second argument. After that, we can allocate the memory necessary to store the `pPhysicalDevices` and call it with that variable as the last argument.

Definition for `vkEnumeratePhysicalDevices`:

```
VkResult vkEnumeratePhysicalDevices(
  VkInstance                                  instance,
  uint32_t*                                   pPhysicalDeviceCount,
  VkPhysicalDevice*                           pPhysicalDevices);
```

Documentation for `vkEnumeratePhysicalDevices`:

- `instance` is a handle to a Vulkan instance previously created with `vkCreateInstance`.
- `pPhysicalDeviceCount` is a pointer to an integer related to the number of physical devices available or queried.
- `pPhysicalDevices` is either `NULL` or a pointer to an array of `VkPhysicalDevice` structures.

Before we create allocate memory to store the physical devices, we need to figure out how many there are. We can do this by calling `vkEnumeratePhysicalDevices` with a value of `NULL` for `pPhysicalDevices`.

Usage for `vkEnumeratePhysicalDevices`:

```
uint32_t deviceCount = 0;
VkResult result = vkEnumeratePhysicalDevices(instance, &deviceCount, NULL);
```

We should make two assertions.

- The function call was successful

```
assert(result == VK_SUCCESS);
```

- We found one or more Vulkan compatible device:

```
assert(deviceCount >= 1);
```

Following the usage guidelines outlined in the specification, a second call to `vkEnumeratePhysicalDevices` with error checking would look like this:

```
std::vector<VkPhysicalDevice> physicalDevices(deviceCount);
result = vkEnumeratePhysicalDevices(instance, &deviceCount, physicalDevices.data());
assert(result == VK_SUCCESS);
```

For now, we will simply choose the first device (at index `0`) in the array of `physicalDevices`.

Physical Device Properties

`VkPhysicalDeviceProperties` is a data type that we will use to represent properties of each physical device. There's not much to say here other than we will pass a pointer of this type to the implementation. The implementation will then write properties for the specified `VkPhysicalDevice`.

Definition for `VkPhysicalDeviceProperties`:

```
typedef struct VkPhysicalDeviceProperties {
  uint32_t                            apiVersion;
  uint32_t                            driverVersion;
  uint32_t                            vendorID;
  uint32_t                            deviceID;
  VkPhysicalDeviceType                deviceType;
  char                                deviceName[VK_MAX_PHYSICAL_DEVICE_NAME_SIZE];
  uint8_t                             pipelineCacheUUID[VK_UUID_SIZE];
  VkPhysicalDeviceLimits              limits;
  VkPhysicalDeviceSparseProperties    sparseProperties;
} VkPhysicalDeviceProperties;
```

Documentation for 'VkPhysicalDeviceProperties:

- `apiVersion` is the version of Vulkan supported by the device (encoded).
- `driverVersion` is the vendor-specified version of the driver.
- `vendorID` is a unique identifier for the vendor of the physical device.
- `deviceID` is a unique identifier for the physical device among devices available from the vendor.
- `deviceType` is a `VkPhysicalDeviceType` specifying the type of device.
- `deviceName` is a null-terminated UTF-8 string containing the name of the device.
- `pipelineCacheUUID` is an array of size `VK_UUID_SIZE`, containing 8-bit values that represent a universally unique identifier for the device.
- `limits` is the `VkPhysicalDeviceLimits` structure which specifies device-specific limits of the physical device.

- `sparseProperties` is the `VkPhysicalDeviceSparseProperties` structure which specifies various sparse related properties of the physical device.

And, just for reference, the definition for `VkPhysicalDeviceType` looks like this:

```
typedef enum VkPhysicalDeviceType {
  VK_PHYSICAL_DEVICE_TYPE_OTHER = 0,
  VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU = 1,
  VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU = 2,
  VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU = 3,
  VK_PHYSICAL_DEVICE_TYPE_CPU = 4,
} VkPhysicalDeviceType;
```

This may be useful if you are trying to detect (for example) if you have an integrated GPU versus a discrete GPU.

Getting Physical Device Properties

A call to `vkGetPhysicalDeviceProperties` can be useful if you are interested in retrieving information about the physical devices in the system. It will tell you API version, driver version, limitations, sparse properties, etc.

Definition for `vkGetPhysicalDeviceProperties`:

```
void vkGetPhysicalDeviceProperties(
  VkPhysicalDevice                            physicalDevice,
  VkPhysicalDeviceProperties*                 pProperties);
```

Documentation for `vkGetPhysicalDeviceProperties`:

- `instance` is a handle to a Vulkan instance previously created with `vkCreateInstance`.
- `pPhysicalDeviceCount` is a pointer to an integer related to the number of physical devices available or queried.
- `pPhysicalDevices` is either `NULL` or a pointer to an array of `VkPhysicalDevice` structures.

Usage for `vkGetPhysicalDeviceProperties`:

```
VkPhysicalDeviceProperties physicalProperties = {};

for (uint32_t i = 0; i < deviceCount; i++)
  vkGetPhysicalDeviceProperties(physicalDevices[i], &physicalProperties);
```

We can output some useful parts of the information using this piece of code in the loop above:

```
fprintf(stdout, "Device Name:    %s\n", physicalProperties.deviceName);
fprintf(stdout, "Device Type:    %d\n", physicalProperties.deviceType);
fprintf(stdout, "Driver Version: %d\n", physicalProperties.driverVersion);
```

As I mentioned before, the API version is encoded. So if we want, we can use three macros that will help make it human readable:

- `VK_VERSION_MAJOR(version)`
- `VK_VERSION_MINOR(version)`
- `VK_VERSION_PATCH(version)`

So, to output the API version, you can use this:

```
fprintf(stdout, "API Version:    %d.%d.%d\n",
        VK_VERSION_MAJOR(physicalProperties.apiVersion),
        VK_VERSION_MINOR(physicalProperties.apiVersion),
        VK_VERSION_PATCH(physicalProperties.apiVersion));
```

Device Queue Create Information

The next step is to create a device using `vkCreateDevice`. However, in order to do that, we must have a `VkDeviceCreateInfo` object. And, as you may have guessed having seen the specification, we need a `VkDeviceQueueCreateInfo` object.

Definition for `VkDeviceQueueCreateInfo`:

```
typedef struct VkDeviceQueueCreateInfo {
  VkStructureType             sType;
  const void*                 pNext;
  VkDeviceQueueCreateFlags    flags;
  uint32_t                    queueFamilyIndex;
  uint32_t                    queueCount;
  const float*                pQueuePriorities;
} VkDeviceQueueCreateInfo;
```

Documentation for `VkDeviceQueueCreateInfo`:

- `sType` is the type of this structure.
- `pNext` is NULL or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `queueFamilyIndex` is an unsigned integer indicating the index of the queue family to create on this device. This index corresponds to the index of an element of the `pQueueFamilyProperties` array that was returned by `vkGetPhysicalDeviceQueueFamilyProperties`.
- `queueCount` is an unsigned integer specifying the number of queues to create in the queue family indicated by `queueFamilyIndex`.
- `pQueuePriorities` is an array of `queueCount` normalized floating point values, specifying priorities of work that will be submitted to each created queue.

Usage for `VkDeviceQueueCreateInfo`:

```
float priorities[] = { 1.0f };
VkDeviceQueueCreateInfo queueInfo{};
queueInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
queueInfo.pNext = NULL;
queueInfo.flags = 0;
queueInfo.queueFamilyIndex = 0;
queueInfo.queueCount = 1;
queueInfo.pQueuePriorities = &priorities[0];
```

You'll note that we create a `float` array with a single value. Each value in that array will tell the implementation the priority of the queue. Values must be between `0.0` and `1.0`. Certain implementations will give higher-priority queues more processing time. However, this is not necessarily true because the specification doesn't require this behavior.

Device Create Info

The parent of `VkDeviceQueueCreateInfo` is `VkDeviceCreateInfo`.

Definition for `VkDeviceCreateInfo`:

```
typedef struct VkDeviceCreateInfo {
  VkStructureType                    sType;
  const void*                        pNext;
  VkDeviceCreateFlags                  flags;
  uint32_t                             queueCreateInfoCount;
  const VkDeviceQueueCreateInfo*       pQueueCreateInfos;
  uint32_t                             enabledLayerCount;
  const char* const*                   ppEnabledLayerNames;
  uint32_t                             enabledExtensionCount;
  const char* const*                   ppEnabledExtensionNames;
  const VkPhysicalDeviceFeatures*      pEnabledFeatures;
} VkDeviceCreateInfo;
```

Documentation for `VkDeviceCreateInfo`:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `queueCreateInfoCount` is the unsigned integer size of the `pQueueCreateInfos` array.
- `pQueueCreateInfos` is a pointer to an array of `VkDeviceQueueCreateInfo` structures describing the queues that are requested to be created along with the logical device.
- `enabledLayerCount` is the number of device layers to enable.
- `ppEnabledLayerNames` is a pointer to an array of `enabledLayerCount` null-terminated UTF-8 strings containing the names of layers to enable for the created device.
- `enabledExtensionCount` is the number of device extensions to enable.
- `ppEnabledExtensionNames` is a pointer to an array of `enabledExtensionCount` null-terminated UTF-8 strings containing the names of extensions to enable for the created device.
- `pEnabledFeatures` is NULL or a pointer to a `VkPhysicalDeviceFeatures` structure that contains boolean indicators of all the features to be enabled.

Usage for `VkDeviceCreateInfo`:

```
std::vector<const char *> enabledExtensions = { VK_KHR_SWAPCHAIN_EXTENSION_NAME };
VkDeviceCreateInfo deviceInfo{};
deviceInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
deviceInfo.pNext = NULL;
deviceInfo.flags = 0;
deviceInfo.queueCreateInfoCount = 1;
deviceInfo.pQueueCreateInfos = &queueInfo;
deviceInfo.enabledExtensionCount = enabledExtensions.size();
deviceInfo.ppEnabledExtensionNames = enabledExtensions.data();
deviceInfo.pEnabledFeatures = NULL;
```

Creating a Device

Finally, to wrap up this section, we need to create a logical device. We'll use the `vkCreateDevice`.

Definition for `vkCreateDevice`:

```
VkResult vkCreateDevice(
  VkPhysicalDevice                          physicalDevice,
  const VkDeviceCreateInfo*                 pCreateInfo,
  const VkAllocationCallbacks*              pAllocator,
  VkDevice*                                 pDevice);
```

Documentation for `vkCreateDevice`:

- `physicalDevice` must be one of the device handles returned from a call to `vkEnumeratePhysicalDevices`.
- `pCreateInfo` is a pointer to a `VkDeviceCreateInfo` structure containing information about how to create the device.
- `pAllocator` controls host memory allocation.
- `pDevice` points to a handle in which the created `VkDevice` is returned.

Usage for `vkCreateDevice`:

```
VkResult result = vkCreateDevice(physicalDevice, &deviceInfo,
                                 NULL, &logicalDevice);
assert(result != VK_SUCCESS);
```

Chapter 4

Cross-Platform Windows

This section is divided up by platform. Please see your respective platform more definitions, usage, explanations, etc.

Creating a Window on Linux

In order to create a window, we're going to be using platform specific code. Please note that this tutorial is for Linux **only**. None of this will apply to Windows, Android, GLFW, etc.

Including Headers

For windowing on Linux, we're going to be using the **X protocol C-language Binding (XCB)**. This, while not as simple as using GLFW is still a pretty easy API to use. Before we can begin, we'll need to include the header:

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

If you're targeting Windows as well, you should surround the include by the `#if defined(__linux__)` directives.

Methods in Our Class

For this chapter, we're going to be writing the contents of these two methods:

```
void initWindow() {}
void renderLoop() {}
```

Again, if you're targeting Windows as well, you should surround both by the `#if defined(__linux__)` directives.

Variables in Our Class

We'll need to keep track of a few variables in our class. I'll be adding these:

```
xcb_connection_t * connection;
xcb_window_t window;
xcb_screen_t * screen;
xcb_atom_t wmProtocols;
xcb_atom_t wmDeleteWin;
```

to our `VulkanSwapchain` class.

Getting a Connection

XCB will allow us to interact with the X server, but we'll need to get a display first. We can use the `xcb_connect` method to get the value of the `DISPLAY` environment variable. Because we don't know it already, we should pass in `NULL` for that argument.

Definition for `xcb_connect`:

```
xcb_connection_t* xcb_connect(const char * displayname, int * screenp);
```

Documentation for `xcb_connect`:

- `displayname` is the name of the display or `NULL` if we want the display specified by the `DISPLAY` variable.
- `screenp` is a pointer to a preferred screen number.
- Returns a newly allocated `xcb_connection_t` structure.

Usage for `xcb_connect`:

```
int screenp = 0;
connection = xcb_connect(NULL, &screenp);
```

Definition for `xcb_connection_has_error`:

Once we've called `xcb_connect`, we should check for any connection errors. It is recommended that we use the `xcb_connection_has_error` method for this.

```
int xcb_connection_has_error(xcb_connection_t * c);
```

Documentation for `xcb_connection_has_error`:

- `c` is the connection.
- Returns a value > `0` if the connection is in an error state; `0` otherwise.

Usage for `xcb_connection_has_error`:

```
if (xcb_connection_has_error(connection))
  exitOnError("Failed to connect to X server using XCB.");
```

Getting Setups

Now that we've handled any errors, we need to get available screens. In order to get data from the X server, we can use `xcb_get_setup`. Then we can get a screen iterator from the setup using `xcb_setup_roots_iterator`. Once we've done that, we should loop through the screens using `xcb_screen_next`.

Definition for `xcb_get_setup`:

```
const struct xcb_setup_t* xcb_get_setup(xcb_connection_t * c);
```

Documentation for `xcb_get_setup`:

- `c` is the connection.
- Returns a pointer to an `xcb_setup_t` structure.

Definition for `xcb_setup_roots_iterator`:

```
xcb_screen_iterator_t xcb_setup_roots_iterator(const xcb_setup_t * R);
```

Documentation for `xcb_setup_roots_iterator`:

- TODO

Definition for `xcb_screen_next`:

```
void xcb_screen_next(xcb_screen_iterator_t * i);
```

Documentation for `xcb_screen_next`:

- `i` is a pointer to a `xcb_screen_iterator_t`.

Usage for all methods:

```
xcb_screen_iterator_t iter =
    xcb_setup_roots_iterator(xcb_get_setup(connection));

for (int s = screenp; s > 0; s--)
    xcb_screen_next(&iter);
```

Once we have looped through all of the screens, we can then say:

```
screen = iter.data;
```

Generating a XID

Before we can create a Window, we'll need to allocate a XID for it. We can use the `xcb_generate_id` method for this.

Definition for `xcb_generate_id`:

```
uint32_t xcb_generate_id(xcb_connection_t * c);
```

Documentation for `xcb_generate_id`:

- `c` is the connection.
- Returns a newly allocated XID.

Usage for `xcb_generate_id`:

```
window = xcb_generate_id(connection);
```

Event Masks

In XCB, we use **event masks** to register event types. Event masks are essentially **bitmasks** which will tell XCB what events we care about. We'll be using two values for our event mask: `XCB_CW_BACK_PIXEL` and `XCB_CW_EVENT_MASK`. The first will tell XCB we're going to be filling the background of the window with one pixel color. The second is required. We're going to then use a bitwise `OR` operator to create the bitmask.

```
uint32_t eventMask = XCB_CW_BACK_PIXEL | XCB_CW_EVENT_MASK;
```

Now we can prepare the values that will get sent over to XCB. This will include the background color we want and the events we're looking for:

```
uint32_t valueList[] = { screen->black_pixel, 0 };
```

For now, we're not going to worry about events such as keypresses. So, we can go ahead and use the `xcb_create_window` method to create our window.

Definition for `xcb_create_window`:

```
xcb_void_cookie_t xcb_create_window(
  xcb_connection_t * c,
  uint8_t            depth,
  xcb_window_t       window,
  xcb_window_t      parent,
  int16_t           x,
  int16_t           y,
  uint16_t          width,
  uint16_t          height,
  uint16_t          border_width,
  uint16_t          class,
  xcb_visualid_t    visual,
  uint32_t          value_mask,
  const uint32_t *  value_list);
```

Documentation for `xcb_create_window`:

- `conn` is the XCB connection to X11.
- `depth` specifies the new window's depth in units. The special value `XCB_COPY_FROM_PARENT`means the depth is taken from the parent window.
- `wid` is the ID with which you will refer to the new window, created by `xcb_generate_id`.
- `parent` is the parent window of the new window.
- `x` is the X coordinate of the new window.
- `y` is the Y coordinate of the new window.
- `width` is the width of the new window.
- `height` is the height of the new window.
- `border_width` must be zero if the class is `InputOnly` or a `xcb_match_error_t` occurs.
- `class` must be one of the following values: `XCB_WINDOW_CLASS_COPY_FROM_PARENT`, `XCB_WINDOW_CLASS_INPUT_OUTPUT`, or `XCB_WINDOW_CLASS_INPUT_ONLY`.
- `visual` specifies the id for the new window's visual.
- Returns a `xcb_void_cookie_t`. Errors (if any) have to be handled in the event loop.

Usage for `xcb_create_window`:

```
xcb_create_window(
  connection,
  XCB_COPY_FROM_PARENT,
  window,
  screen->root,
  0,
  0,
  windowWidth,
  windowHeight,
  0,
  XCB_WINDOW_CLASS_INPUT_OUTPUT,
  screen->root_visual,
  eventMask,
  valueList);
```

Modifying Properties

Before we map the window to the screen, we're going to change the title. We can use the `xcb_change_property` method to do this.

Definition for `xcb_change_property`:

```
xcb_void_cookie_t xcb_change_property(
  xcb_connection_t * conn,
  uint8_t            mode,
  xcb_window_t       window,
  xcb_atom_t         property,
  xcb_atom_t         type,
  uint8_t            format,
  uint32_t           data_len,
  const void *       data);
```

Documentation for `xcb_change_property`:

- `conn` is the XCB connection to X11.
- `mode` is one of the following values: `XCB_PROP_MODE_REPLACE` (Discard the previous property value and store the new data), `XCB_PROP_MODE_PREPEND` (Insert the new data before the beginning of existing data), or `XCB_PROP_MODE_APPEND` (Insert the new data after the beginning of existing data).
- `window` is the window whose property you want to change.
- `property` is the property you want to change (an atom).
- `type` is the type of the property you want to change (an atom).
- `format` specifies whether the data should be viewed as a list of 8-bit, 16-bit or 32-bit quantities. Possible values are 8, 16 and 32.
- `data_len` specifies the number of elements (see format).
- `data` is the property data.
- Returns an `xcb_void_cookie_t`. Errors (if any) have to be handled in the event loop.

Usage for `xcb_change_property`:

```
xcb_change_property(
  connection,
  XCB_PROP_MODE_REPLACE,
  window,
  XCB_ATOM_WM_NAME,
  XCB_ATOM_STRING,
  8,
  strlen(applicationName),
  applicationName);
```

Window Mapping and Event Stream Flushing

Now we can simply use `xcb_map_window` and `xcb_flush` to map the window to the screen and flush the stream to the server.

Definition for `xcb_map_window`:

```
xcb_void_cookie_t xcb_map_window(
  xcb_connection_t * conn,
  xcb_window_t       window);
```

Documentation for `xcb_map_window`:

- `conn` is the XCB connection to X11.
- `window` is the window to make visible.
- Returns an `xcb_void_cookie_t`. Errors (if any) have to be handled in the event loop.

Definition for `xcb_flush`:

```
int xcb_flush(xcb_connection_t * c);
```

Documentation for `xcb_flush`:

- `c` is the connection to the X server.
- Returns > `0` on success, <= `0` otherwise.

Usage for both methods:

```
xcb_map_window(connection, window);
xcb_flush(connection);
```

Close Button

I won't go too far into depth about the code below. Essentially what it does is tells the server we want to be alerted when the window manager attempts to destroy the window. This piece of code goes right before we map the window to the screen and flush the stream:

```
xcb_intern_atom_cookie_t wmDeleteCookie = xcb_intern_atom(
    connection, 0, strlen("WM_DELETE_WINDOW"), "WM_DELETE_WINDOW");
xcb_intern_atom_cookie_t wmProtocolsCookie =
    xcb_intern_atom(connection, 0, strlen("WM_PROTOCOLS"), "WM_PROTOCOLS");
xcb_intern_atom_reply_t *wmDeleteReply =
    xcb_intern_atom_reply(connection, wmDeleteCookie, NULL);
xcb_intern_atom_reply_t *wmProtocolsReply =
    xcb_intern_atom_reply(connection, wmProtocolsCookie, NULL);
wmDeleteWin = wmDeleteReply->atom;
wmProtocols = wmProtocolsReply->atom;

xcb_change_property(connection, XCB_PROP_MODE_REPLACE, window,
                    wmProtocolsReply->atom, 4, 32, 1, &wmDeleteReply->atom);
```

Our Update Loop

All we need to do now is write the contents of the `renderLoop` method. Again, I'm not going to be going too far into depth because this is just glue and we're focused on Vulkan here. Anyways, we'll be using the `xcb_wait_for_event` method to do just that: wait for events. In order to convert the server's response to a value we can check in a **switch case** block, we need to use the bitwise AND (&) operator with the value: `~0x80`. You can see what I'm talking about below:

```
void VulkanSwapchain::renderLoop() {
  bool running = true;
  xcb_generic_event_t *event;
  xcb_client_message_event_t *cm;

  while (running) {
    event = xcb_wait_for_event(connection);

    switch (event->response_type & ~0x80) {
      case XCB_CLIENT_MESSAGE: {
        cm = (xcb_client_message_event_t *)event;
        if (cm->data.data32[0] == wmDeleteWin)
          running = false;

        break;
      }
    }

    free(event);
  }

  xcb_destroy_window(connection, window);
}
```

Once we get the `wmDeleteWin` response, we should use `xcb_destroy_window` to destroy our main window.

Creating a Window on Microsoft Windows

In order to create a window, we're going to be using platform specific code. Please note that this tutorial is for Windows **only**. None of this will apply to Linux, Android, GLFW, etc. I'll make the warning ahead of time: this chapter makes use of a lot of functions. Their definitions are not super relevant to you as a Vulkan developer because this will be written once.

Including Headers

Because we're going to be writing this from scratch, we're going to have to interact with Windows directly. Before we can do anything, we need to include the Windows header. If you're only targeting Windows, you can write:

```
#include <windows.h>
```

If you're targeting Linux as well, you should surround both by the `#if defined(_WIN32)` directives.

Setting Up a Console Window

Because we're now switching from a **Windows Console Application** to a **Windows Application**, we'll need to make sure we have a console to view the output of `stdout` and `stderr`. Also, because we're exiting right after we encounter an error, we should:

- Show a message box
- Wait for user input (keypress)
- Close after the user has acknowledged the error

We'll be using four methods to do this work.

Definition for `AllocConsole`:

```
BOOL WINAPI AllocConsole(void);
```

Documentation for `AllocConsole`:

- This function takes no arguments

Usage for `AllocConsole`:

```
AllocConsole();
```

Definition for `AttachConsole`:

```
BOOL WINAPI AttachConsole(
  DWORD dwProcessId
);
```

Documentation for `AttachConsole`:

- `dwProcessId` is the identifier of the process whose console is to be used.

Usage for `AttachConsole`:

```
AttachConsole(GetCurrentProcessId());
```

Definition for `freopen`:

```
FILE * freopen (
  const char * filename,
  const char * mode,
  FILE * stream );
```

Documentation for `freopen`:

- `fileName` is a C string containing the name of the file to be opened.
- `mode` is a C string containing a file access mode. It can be:
- `"r"`
- `"w"`
- `"a"`
- etc.
- `stream` is a pointer to a `FILE` object that identifies the stream to be reopened.

Usage for `freopen`:

```
freopen("CON", "w", stdout);
freopen("CON", "w", stderr);
```

Definition for `SetConsoleTitle`:

```
BOOL WINAPI SetConsoleTitle(
  LPCTSTR lpConsoleTitle
);
```

Documentation for `SetConsoleTitle`:

- `lpConsoleTitle` is the string to be displayed in the title bar of the console window. The total size must be less than 64K.

Usage for `SetConsoleTitle`:

```
SetConsoleTitle(TEXT(applicationName));
```

If you put these methods together you can:

- Allocate a console
- Attach the console to the current process
- Redirect `stdout` and `stderr` to said console
- Set the title of the console window

Now, let's modify our `exitOnError` method to show a error message box. We'll need to use the `MessageBox` method.

Definition for `MessageBox`:

```
int WINAPI MessageBox(
  HWND    hWnd,
  LPCTSTR lpText,
  LPCTSTR lpCaption,
  UINT    uType
);
```

Documentation for `MessageBox`:

- `hWnd` is a handle to the owner window of the message box to be created. If this parameter is `NULL`, the message box has no owner window.
- `lpText` is the message to be displayed. If the string consists of more than one line, you can separate the lines using a carriage return and/or linefeed character between each line.
- `lpCaption` is the dialog box title. If this parameter is `NULL`, the default title is "Error".
- `uType` is the contents and behavior of the dialog box. This parameter can be a combination of flags.

Usage for `MessageBox`:

```
MessageBox(NULL, msg, applicationName, MB_ICONERROR);
```

Creating a Window

As mentioned before, because we're writing this without any windowing libraries, we'll have to use the Win32 API. I won't go too much into detail because **our focus is Vulkan** and not Windows. Let's go ahead and list the variables we'll be using:

```
const uint32_t windowWidth = 1280;
const uint32_t windowHeight = 720;

HINSTANCE windowInstance;
HWND window;
```

In this section we'll be writing the body this method:

```
void initWindow(HINSTANCE hInstance) {}
```

Don't worry about `hInstance` for now. It is passed from the `WinMain` method we'll write later on. To setup our window, we'll need to register it with Windows, but first, we need to create a `WNDCLASSEX` object to pass during registration.

```
typedef struct WNDCLASSEX {
  UINT      cbSize;
  UINT      style;
  WNDPROC   lpfnWndProc;
  int       cbClsExtra;
  int       cbWndExtra;
  HINSTANCE hInstance;
  HICON     hIcon;
  HCURSOR   hCursor;
  HBRUSH    hbrBackground;
  LPCTSTR   lpszMenuName;
  LPCTSTR   lpszClassName;
  HICON     hIconSm;
} WNDCLASSEX, *PWNDCLASSEX;
```

Documentation for `WNDCLASSEX`:

- cbSize is the size, in bytes, of this structure. Set this member to sizeof(WNDCLASSEX). Be sure to set this member before calling the GetClassInfoEx function.
- style is the class style(s). This member can be any combination of the Class Styles.
- lpfnWndProc is a pointer to the window procedure. You must use the CallWindowProc function to call the window procedure.
- cbClsExtra is the number of extra bytes to allocate following the window-class structure.
- cbWndExtra is the number of extra bytes to allocate following the window instance.
- hInstance is a handle to the instance that contains the window procedure for the class.
- hIcon is a handle to the class icon. This member must be a handle to an icon resource. If this member is NULL, the system provides a default icon.
- hCursor is a handle to the class cursor.
- hbrBackground A handle to the class background brush.
- lpszMenuName is a pointer to a null-terminated character string that specifies the resource name of the class menu, as the name appears in the resource file. If you use an integer to identify the menu, use the MAKEINTRESOURCE macro.
- lpszClassName is a pointer to a null-terminated string or is an atom.
- hIconSm is a handle to a small icon that is associated with the window class.

Usage for WNDCLASSEX:

```
WNDCLASSEX wcex;

wcex.cbSize = sizeof(WNDCLASSEX);
wcex.style = CS_HREDRAW | CS_VREDRAW;
wcex.lpfnWndProc = WndProc;
wcex.cbClsExtra = 0;
wcex.cbWndExtra = 0;
wcex.hInstance = hInstance;
wcex.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_APPLICATION));
wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
wcex.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
wcex.lpszMenuName = NULL;
wcex.lpszClassName = applicationName;
wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_APPLICATION));
```

Now, we can make a call to RegisterClassEx to get Windowss to register the Window class.

Definition for RegisterClassEx:

```
ATOM WINAPI RegisterClassEx(
  const WNDCLASSEX *lpwcx
);
```

Documentation for RegisterClassEx:

- lpwcx is a pointer to a WNDCLASSEX structure. You must fill the structure with the appropriate class attributes before passing it to the function.

Usage for RegisterClassEx:

Calling RegisterClassEx returns NULL upon failure so we should make sure we check for that.

```
if (!RegisterClassEx(&wcex))
  exitOnError("Failed to register window");
```

Assuming all has gone well, we can set our windowInstance variable.

```
windowInstance = hInstance;
```

While it's not required, I'd recommend centering the window on the screen upon showing it. We'll need to use the `GetSystemMetrics` method to get two values:

- Screen width (`SM_CXSCREEN`)
- Screen height (`SM_CYSCREEN`)

Once we have these, we can do the math required to center the window:

$$W_{left} = S_{width}/2 - W_{width}/2$$

$$W_{top} = S_{height}/2 - W_{height}/2$$

To get this information and compute the window's left and top positions, we can write the following code:

```
int screenWidth = GetSystemMetrics(SM_CXSCREEN);
int screenHeight = GetSystemMetrics(SM_CYSCREEN);
int windowLeft = screenWidth / 2 - windowWidth / 2;
int windowTop = screenHeight / 2 - windowHeight / 2;
```

Finally, we can call Window's `CreateWindow` method. This will, as the name suggests, create the window like we want. We'll specify dimensions, location, and other parameters.

Definition for `CreateWindow`:

```
HWND WINAPI CreateWindow(
  LPCTSTR  lpClassName,
  LPCTSTR   lpWindowName,
  DWORD     dwStyle,
  int       x,
  int       y,
  int       nWidth,
  int       nHeight,
  HWND      hWndParent,
  HMENU     hMenu,
  HINSTANCE hInstance,
  LPVOID    lpParam
);
```

Documentation for `CreateWindow`:

- `lpClassName` is a null-terminated string or a class atom created by a previous call to the `RegisterClass` or `RegisterClassEx` function. The atom must be in the low-order word of `lpClassName`; the high-order word must be zero. If `lpClassName` is a string, it specifies the window class name.
- `lpWindowName` is the window name. If the window style specifies a title bar, the window title pointed to by lpWindowName is displayed in the title bar.
- `dwStyle` is the style of the window being created. This parameter can be a combination of the window style values.
- `x` is the initial horizontal position of the window.
- `y` is the initial vertical position of the window.
- `nWidth` is the width, in device units, of the window.
- `nHeight` is the height, in device units, of the window.
- `hWndParent` is a handle to the parent or owner window of the window being created or `NULL` in our case.

- `hMenu` is a handle to a menu, or specifies a child-window identifier depending on the window style or `NULL` in our case.
- `hInstance` is a handle to the instance of the module to be associated with the window.
- `lpParam` is a pointer to a value to be passed to the window through the `CREATESTRUCT` structure (`lpCreateParams` member) pointed to by the `lParam` param of the `WM_CREATE` message or `NULL` in our case.

Usage for `CreateWindow`:

```
window = CreateWindow(
  applicationName,
  applicationName,
  WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS | WS_CLIPCHILDREN,
  windowX,
  windowY,
  windowWidth,
  windowHeight,
  NULL,
  NULL,
  windowInstance,
  NULL);
```

The `CreateWindow` method returns `NULL` upon failure. Let's deal with that possibility before we move on:

```
if (!window)
  exitOnError("Failed to create window");
```

Last, but not least, we should show the window, set it in the foreground, and focus it. Windows provides three methods that do exactly that. These are very self explanatory:

```
ShowWindow(window, SW_SHOW);
SetForegroundWindow(window);
SetFocus(window);
```

Window Process

For this section, we'll be writing the body of a method called `WndProc`. This is required by Windows to process window events.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam) {}
```

We need to destroy the window and tell Windows we quit if the user attempted to close the window. If we're told we need to paint, we'll simply update the window. If neither of those cases we're met, we'll use the default procedure to handle events we didn't process. You can do this like so:

```
switch (message) {
  case WM_DESTROY:
    DestroyWindow(hWnd);
    PostQuitMessage(0);
    break;
  case WM_PAINT:
    ValidateRect(hWnd, NULL);
    break;
  default:
    return DefWindowProc(hWnd, message, wParam, lParam);
    break;
}
```

Our Update Loop

For this section, we'll write the body of this method:

```
void VulkanSwapchain::renderLoop() {}
```

We're calling it `renderLoop` because later we'll make calls to rendering functions within it. For now, however, we're going to:

- Create a message, loop while we have Windows set it
- Windows translate it into a character message then add it to the thread queue
- Dispatch the message to the windows procedure.

While that sounds complicated, it can be done with just a few lines of code:

```
MSG message;

while (GetMessage(&message, NULL, 0, 0)) {
  TranslateMessage(&message);
  DispatchMessage(&message);
}
```

A New Entry-Point

This is our application's new entry-point. We will **not** be using your typical `int main()` entry-point. This is because Windows requires GUI applications to enter at `WinMain`. We need to:

- Create an instance of our class
- Call our `initWindow` method
- Call our `renderLoop`

Definition for `WinMain`:

```
int CALLBACK WinMain(
  HINSTANCE hInstance,
  HINSTANCE hPrevInstance,
  LPSTR     lpCmdLine,
  int       nCmdShow
);
```

Documentation for `WinMain`:

- `hInstance` is a handle to the current instance of the application.
- `hPrevInstance` is a handle to the previous instance of the application. This parameter is always `NULL`.
- `lpCmdLine` is the command line for the application, excluding the program name.
- `nCmdShow` controls how the window is to be shown.

Usage for `WinMain`:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine, int nCmdShow) {
  VulkanSwapchain ve = VulkanSwapchain();
  ve.initWindow(hInstance);
  ve.renderLoop();
}
```

Chapter 5

Working With Surfaces

This section is divided up by platform. You must begin with the sections below before clicking on platform. The code below is platform independent and must be included in order to create a surface.

Extensions

You may remember a while back that when we created our instance, we enabled a few extensions. These were platform specific extensions for surfaces. Because we did it already, there's no need to write that code now.

Procedures

Vulkan doesn't directly expose functions for all platforms. Thus, we'll have to query Vulkan for them at run-time. We'll be getting the function pointers at the instance level and device level using the `vkGetInstanceProcAddr` and `vkGetDeviceProcAddr` methods.

Definition for `vkGetInstanceProcAddr`:

```
PFN_vkVoidFunction vkGetInstanceProcAddr(
  VkInstance  instance,
  const char* pName);
```

Documentation for `vkGetInstanceProcAddr`:

- `instance` is the instance that the function pointer will be compatible with.
- `pName` is the name of the command to obtain.

Usage for `vkGetInstanceProcAddr`:

```
#define GET_INSTANCE_PROC_ADDR(inst, entry)                              \
  {                                                                      \
    fp##entry = (PFN_vk##entry)vkGetInstanceProcAddr(inst, "vk" #entry); \
    if (!fp##entry)                                                      \
      exitOnError("vkGetInstanceProcAddr failed to find vk" #entry);     \
  }
```

Definition for `vkGetDeviceProcAddr`:

```
PFN_vkVoidFunction vkGetDeviceProcAddr(
  VkDevice    device,
  const char* pName);
```

Documentation for `vkGetDeviceProcAddr`:

- `device` is the logical device that provides the function pointer.
- `pName` is the name of any Vulkan command whose first parameter is one of
- `VkDevice`
- `VkQueue`
- `VkCommandBuffer`

Usage for `vkGetDeviceProcAddr`:

```
#define GET_DEVICE_PROC_ADDR(dev, entry)                            \
  {                                                                 \
    fp##entry = (PFN_vk##entry)vkGetDeviceProcAddr(dev, "vk" #entry); \
    if (!fp##entry)                                                 \
      exitOnError("vkGetDeviceProcAddr failed to find vk" #entry);    \
  }
```

These are based on the macros from the GLFW Vulkan example here. The reason we're using macros is because it would be tedious to repeat this for every time we want a function pointer.

Using the Macros

The reason we wrote the macros was to get function pointers, but I feel the need to explain what these function pointers are for. Basically, Vulkan has a concept like EGL. Vulkan provides us an interface between itself (the API) and our platform's windowing system. This is not visible to us. We instead work with abstractions over the systems Vulkan targets. The GPU, like with EGL, will be able to tell us extension support and capabilities of the system.

We'll be using the macros to verify the windowing system has support for surfaces. We'll also check for capabilities, supported formats, and presentation modes. You can learn more about the **Window System Integration / Interface (WSI)** here on page 9. We'll be asking for four different function pointers. Here's the piece of code we'll be using:

```
GET_INSTANCE_PROC_ADDR(instance, GetPhysicalDeviceSurfaceSupportKHR);
GET_INSTANCE_PROC_ADDR(instance, GetPhysicalDeviceSurfaceCapabilitiesKHR);
GET_INSTANCE_PROC_ADDR(instance, GetPhysicalDeviceSurfaceFormatsKHR);
GET_INSTANCE_PROC_ADDR(instance, GetPhysicalDeviceSurfacePresentModesKHR);
GET_DEVICE_PROC_ADDR(device, CreateSwapchainKHR);
GET_DEVICE_PROC_ADDR(device, DestroySwapchainKHR);
GET_DEVICE_PROC_ADDR(device, GetSwapchainImagesKHR);
GET_DEVICE_PROC_ADDR(device, AcquireNextImageKHR);
GET_DEVICE_PROC_ADDR(device, QueuePresentKHR);
```

We also need to add a few members of our `VulkanSwapchain` class:

```
PFN_vkGetPhysicalDeviceSurfaceSupportKHR fpGetPhysicalDeviceSurfaceSupportKHR;
PFN_vkGetPhysicalDeviceSurfaceCapabilitiesKHR
    fpGetPhysicalDeviceSurfaceCapabilitiesKHR;
PFN_vkGetPhysicalDeviceSurfaceFormatsKHR fpGetPhysicalDeviceSurfaceFormatsKHR;
PFN_vkGetPhysicalDeviceSurfacePresentModesKHR
    fpGetPhysicalDeviceSurfacePresentModesKHR;
```

```
PFN_vkCreateSwapchainKHR fpCreateSwapchainKHR;
PFN_vkDestroySwapchainKHR fpDestroySwapchainKHR;
PFN_vkGetSwapchainImagesKHR fpGetSwapchainImagesKHR;
PFN_vkAcquireNextImageKHR fpAcquireNextImageKHR;
PFN_vkQueuePresentKHR fpQueuePresentKHR;
```

These variables will be written to by our macros so we have the function pointers for future use. We'll make use of the function pointers later in this chapter.

Platform Specific Code

Now, please visit your platform (or all platforms) to get the specific code. You'll need this to create a surface.

Surface Creation on Microsoft Windows

We're going to be writing the Windows specific code for getting a surface in this section.

Surface Create Information

Before we create a surface, we must specify information ahead of time like most Vulkan objects. We'll be using `VkWin32SurfaceCreateInfoKHR`.

Definition for `VkWin32SurfaceCreateInfoKHR`:

```
typedef struct VkWin32SurfaceCreateInfoKHR {
  VkStructureType              sType;
  const void*                  pNext;
  VkWin32SurfaceCreateFlagsKHR flags;
  HINSTANCE                    hinstance;
  HWND                         hwnd;
} VkWin32SurfaceCreateInfoKHR
```

Documentation for `VkWin32SurfaceCreateInfoKHR`:

- `sType` is the type of this structure and must be `VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR`.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `hinstance` and hwnd are the `WIN32 HINSTANCE` and `HWND` for the window to associate the surface with.

Usage for `VkWin32SurfaceCreateInfoKHR`:

```
VkWin32SurfaceCreateInfoKHR surfaceCreateInfo;
surfaceCreateInfo.sType = VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;
surfaceCreateInfo.pNext = NULL;
surfaceCreateInfo.flags = 0;
surfaceCreateInfo.hinstance = windowInstance;
surfaceCreateInfo.hwnd = window;
```

Creating the Surface

Now we can create the surface. We can make a call to the `vkCreateWin32SurfaceKHR` method.

Definition for `vkCreateWin32SurfaceKHR`:

```
VkResult vkCreateWin32SurfaceKHR(
  VkInstance                         instance,
  const VkWin32SurfaceCreateInfoKHR* pCreateInfo,
  const VkAllocationCallbacks*       pAllocator,
  VkSurfaceKHR*                      pSurface
)
```

Documentation for `vkCreateWin32SurfaceKHR`:

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to an instance of the `VkWin32SurfaceCreateInfoKHR` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available.
- `pSurface` points to a `VkSurfaceKHR` handle in which the created surface object is returned.

Usage for `vkCreateWin32SurfaceKHR`:

```
VkResult result =
    vkCreateWin32SurfaceKHR(instance, &surfaceCreateInfo, NULL, &surface);
assert(result == VK_SUCCESS);
```

Please go back to the Chapter 5 page to read about determining the color formats and color spaces for the surface.

Surface Creation on Linux

We're going to be writing the Linux specific code for getting a surface in this section. While this code may work on another operating system that uses the XCB library, I cannot guarantee it will.

Surface Create Information

Before we create a surface, we must specify information ahead of time like most Vulkan objects. We'll be using `VkXcbSurfaceCreateInfoKHR`.

Definition for `VkXcbSurfaceCreateInfoKHR`:

```
typedef struct VkXcbSurfaceCreateInfoKHR {
  VkStructureType              sType;
  const void*                  pNext;
  VkXcbSurfaceCreateFlagsKHR   flags;
  xcb_connection_t*            connection;
  xcb_window_t                 window;
} VkXcbSurfaceCreateInfoKHR;
```

Documentation for `VkXcbSurfaceCreateInfoKHR`:

- `sType` is the type of this structure and must be `VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR`.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.

- `connection` is a pointer to an `xcb_connection_t` to the X server.
- `window` is the `xcb_window_t` for the X11 window to associate the surface with.

Usage for `VkXcbSurfaceCreateInfoKHR`:

```
VkXcbSurfaceCreateInfoKHR surfaceCreateInfo = {};
surfaceCreateInfo.sType = VK_STRUCTURE_TYPE_XCB_SURFACE_CREATE_INFO_KHR;
surfaceCreateInfo.pNext = NULL;
surfaceCreateInfo.flags = 0;
surfaceCreateInfo.connection = connection;
surfaceCreateInfo.window = window;
```

Creating the Surface

Now we can create the surface. We'll be calling the `vkCreateXcbSurfaceKHR` method to do so.

Definition for `vkCreateXcbSurfaceKHR`:

```
VkResult vkCreateXcbSurfaceKHR(
  VkInstance                        instance,
  const VkXcbSurfaceCreateInfoKHR*  pCreateInfo,
  const VkAllocationCallbacks*      pAllocator,
  VkSurfaceKHR*                     pSurface);
```

Documentation for `vkCreateXcbSurfaceKHR`:

- `instance` is the instance to associate the surface with.
- `pCreateInfo` is a pointer to an instance of the `VkXcbSurfaceCreateInfoKHR` structure containing parameters affecting the creation of the surface object.
- `pAllocator` is the allocator used for host memory allocated for the surface object when there is no more specific allocator available.
- `pSurface` points to a `VkSurfaceKHR` handle in which the created surface object is returned.

Usage for ‘vkCreateXcbSurfaceKHR:

```
VkResult result =
    vkCreateXcbSurfaceKHR(instance, &surfaceCreateInfo, NULL, &surface);
assert(result == VK_SUCCESS);
```

Please go back to the Chapter 5 page to read about determining the color formats and color spaces for the surface.

Queues

In Vulkan, we have a concept called command buffers. We'll get to this later, but for now, all you need to know is they make use of queues for executing operations. Queues are specific in the operations they support. Because we'll be rendering a 3D scene, we'll be looking to use a queue that supports graphics operations. And, because we want to display the images, we'll need to verify we have presentation support.

Checking Graphics / Present Support

In this section, we're going to find a queue that supports both graphics operations and presenting images. But first, we'll need to get the number of queues to store properties in. We'll be using `vkGetPhysicalDeviceQueueFamilyProperties`.

Definition for `vkGetPhysicalDeviceQueueFamilyProperties`:

```
void vkGetPhysicalDeviceQueueFamilyProperties(
  VkPhysicalDevice         physicalDevice,
  uint32_t*                pQueueFamilyPropertyCount,
  VkQueueFamilyProperties* pQueueFamilyProperties);
```

Documentation for `vkGetPhysicalDeviceQueueFamilyProperties`:

- `physicalDevice` is the handle to the physical device whose properties will be queried.
- `pQueueFamilyPropertyCount` is a pointer to an integer related to the number of queue families available or queried.
- `pQueueFamilyProperties` is either `NULL` or a pointer to an array of `VkQueueFamilyProperties` structures.

Usage for `vkGetPhysicalDeviceQueueFamilyProperties`:

Per usual, we'll call it with `NULL` as the last argument to get the number of queues before we allocate memory.

```
uint32_t queueCount = 0;
vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueCount, NULL);

assert(queueCount >= 1);

std::vector<VkQueueFamilyProperties> queueProperties(queueCount);
vkGetPhysicalDeviceQueueFamilyProperties(physicalDevice, &queueCount,
                                         queueProperties.data());
```

Now let's add a variable in our `VulkanExample` class called `queueIndex` to store the index of the queue we want:

```
uint32_t queueIndex;
```

We'll make the default value `UINT32_MAX` so we can later check if we found any suitable queue. Next, we need to look for a device queue that allows for drawing images and presenting images like I mentioned before. You can use two different queues, but we'll discuss that in a later chapter. We can use the `fpGetPhysicalDeviceSurfaceSupportKHR` function pointer from earlier. The definition is the same as `vkGetPhysicalDeviceSurfaceSupportKHR`.

Definition for `vkGetPhysicalDeviceSurfaceSupportKHR`:

```
VkResult vkGetPhysicalDeviceSurfaceSupportKHR(
  VkPhysicalDevice physicalDevice,
  uint32_t queueFamilyIndex,
  VkSurfaceKHR surface,
  VkBool32* pSupported);
```

Documentation for `vkGetPhysicalDeviceSurfaceSupportKHR`:

- `physicalDevice` is the physical device.
- `queueFamilyIndex` is the queue family.
- `surface` is the surface.
- `pSupported` is a pointer to a `VkBool32`, which is set to `VK_TRUE` to indicate support, and `VK_FALSE` otherwise.

Usage for `vkGetPhysicalDeviceSurfaceSupportKHR`:

```
std::vector<VkBool32> supportsPresenting(queueCount);

for (uint32_t i = 0; i < queueCount; i++) {
  fpGetPhysicalDeviceSurfaceSupportKHR(physicalDevice, i, surface,
                                       &supportsPresenting[i]);
}
```

To check if our queue supports graphics operations, we can check if the queue flag contains `VK_QUEUE_GRAPHICS_BIT`. You may or may not be aware that this can be done using the & operator. We can use the following syntax:

```
if ((QUEUE_FLAG & VK_QUEUE_GRAPHICS_BIT) != 0)
  // We support graphics operations!
```

Now, let's finish the body of the loop:

```
if ((queueProperties[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) != 0) {
  if (supportsPresenting[i] == VK_TRUE) {
    queueIndex = i;
    break;
  }
}
```

If we didn't find anything, the value of `queueIndex` will still be `UINT32_MAX`. Let's handle that possibility before moving on:

```
assert(queueIndex != UINT32_MAX);
```

Color Formats and Color Spaces

For rendering purposes, we'll need some information on the surface formats our device supports. Specifically, we're going to check right now for color support. Vulkan breaks up this into two categories: color formats and color spaces. Color formats can describe the number of components, size of components, compression types, etc. In contrast, color spaces tells the Vulkan implementation how to interpret that data. For example, if we are telling Vulkan we have an RGBA image, it would need to understand what `0` and `255` mean. A color space describes the range of colors or **gamut** if you prefer.

A common and well supported color format in Vulkan is `VK_FORMAT_B8G8R8A8_UNORM`. You can find documentation on available formats here. Note, that page is a stub and lacks full documentation right now. We can use `fpGetPhysicalDeviceSurfaceFormatsKHR` which has the definition is the same as `vkGetPhysicalDeviceSurfaceFormatsKHR`.

Definition for `vkGetPhysicalDeviceSurfaceFormatsKHR`:

```
VkResult vkGetPhysicalDeviceSurfaceFormatsKHR(
  VkPhysicalDevice     physicalDevice,
  VkSurfaceKHR         surface,
  uint32_t*            pSurfaceFormatCount,
  VkSurfaceFormatKHR*  pSurfaceFormats);
```

Documentation for `vkGetPhysicalDeviceSurfaceFormatsKHR`:

- `physicalDevice` is the physical device that will be associated with the swapchain to be created.
- `surface` is the surface that will be associated with the swapchain.
- `pSurfaceFormatCount` is a pointer to an integer related to the number of format pairs available or queried.
- `pSurfaceFormats` is either `NULL` or a pointer to an array of `VkSurfaceFormatKHR` structures.

Usage for `vkGetPhysicalDeviceSurfaceFormatsKHR`:

```
uint32_t formatCount = 0;
result = fpGetPhysicalDeviceSurfaceFormatsKHR(physicalDevice, surface,
                                              &formatCount, NULL);
assert(result == VK_SUCCESS);
assert(formatCount >= 1);
```

Now we can get the actual formats and verify success again:

```
std::vector<VkSurfaceFormatKHR> surfaceFormats(formatCount);
result = fpGetPhysicalDeviceSurfaceFormatsKHR(
    physicalDevice, surface, &formatCount, surfaceFormats.data());
  assert(result == VK_SUCCESS);
```

Now that we've found the formats our device supports, we can go ahead and set them. Let's add two variables to our `VulkanExample` class:

```
VkFormat colorFormat;
VkColorSpaceKHR colorSpace;
```

Here is the process we should follow when choosing a color format and color space:

- Check if we support **only** one format and that format is `VK_FORMAT_UNDEFINED`
- Yes: We should set our own format because the device does not default to any
- No: Take the first color format we support.
- Take the first color space that's available

The code would look like this:

```
if (formatCount == 1 && surfaceFormats[0].format == VK_FORMAT_UNDEFINED)
  colorFormat = VK_FORMAT_B8G8R8A8_UNORM;
else
  colorFormat = surfaceFormats[0].format;

colorSpace = surfaceFormats[0].colorSpace;
```

Chapter 6

Getting Started With Swapchains

You might have noticed that in a previous section, we made used this value: `VK_KHR_SWAPCHAIN_EXTENSION_NAME`. So, what *is* a swap chain? It is essentially an **array of images** ready to be presented. One use is frame rate control. Using two buffers is called **double buffering**. The GPU renders completely to a single frame and then displays it. Once it has finished drawing the first frame, it begins drawing the second frame. This occurs even if we're rendering above the rate we're supposed to. Let's assume we're rendering faster than the physical display can display our images. We would then wait and **flip** the second buffer onto the screen. By flipping the image onto the screen during the **vertical blanking interval**, we can write our data while the display is blank. When it refreshes, our image appears on the screen. Other techniques such as **triple buffering** exist.

Initializing the Swapchain

For the next few sections, we're going to be focusing on writing the `initSwapchain` method in our `VulkanSwapchain` class. It is defined simply as:

```
void VulkanSwapchain::initSwapchain() {}
```

Getting Physical Device Surface Capabilities

We created a surface in the last chapter. Now we need to check the surface resolution so we can later inform our swapchain. To get the resolution of the surface, we'll have to ask it for its capabilities. We'll be using `fpGetPhysicalDeviceSurfaceCapabilitiesKHR` which has the same definition as `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`.

Usage for `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`:

```
VkResult vkGetPhysicalDeviceSurfaceCapabilitiesKHR(
  VkPhysicalDevice            physicalDevice,
  VkSurfaceKHR                surface,
  VkSurfaceCapabilitiesKHR*   pSurfaceCapabilities);
```

Documentation for 'vkGetPhysicalDeviceSurfaceCapabilitiesKHR:

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `surface` is the surface that will be associated with the swapchain.
- `pSurfaceCapabilities` is a pointer to an instance of the `VkSurfaceCapabilitiesKHR` structure that will be filled with information.

Usage for `vkGetPhysicalDeviceSurfaceCapabilitiesKHR`:

Note that it takes a pointer to a `VkSurfaceCapabilitiesKHR` object. We'll have to create that to pass it in. Let's do that and verify we were successful:

```
VkSurfaceCapabilitiesKHR caps = {};
VkResult result = fpGetPhysicalDeviceSurfaceCapabilitiesKHR(
    physicalDevice, surface, &caps);
assert(result == VK_SUCCESS);
```

Vulkan Extents

Windows uses `RECT` to define properties of rectangles. Similarly, Vulkan uses `VkExtent2D` for the same purpose. The `VkExtent2D` object will be used later, but for now, let's check something. The `caps` variable has a `currentExtent` field. This informs us about the surface's size. In the case that either the `width` or `height` are `-1`, we'll need to set the extent size ourselves. Otherwise, we can just use `caps.currentExtent` for the swapchain. Let's see how this looks in code:

```
VkExtent2D swapchainExtent = {};

if (caps.currentExtent.width == -1 || caps.currentExtent.height == -1) {
  swapchainExtent.width = windowWidth;
  swapchainExtent.height = windowHeight;
} else {
  swapchainExtent = caps.currentExtent;
}
```

Get Physical Device Surface Present Modes

In Vulkan, there are multiple ways images can be presented. We'll talk about the options later in this section, but for now, we need to figure out which are supported. We can use `fpGetPhysicalDeviceSurfacePresentModesKHR` to get the present modes as the name suggests. The definition is the same as `vkGetPhysicalDeviceSurfacePresentModesKHR`.

Definition for `vkGetPhysicalDeviceSurfacePresentModesKHR`:

```
VkResult vkGetPhysicalDeviceSurfacePresentModesKHR(
  VkPhysicalDevice   physicalDevice,
  VkSurfaceKHR       surface,
  uint32_t*          pPresentModeCount,
  VkPresentModeKHR*  pPresentModes);
```

Documentation for `vkGetPhysicalDeviceSurfacePresentModesKHR`:

- `physicalDevice` is the physical device that will be associated with the swapchain to be created, as described for `vkCreateSwapchainKHR`.
- `surface` is the surface that will be associated with the swapchain.
- `pPresentModeCount` is a pointer to an integer related to the number of format pairs available or queried, as described below.
- `pPresentModes` is either `NULL` or a pointer to an array of `VkPresentModeKHR` structures.

Usage for `vkGetPhysicalDeviceSurfacePresentModesKHR`:

```
uint32_t presentModeCount = 0;
result = fpGetPhysicalDeviceSurfacePresentModesKHR(
    physicalDevice, surface, &presentModeCount, NULL);
```

Before we move on, let's verify success. We should check the `result` and check make sure one or more present modes are available.

```
assert(result == VK_SUCCESS);
assert(presentModeCount >= 1);
```

Now let's go ahead and call the function again with our `vector`:

```
std::vector<VkPresentModeKHR> presentModes(presentModeCount);
result = fpGetPhysicalDeviceSurfacePresentModesKHR(
    physicalDevice, surface, &presentModeCount, presentModes.data());

assert(result == VK_SUCCESS);
```

Now we should take a look at the available present modes. There are a few different types:

- `VK_PRESENT_MODE_IMMEDIATE_KHR` - Our engine **will not ever** wait for the vertical blanking interval. This *may* result in visible tearing if our frame misses the interval and is presented too late.
- `VK_PRESENT_MODE_MAILBOX_KHR` - Our engine waits for the next vertical blanking interval to update the image. If we render another image, the image waiting to be displayed is overwritten.
- You can think of the underlying data structure as an array of images of length 1.
- `VK_PRESENT_MODE_FIFO_KHR` - Our engine waits for the next vertical blanking interval to update the image. If we've missed an interval, we wait until the next one. We will append already rendered images to the pending presentation queue. We follow the "first in first out" (FIFO) philosophy as the name suggests. There **will not** be any visible tearing.
- You can think of the underlying data structure as something similar to a heap.
- `VK_PRESENT_MODE_FIFO_RELAXED_KHR` - Our engine waits for the next vertical blanking interval to update the image. If we've missed the interval, we **do not** wait. We will append already rendered images to the pending presentation queue. We present as soon as possible. We follow the "first in first out" (FIFO) philosophy as the name suggests. This *may* result in tearing.
- You can think of the underlying data structure as something similar to a heap.

If you do not care about tearing, you might want `VK_PRESENT_MODE_IMMEDIATE_KHR`. However, if you want a low-latency tear-less presentation mode, you would choose `VK_PRESENT_MODE_MAILBOX_KHR`. Now let's look through our present modes and see if we can find `VK_PRESENT_MODE_MAILBOX_KHR`. If we find it, stop looking. Otherwise, if we see our next preference `VK_PRESENT_MODE_IMMEDIATE_KHR`, keep that. Finally, if we didn't find anything, we should continue with our default `VK_PRESENT_MODE_FIFO_KHR`. The code would look like this:

```
VkPresentModeKHR presentMode = VK_PRESENT_MODE_FIFO_KHR;

for (uint32_t i = 0; i < presentModeCount; i++) {
  if (presentModes[i] == VK_PRESENT_MODE_MAILBOX_KHR) {
    presentMode = VK_PRESENT_MODE_MAILBOX_KHR;
    break;
  }

  if (presentModes[i] == VK_PRESENT_MODE_IMMEDIATE_KHR)
    presentMode = VK_PRESENT_MODE_IMMEDIATE_KHR;
}
```

Swapchain Create Information

Next up, we're going to prepare the information needed to create our VkSwapchainKHR.

Definition for VkSwapchainCreateInfoKHR:

```
typedef struct VkSwapchainCreateInfoKHR {
  VkStructureType sType;
  const void* pNext;
  VkSwapchainCreateFlagsKHR flags;
  VkSurfaceKHR surface;
  uint32_t minImageCount;
  VkFormat imageFormat;
  VkColorSpaceKHR imageColorSpace;
  VkExtent2D imageExtent;
  uint32_t imageArrayLayers;
  VkImageUsageFlags imageUsage;
  VkSharingMode imageSharingMode;
  uint32_t queueFamilyIndexCount;
  const uint32_t* pQueueFamilyIndices;
  VkSurfaceTransformFlagBitsKHR preTransform;
  VkCompositeAlphaFlagBitsKHR compositeAlpha;
  VkPresentModeKHR presentMode;
  VkBool32 clipped;
  VkSwapchainKHR oldSwapchain;
} VkSwapchainCreateInfoKHR
```

Documentation for VkSwapchainCreateInfoKHR:

- sType is the type of this structure and must be VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR.
- pNext is NULL or a pointer to an extension-specific structure.
- flags is reserved for future use, and must be zero.
- surface is the surface that the swapchain will present images to.
- minImageCount is the minimum number of presentable images that the application needs. The platform will either create the swapchain with at least that many images, or will fail to create the swapchain.
- imageFormat is a VkFormat that is valid for swapchains on the specified surface.
- imageColorSpace is a VkColorSpaceKHR that is valid for swapchains on the specified surface.
- imageExtent is the size (in pixels) of the swapchain. Behavior is platform-dependent when the image extent does not match the surface's currentExtent as returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR.
- imageArrayLayers is the number of views in a multiview/stereo surface. For non-stereoscopic-3D applications, this value is 1.
- imageUsage is a bitfield of VkImageUsageFlagBits, indicating how the application will use the swapchain's presentable images.
- imageSharingMode is the sharing mode used for the images of the swapchain.
- queueFamilyIndexCount is the number of queue families having access to the images of the swapchain in case imageSharingMode is VK_SHARING_MODE_CONCURRENT.
- pQueueFamilyIndices is an array of queue family indices having access to the images of the swapchain in case imageSharingMode is VK_SHARING_MODE_CONCURRENT.
- preTransform is a bitfield of VkSurfaceTransformFlagBitsKHR, describing the transform, relative to the presentation engine's natural orientation, applied to the image content prior to presentation. If it does not match the currentTransform value returned by vkGetPhysicalDeviceSurfaceCapabilitiesKHR, the presentation engine will transform the image content as part of the presentation operation.

- compositeAlpha is a bitfield of VkCompositeAlphaFlagBitsKHR, indicating the alpha compositing mode to use when this surface is composited together with other surfaces on certain window systems.
- presentMode is the presentation mode the swapchain will use. A swapchain's present mode determines how incoming present requests will be processed and queued internally.
- clipped indicates whether the Vulkan implementation is allowed to discard rendering operations that affect regions of the surface which aren't visible.

Usage for VkSwapchainCreateInfoKHR:

That's quite a definition. Before we can fill in the values, we'll need to figure out minImageCount. First let's check verify our surface supports images:

```
assert(caps.maxImageCount >= 1);
```

Now let's say we want at least one image:

```
uint32_t imageCount = caps.minImageCount + 1;
```

If we asked for more images than the implementation supports, we'll have to settle for less:

```
if (imageCount > caps.maxImageCount)
  imageCount = caps.maxImageCount;
```

Finally, we can put it all together:

```
VkSwapchainCreateInfoKHR swapchainCreateInfo = {};
swapchainCreateInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
swapchainCreateInfo.surface = surface;
swapchainCreateInfo.minImageCount = imageCount;
swapchainCreateInfo.imageFormat = colorFormat;
swapchainCreateInfo.imageColorSpace = colorSpace;
swapchainCreateInfo.imageExtent = { swapchainExtent.width, swapchainExtent.height };
swapchainCreateInfo.imageArrayLayers = 1;
swapchainCreateInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
swapchainCreateInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
swapchainCreateInfo.queueFamilyIndexCount = 1;
swapchainCreateInfo.pQueueFamilyIndices = { 0 };
swapchainCreateInfo.preTransform = VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
swapchainCreateInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
swapchainCreateInfo.presentMode = presentMode;
```

Creating the Swapchain

This function will simply take in our swapchainCreateInfo and create the swapchain with that configuration.

Definition for vkCreateSwapchainKHR:

```
VkResult vkCreateSwapchainKHR(
  VkDevice                                    device,
  const VkSwapchainCreateInfoKHR*             pCreateInfo,
  const VkAllocationCallbacks*                pAllocator,
  VkSwapchainKHR*                             pSwapchain);
```

Documentation for vkCreateSwapchainKHR:

- `device` is the device to create the swapchain for.
- `pCreateInfo` is a pointer to an instance of the `VkSwapchainCreateInfoKHR` structure specifying the parameters of the created swapchain.
- `pAllocator` is the allocator used for host memory allocated for the swapchain object when there is no more specific allocator available.
- `pSwapchain` is a pointer to a `VkSwapchainKHR` handle in which the created swapchain object will be returned.

Usage for `vkCreateSwapchainKHR`:

```
result = fpCreateSwapchainKHR(device, &swapchainCreateInfo, NULL, &swapchain);
assert(result == VK_SUCCESS);
```

Getting Swapchain Images

We will need to get the available images from the swapchain. In a later section of this chapter, we'll actually get them ready for use, but right now, let's focus on this part. We'll be using a function pointer we got earlier called `fpGetSwapchainImagesKHR`. The definition is the same as `vkGetSwapchainImagesKHR`.

Definition for `vkGetSwapchainImagesKHR`:

```
VkResult vkGetSwapchainImagesKHR(
  VkDevice device,
  VkSwapchainKHR swapchain,
  uint32_t* pSwapchainImageCount,
  VkImage* pSwapchainImages);
```

Documentation for `vkGetSwapchainImagesKHR`:

- `device` is the device associated with swapchain.
- `swapchain` is the swapchain to query.
- `pSwapchainImageCount` is a pointer to an integer related to the number of format pairs available or queried, as described below.
- `pSwapchainImages` is either `NULL` or a pointer to an array of `VkSwapchainImageKHR` structures.

Usage for `vkGetSwapchainImagesKHR`:

```
result = fpGetSwapchainImagesKHR(device, swapchain, &imageCount, NULL);
assert(result == VK_SUCCESS);
assert(imageCount > 0);
```

We'll need to make a new `struct` to contain a few things. Let's look at it:

```
struct SwapChainBuffer {
  VkImage image;
  VkImageView view;
  VkFramebuffer frameBuffer;
};
```

This will become more relevant later on when we get closer to rendering. For now, we'll create two variables in the `VulkanSwapchain` class. These will be:

```
std::vector<VkImage> images;
std::vector<SwapChainBuffer> buffers;
```

Let's make sure we `resize` these to fit the number of images we'll get back:

```
images.resize(imageCount);
buffers.resize(imageCount);
```

And of course, we'll call the function again and check for errors:

```
result =
    fpGetSwapchainImagesKHR(device, swapchain, &imageCount, images.data());
assert(result == VK_SUCCESS);
```

Chapter 7

Image Layouts

In this chapter, we'll be writing a `setImageLayout` method. We'll make sure to come back to `initSwapchain` later.

```
void setImageLayout(VkCommandBuffer cmdBuffer, VkImage image,
                    VkImageAspectFlags aspects,
                    VkImageLayout oldLayout,
                    VkImageLayout newLayout) {}
```

This will take a `VkCommandBuffer` and a `VkImage` whose image layout we want to set. While it's not necessary to build out this method, it will be useful later on. We'll also take in two `VkImageLayouts`.

Image Memory Barriers

In Vulkan, we have a new concept called barriers which are called `VkImageMemoryBarrier`. They make sure our operations done on the GPU occur in a particular order which assure we get the expected result. A barrier separates two operations in a queue: before the barrier and after the barrier. Work done before the barrier will always finish before it can be used again.

Definition for `VkImageMemoryBarrier`:

```
typedef struct VkImageMemoryBarrier {
  VkStructureType            sType;
  const void*                pNext;
  VkAccessFlags              srcAccessMask;
  VkAccessFlags              dstAccessMask;
  VkImageLayout              oldLayout;
  VkImageLayout              newLayout;
  uint32_t                   srcQueueFamilyIndex;
  uint32_t                   dstQueueFamilyIndex;
  VkImage                    image;
  VkImageSubresourceRange    subresourceRange;
} VkImageMemoryBarrier;
```

Documentation for `VkImageMemoryBarrier`:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `srcAccessMask` is a mask of the classes of memory accesses performed by the first set of commands that will participate in the dependency.

- `dstAccessMask` is a mask of the classes of memory accesses performed by the second set of commands that will participate in the dependency.
- `oldLayout` describes the current layout of the image subresource(s).
- `newLayout` describes the new layout of the image subresource(s).
- `srcQueueFamilyIndex` is the queue family that is relinquishing ownership of the image subresource(s) to another queue, or `VK_QUEUE_FAMILY_IGNORED` if there is no transfer of ownership).
- `dstQueueFamilyIndex` is the queue family that is acquiring ownership of the image subresource(s) from another queue, or `VK_QUEUE_FAMILY_IGNORED` if there is no transfer of ownership).
- `image` is a handle to the image whose backing memory is affected by the barrier.
- `subresourceRange` describes an area of the backing memory for image, as well as the set of subresources whose image layouts are modified.

Usage for `VkImageMemoryBarrier`:

```
VkImageMemoryBarrier imageBarrier = {};
imageBarrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
imageBarrier.pNext = NULL;
imageBarrier.oldLayout = oldLayout;
imageBarrier.newLayout = newLayout;
imageBarrier.image = image;
imageBarrier.subresourceRange.aspectMask = aspects;
imageBarrier.subresourceRange.baseMipLevel = 0;
imageBarrier.subresourceRange.levelCount = 1;
imageBarrier.subresourceRange.layerCount = 1;
```

Notice we left our two parts: `srcAccessMask` and `dstAccessMask`. Depending on the values `oldLayout` and `newLayout` take, we'll change how we set up our `VkImageMemoryBarrier`. Here is the way I handle the transition between the two layouts:

```
switch (oldLayout) {
  case VK_IMAGE_LAYOUT_PREINITIALIZED:
    imageBarrier.srcAccessMask =
        VK_ACCESS_HOST_WRITE_BIT | VK_ACCESS_TRANSFER_WRITE_BIT;
    break;
  case VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL:
    imageBarrier.srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
    break;
  case VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL:
    imageBarrier.srcAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    break;
  case VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL:
    imageBarrier.srcAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
    break;
  case VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL:
    imageBarrier.srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
    break;
}

switch (newLayout) {
  case VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL:
    imageBarrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
    break;
  case VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL:
    imageBarrier.srcAccessMask |= VK_ACCESS_TRANSFER_READ_BIT;
    imageBarrier.dstAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
    break;
  case VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL:
    imageBarrier.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
    imageBarrier.srcAccessMask = VK_ACCESS_TRANSFER_READ_BIT;
    break;
  case VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL:
    imageBarrier.dstAccessMask |=
        VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
    break;
  case VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL:
    imageBarrier.srcAccessMask =
        VK_ACCESS_HOST_WRITE_BIT | VK_ACCESS_TRANSFER_WRITE_BIT;
    imageBarrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
    break;
}
```

While there is a `VK_IMAGE_LAYOUT_GENERAL` that will work in all cases, it's not always optimal. For example, we have different layouts meant for:

- Color attachments (framebuffer)
- Depth stencils attachments (framebuffer)
- Shader reading via sampling

Images will start as `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED` depending on which you pick. Note that moving from `VK_IMAGE_LAYOUT_UNDEFINED` to another layout may not preserve the existing data. However, moving from `VK_IMAGE_LAYOUT_PREINITIALIZED` to another gurantees the data is preserved. The documentation says that any layout can be used for `oldLayout` while `newLayout` cannot use `VK_IMAGE_LAYOUT_UNDEFINED` or `VK_IMAGE_LAYOUT_PREINITIALIZED`. You can find the documentation I'm reading from here.

Recording Commands Image Layout Commands

Before we can finish our `setImageLayout` method, we need to call `vkCmdPipelineBarrier`. This will record the command and insert our execution dependencies and memory dependencies between two sets of commands.

Definition of `vkCmdPipelineBarrier`:

```
void vkCmdPipelineBarrier(
  VkCommandBuffer              commandBuffer,
  VkPipelineStageFlags         srcStageMask,
  VkPipelineStageFlags         dstStageMask,
  VkDependencyFlags            dependencyFlags,
  uint32_t                     memoryBarrierCount,
  const VkMemoryBarrier*       pMemoryBarriers,
  uint32_t                     bufferMemoryBarrierCount,
  const VkBufferMemoryBarrier* pBufferMemoryBarriers,
  uint32_t                     imageMemoryBarrierCount,
  const VkImageMemoryBarrier*  pImageMemoryBarriers);
```

Documentation for `vkCmdPipelineBarrier`:

- `commandBuffer` is the command buffer into which the command is recorded.
- `srcStageMask` is a bitmask of `VkPipelineStageFlagBits` specifying a set of source pipeline stages.
- `dstStageMask` is a bitmask specifying a set of destination pipeline stages.
- `dependencyFlags` is a bitmask of `VkDependencyFlagBits`. The execution dependency is by-region if the mask includes VK_DEPENDENCY_BY_REGION_BIT‘.
- `memoryBarrierCount` is the length of the `pMemoryBarriers` array.
- `pMemoryBarriers` is a pointer to an array of `VkMemoryBarrier` structures.
- `bufferMemoryBarrierCount` is the length of the `pBufferMemoryBarriers` array.
- `pBufferMemoryBarriers` is a pointer to an array of `VkBufferMemoryBarrier` structures.
- `imageMemoryBarrierCount` is the length of the `pImageMemoryBarriers` array.
- `pImageMemoryBarriers` is a pointer to an array of `VkImageMemoryBarrier` structures.

The only arguments we're not sure about are `srcFlags` and `dstFlags`. We know we want our execution / memory dependencies to be staged at the top of the command buffer. So, we'll use `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` to notify Vulkan of our intentions. You can find more information on pipeline state flags like `VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT` here.

Usage for `vkCmdPipelineBarrier`:

```
VkPipelineStageFlagBits srcFlags = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
VkPipelineStageFlagBits dstFlags = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
vkCmdPipelineBarrier(cmdBuffer, srcFlags, dstFlags, 0, 0, NULL, 0, NULL, 1,
                     &imageBarrier);
```

Chapter 8

Image Views

For this chapter, we'll be back in our `initSwapchain` method. We'll start with a `for` loop where we will be setting image layouts and creating image views. It will look like this:

```
for (uint32_t i = 0; i < imageCount; i++) {
  // We'll be filling this out
}
```

Image View Create Information

In Vulkan, we don't directly access images from shaders for reading and writing. Instead, we make use of image views. We can use `VkImageViewCreateInfo` to specify certain traits for each image view we want to create. Essentially, image views wrap up image objects and can provide additional metadata.

Definition for `VkImageViewCreateInfo`:

```
typedef struct VkImageViewCreateInfo {
  VkStructureType            sType;
  const void*                pNext;
  VkImageViewCreateFlags     flags;
  VkImage                    image;
  VkImageViewType            viewType;
  VkFormat                   format;
  VkComponentMapping         components;
  VkImageSubresourceRange    subresourceRange;
} VkImageViewCreateInfo;
```

Documentation for `VkImageViewCreateInfo`:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `image` is a `VkImage` on which the view will be created.
- `viewType` is the type of the image view.
- `format` is a `VkFormat` describing the format and type used to interpret data elements in the image.
- `components` specifies a remapping of color components (or of depth or stencil components after they have been converted into color components).
- `subresourceRange` selects the set of mipmap levels and array layers to be accessible to the view.

Usage for `VkImageViewCreateInfo`:

```
VkImageViewCreateInfo imageCreateInfo = {};
imageCreateInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
imageCreateInfo.pNext = NULL;
imageCreateInfo.format = colorFormat;
imageCreateInfo.components = {}; // We'll get to this in a second!
imageCreateInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
imageCreateInfo.subresourceRange.baseMipLevel = 0;
imageCreateInfo.subresourceRange.levelCount = 1;
imageCreateInfo.subresourceRange.baseArrayLayer = 0;
imageCreateInfo.subresourceRange.layerCount = 1;
imageCreateInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
imageCreateInfo.flags = 0;
```

Notice I did not finish the `components` line. You can see this section for more information on component mapping. But, what we're doing is telling Vulkan how to map image components to vector components output by our shaders. We'll simply tell Vulkan we want our `(R, G, B, A)` images to map to a vector in the form `<R, G, B, A>`. We can do that like so:

```
imageCreateInfo.components = {
    VK_COMPONENT_SWIZZLE_R,
    VK_COMPONENT_SWIZZLE_G,
    VK_COMPONENT_SWIZZLE_B,
    VK_COMPONENT_SWIZZLE_A};
```

Unless you know what you're doing, just use RGBA values. Before we create the image, let's make sure we store it and call the `setImageLayout` method with it. We'll also finish filling out the `imageCreateInfo`:

```
buffers[i].image = images[i];
setImageLayout(cmdBuffer, buffers[i].image, VK_IMAGE_ASPECT_COLOR_BIT,
               VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_PRESENT_SRC_KHR);
imageCreateInfo.image = buffers[i].image;
```

Creating an Image View

To create our image view, we simply call `vkCreateImageView`.

Definition for `vkCreateImageView`:

```
VkResult vkCreateImageView(
  VkDevice                     device,
  const VkImageViewCreateInfo* pCreateInfo,
  const VkAllocationCallbacks* pAllocator,
  VkImageView*                 pView);
  device
```

Documentation for `vkCreateImageView`:

- `device` is the logical device that creates the image view.
- `pCreateInfo` is a pointer to an instance of the `VkImageViewCreateInfo` structure containing parameters to be used to create the image view.
- `pAllocator` controls host memory allocation.
- `pView` points to a `VkImageView` handle in which the resulting image view object is returned.

Usage for `vkCreateImageView`:

```
result =
    vkCreateImageView(device, &imageCreateInfo, NULL, &buffers[i].view);
assert(result == VK_SUCCESS);
```

Framebuffer Create Information

We're going to be creating framebuffers for every image in the swapchain. Thus, we'll continue writing the body of our `for` loop. Like most types in Vulkan, we'll need to create an info structure before we can create a framebuffer. This is called `VkFramebufferCreateInfo`.

Definition for `VkFramebufferCreateInfo`:

```
typedef struct VkFramebufferCreateInfo {
  VkStructureType             sType;
  const void*                 pNext;
  VkFramebufferCreateFlags    flags;
  VkRenderPass                renderPass;
  uint32_t                    attachmentCount;
  const VkImageView*          pAttachments;
  uint32_t                    width;
  uint32_t                    height;
  uint32_t                    layers;
} VkFramebufferCreateInfo;
```

Documentation for `VkFramebufferCreateInfo`:

- `sType` is the type of this structure.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `flags` is reserved for future use.
- `renderPass` is a render pass that defines what render passes the framebuffer will be compatible with.
- `attachmentCount` is the number of attachments.
- `pAttachments` is an array of `VkImageView` handles, each of which will be used as the corresponding attachment in a render pass instance. width, height and layers define the dimensions of the framebuffer.

Usage for `VkFramebufferCreateInfo`:

We'll set the width and height to that of the `swapchainExtent` from earlier.

```
VkFramebufferCreateInfo fbCreateInfo = {};
fbCreateInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
fbCreateInfo.attachmentCount = 1;
fbCreateInfo.pAttachments = &buffers[i].view;
fbCreateInfo.width = swapchainExtent.width;
fbCreateInfo.height = swapchainExtent.height;
fbCreateInfo.layers = 1;
```

Creating a Framebuffer

Now we can create each framebuffer. We'll call `vkCreateFramebuffer`.

Definition for `vkCreateFramebuffer`:

```
VkResult vkCreateFramebuffer(
  VkDevice                       device,
  const VkFramebufferCreateInfo* pCreateInfo,
  const VkAllocationCallbacks*   pAllocator,
  VkFramebuffer*                 pFramebuffer);
```

Documentation for `vkCreateFramebuffer`:

- `device` is the logical device that creates the framebuffer.
- `pCreateInfo` points to a `VkFramebufferCreateInfo` structure which describes additional information about framebuffer creation.
- `pAllocator` controls host memory allocation.
- `pFramebuffer` points to a `VkFramebuffer` handle in which the resulting framebuffer object is returned.

Usage for `vkCreateFramebuffer`:

```
result = vkCreateFramebuffer(device, &fbCreateInfo, NULL,
                             &buffers[i].frameBuffer);
assert(result == VK_SUCCESS);
```

That's all we'll need for our `initSwapchain` method! Now it's time for acquiring the next image in the swapchain and presenting images!

Chapter 9

Acquiring, Presenting, Cleaning Up

For this chapter, we'll be focusing on:

- Acquiring the next image in the swapchain
- Presenting images
- Cleaning up when we're done with the swapchain

Acquiring the Next Image

For this section, we'll be writing a small helper method. The definition looks like this:

```
void getSwapchainNext(VkSemaphore presentCompleteSemaphore, uint32_t buffer) {}
```

We already have the function pointer from earlier called `fpAcquireNextImageKHR`. This has the same definition as `vkAcquireNextImageKHR`.

Definition for `vkAcquireNextImageKHR`:

```
VkResult vkAcquireNextImageKHR(
  VkDevice       device,
  VkSwapchainKHR swapchain,
  uint64_t       timeout,
  VkSemaphore    semaphore,
  VkFence        fence,
  uint32_t*      pImageIndex);
```

Documentation for `vkAcquireNextImageKHR`:

- `device` is the device assocated with swapchain.
- `swapchain` is the swapchain from which an image is being acquired.
- `timeout` indicates how long the function waits, in nanoseconds, if no image is available.
- `semaphore` is `VK_NULL_HANDLE` or a semaphore to signal.
- `fence` is `VK_NULL_HANDLE` or a fence to signal.
- `pImageIndex` is a pointer to a `uint32_t` that is set to the index of the next image to use (i.e. an index into the array of images returned by `vkGetSwapchainImagesKHR`).

Usage for `vkAcquireNextImageKHR`:

```
VkResult result =
    fpAcquireNextImageKHR(device, swapchain, UINT64_MAX,
                              presentCompleteSemaphore, (VkFence)0, &buffer);
assert(result == VK_SUCCESS);
```

Swapchain Image Presentation

For this section and the next, we'll be writing the body of this method:

```
void swapchainPresent(VkCommandBuffer cmdBuffer, VkQueue queue,
                      uint32_t buffer) {}
```

In order for the swapchain to present images, we'll have to inform Vulkan of some things. We can use `VkPresentInfoKHR` to do this.

Definition for `VkPresentInfoKHR`:

```
typedef struct VkPresentInfoKHR {
  VkStructureType           sType;
  const void*               pNext;
  uint32_t                  waitSemaphoreCount;
  const VkSemaphore*        pWaitSemaphores;
  uint32_t                  swapchainCount;
  const VkSwapchainKHR*     pSwapchains;
  const uint32_t*           pImageIndices;
  VkResult*                 pResults;
} VkPresentInfoKHR;
```

Documentation for `VkPresentInfoKHR`:

- `sType` is the type of this structure and must be `VK_STRUCTURE_TYPE_PRESENT_INFO_KHR`.
- `pNext` is `NULL` or a pointer to an extension-specific structure.
- `waitSemaphoreCount` is the number of semaphores to wait for before issuing the present request. The number may be zero.
- `pWaitSemaphores`, if non-`NULL`, is an array of `VkSemaphore` objects with `waitSemaphoreCount` entries, and specifies the semaphores to wait for before issuing the present request.
- `swapchainCount` is the number of swapchains being presented to by this command.
- `pSwapchains` is an array of `VkSwapchainKHR` objects with `swapchainCount` entries. A given swapchain must not appear in this list more than once.
- `pImageIndices` is an array of indices into the array of each swapchain's presentable images, with `swapchainCount` entries. Each entry in this array identifies the image to present on the corresponding entry in the `pSwapchains` array.
- `pResults` is an array of `VkResult` typed elements with `swapchainCount` entries. Applications that don't need per-swapchain results can use `NULL` for pResults. If non-`NULL`, each entry in `pResults` will be set to the `VkResult` for presenting the swapchain corresponding to the same index in `pSwapchains`.

Usage for `VkPresentInfoKHR`:

```
VkPresentInfoKHR presentInfo = {};
presentInfo.sType = VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
presentInfo.pNext = NULL;
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains = &swapchain;
presentInfo.pImageIndices = &buffer;
```

Queue Presentation

As the last part of the `swapchainPresent` method, we actually get to present! We'll be using the function pointer from earlier called `fpQueuePresentKHR` which has the definition is the same as `vkQueuePresentKHR`.

Definition for `vkQueuePresentKHR`:

```
VkResult vkQueuePresentKHR(
    VkQueue                 queue,
    const VkPresentInfoKHR* pPresentInfo);
```

Documentation for `vkQueuePresentKHR`:

- `queue`is a queue that is capable of presentation to the target surface's platform on the same device as the image's swapchain.
- `pPresentInfo` is a pointer to an instance of the `VkPresentInfoKHR` structure specifying the parameters of the presentation.

Usage for `vkQueuePresentKHR`:

```
VkResult result = fpQueuePresentKHR(queue, &presentInfo);
assert(result == VK_SUCCESS);
```

Cleaning Up

For our new destructor, we will:

- Destroy all image views we stored
- Destroy the swapchain
- Destroy the surface
- Destroy the instance

That can be done easily with the following lines of code:

```
for (SwapChainBuffer buffer : buffers)
  vkDestroyImageView(device, buffer.view, NULL);

fpDestroySwapchainKHR(device, swapchain, NULL);
vkDestroySurfaceKHR(instance, surface, NULL);
vkDestroyInstance(instance, NULL);
```