# Using Vulkan with Rust via `ash`

Brian Merchant

0.1

## Contents



## Setup

Make sure:

1. you have the rust compiler installed: https://www.rust-lang.org (I am going to assume you've gone through the basics of Rust)

2. you have the Vulkan SDK installed: https://www.lunarg.com/vulkan-sdk/

## A note about C-ish for those unfamiliar with it

C-ish my catchall term for C/C++. For some parts of this guide, I am going to assume you know the following (and I think you would not need more than an hour to figure these things out):

- how arrays, `structs` and `enums` defined in C-ish, and what th keyword `typdef` does
- what a pointer is, and how it is related to C-ish arrays
- the syntax of a C-ish functions
- what a bit mask is, and how it's used in C-ish

If you can answer all these questions, you're good to go:

- what is `void`?
- why is a C-ish pointer evil compared to a Rust reference?
- what is `NULL`?
- why will you sometimes see `int*` written as the type for an array of `int`s?
- what is a function prototype?
- where is a function's return type and the type of its arguments, indicated in its prototype?
- do you know what a bit mask is in a broad sense (no need for details)?

I think these are good things to know, since you're dealing with a Rust crate that is a wrapper around a C-ish interface. A lot of the documentation on Vulkan is in C-ish, and this documentation is good to read because:

- it tells you what C-ish analogues you need to look for in the `ash` source code/documentation, when exploring `ash`
- lets you read the basics of Vulkan tutorials that operate in C-ish, in case you need to read about some stuff outside the scope of this tutorial

# Introduction

Vulkan provides an interface between your application and a Vulkan-compatible physical device installed in your system, which the application wants to use for highly-parallel computation typical of graphics tasks, but also useful elsewhere. Vulkan can interface with many different types of physical devices:

- the CPU of your system
- a software abstraction running on top of the CPU
- special purpose hardware for highly-parallel computing, e.g. a GPU;
- many CPUs configured to share data with each other
- et cetera

Note that each one of these physical devices also have some associated memory, where instructions, input data, and output data may be stored. Vulkan allows your application to interface with many varieties of physical devices, regardless of the details of each physical device's configuration, as long as the physical device provides a Vulkan-compatible "driver", which would be a piece of software that provides a Vulkan interface to the underlying hardware. Thus, as an application programmer, you only need to worry about interfacing your application with Vulkan, while the drivers handle interfacing Vulkan with the hardware.

Note that the driver doesn't do anything beyond helping Vulkan pass instructions to the hardware, so verifying whether the instructions you send make sense, or whether the resulting data from a computation is not garbage, is your task. However, during development of your application, you can specify additional software *layers* between your application's "loader" (i.e. its Vulkan communication object) and the physical device's driver, which can help to debug and verify the correctness of the instructions you are passing through the loader. You can choose to turn on and off these loaders as you wish.

Let's now take a closer look at the loader, which is more specifically called an "instance". When you first create the instance, it searches your system for physical devices with Vulkan-compatible drivers, and enumerates these for you, so that you can choose which physical device(s) you would like to work with. Once you have chosen specific physical devices, you can describe more specifically which features of each physical device you would like to use (e.g. 64-bit floats, or 32-bit floats?). Each physical device can then be abstracted into "logical devices", which is a software represent subsets of a physical device's resources. For example, let us say your application needs to do some graphics calculations, and some simulation related calculations: so to organize your physical device(s)' computing resources, you could set one logical device to deal with the graphics, and the other logical device could be set to deal with calculations. Thus, specification of logical devices is an abstraction that helps you organize computing resources. Each logical device allows you to create "queues", which are selected from "queue families". Queues organize the communication of data from your application (e.g. computation instructions, or some data to be manipulated) to the physical device's driver, and they also organize communication of data from your physical device to your application (e.g. results of computations, or status of the physical device's computing efforts). Queue families represent specialization of queues: queues to handle data (memory) transfers, queues to handle instruction transfer, etc. Note that sometimes, only certain queue families may be available on certain physical devices (and their availability may indicate what role the physical device is specialized for), but common physical devices tend to support all commonly used queue families. Finally, we may also take advantage of functionality provided by Vulkan "extensions", which are API extensions that provide common functionality for some often-occurring Vulkan use-cases (e.g. graphics display).

We are interested in using Rust to write our application, but the Vulkan API is written in C, so we use `ash`, which is a lightweight (thus unsafe) Rust wrapper around Vulkan. This document will teach you how to use Vulkan through `ash`, by helping you build a series of small programs ("samples") which iterate upon each other towards a final product (displaying a cube).

# Instances, Logical Devices and Queues

A Vulkan *instance* is a software object (with C-ish type `vkInstance`) which your application uses to interact with Vulkan-compatible physical devices which are represented as members of the instance. Commands from your application are passed through a Vulkan instance into a *queue* (each associated with a logical device abstracting a subset of the physical device's resources) which pipes to and from the physical device.

Note that since an instance is a software abstraction, if necessary, one could have several instances interacting with one physical device, if it provides value to how your application abstracts your physical devices. For now though, we'll focus on using one instance.

## `init-instance.rs`

If we were using C-ish, an instance is created by calling the function `vkCreateInstance`, which has the prototype:

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

Let's take this apart:

- return type `vkResult`: a C-ish `enum`, which contains a bunch of constants indicating the result of different Vulkan commands: in this case, success (i.e. successful creation of an instance), or some sort of a failure;
- argument `pCreateInfo`: a pointer to a `vkInstanceCreateInfo struct`;
- argument `pAllocator`: a pointer to a `vkAllocationCallbacks struct` whose members contain various functions you might have written to help the physical device organize its memory usage—we will tend to go simple, and not provide anything, in which case the physical device will use its default memory management routines;
- argument `pInstance` an "opaque pointer" to an instance (an opaque pointer is a C-ish concept which provides a pointer to a data structure, but the pointer cannot be used to query details of the data structure, nor can it be de-referenced)

Let's now have a closer look at `vkInstanceCreateInfo struct` definition, which has some members typical to many other Vulkan "info `struct`"s:

```
typedef struct VkInstanceCreateInfo {
        VkStructureType             sType;
        const void*                 pNext;
        VkInstanceCreateFlags       flags;
        const VkApplicationInfo*    pApplicationInfo;
        uint32_t                    enabledLayerCount;
```

```c
        const char* const*          ppEnabledLayerNames;
        uint32_t                    enabledExtensionCount;
        const char* const*          ppEnabledExtensionNames;
} VkInstanceCreateInfo;
```

A brief description of the members:

- `sType`: indicates the type of the info `struct`, in this case `VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO`—this is useful because in C, you might sometimes get a typeless (`void*`) pointer to a particular `struct`, and to determine what kind of `struct` it is, you could query the `sType` field
- `pNext`: another typical info `struct` member, usually set to `NULL`, and can be used to pass additional `structs` (whose type will be defined by the `sType` field)—this is only used by API extensions, and not the core Vulkan API!
- `flags`: currently no flags defined, set to `0`
- `pApplicationInfo`: pointer to a `VkApplicationInfo` `struct` which we will study next
- `ppEnabledLayerNames`: in this tutorial, we will not be using layers, so we can set this to `NULL`
- `enabledLayerCount`: length of the `ppEnabledLayerNames` list, should be zero if `ppEnabledLayerNames == NULL`
- `ppEnabledExtensionNames`: at this point in the tutorial, we are not using extensions, so we'll be setting this to `NULL`
- `enabledExtensionCount`: length of the `ppEnabledExtensionNames` list

The `vkInstanceCreateInfo` `struct` contains a `VkApplicationInfo` `struct` which has the following definition:

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

This `struct` is meant to provide some information regarding your application; a brief overview of its members:

- `sType`, `pNext`: common features of many Vulkan info `structs`, see the `vkInstanceCreateInfo` `struct`'s member overview, as these are the same;

- `pApplicationName`, `applicationVersion`, `pEngineName`, `engineVersion`: fields that you may fill out if you desire to annotate general report/debugging data, or if you want a tip a driver with specific behaviour for your application implemented;

- `apiVersion`: this field communicates the major, minor, and patch levels of the Vulkan API used by the application; we'll be using `VK\_API\_VERSION\_1\_0` (major is `1`, minor is `0`).

With these prototypes in mind, let's see how we can use Vulkan through `ash`. Open up `/init-instance/src/main.rs`, and:

- note comments explaining what **`use`** statements, if you need some clarification

- note the **`unsafe`** block

- note how for fields like `pApplicationName` we create a `std::ffi::CString` and then pass it "raw" (i.e. as a pointer) by calling `as_ptr`

- note how `vk::StructureType` is used to fill out `s_type` fields

- can you explain how the `flags` field is filled out? (hint: look up the definition of the `vk::InstanceCreateInfo struct` in `ash`'s documentation and then answer: what type is `flags`? How is the `std::Default` trait implemented for it?)

- to actually use Vulkan, we first initialize an `ash::Entry struct` which contains the supporting infrastructure to allow Rust to interface with Vulkan's C implementation

- the `ash::Entry struct` also has some **`impl`**s which put syntactic sugar around calling functions like `vkCreateInstance`, in particular the function `create_instance`—can you find where that function is implemented? (hint: open up `ash`'s documentation, and 1) figure out where the `EntryV1_0` trait is defined, 2) figure out where `Entry<V>` is defined, and 3) how is the `EntryV1_0` implemented for `Entry<V1_0>`?)

- note how we have to manually destroy the instance once we're done

To see everything in action:

```
cargo run --bin init-instance
```

## `enumerate-devices.rs`

Now that we know how to initialize an instance, let us learn how to use it to enumerate the physical devices on our system. In general, obtaining a list of stuff is a fairly common operation in Vulkan, so there's a generalized pattern behind what we want to do on the C side of things. Say you have a function `getListData` to get a list of some stuff from some instance of a `struct aStruct`:

- `getListData` will have prototype:

```
vkResult get_list_data(
        vkSomeStruct        aStruct,
        uint32_t*               pCount,
        vkStuff*              pStuff,
);
```

- get a pointer to some memory set aside for an unsigned 32 bit integer, and call it `pCount`
- call `getListData(aStruct, pCount, NULL)`—the `NULL` pointer for `pStuff` indicates that we don't yet know how many things will be in the list, so we want that information to be put in the memory pointed to by `pCount`, and if the `vkResult` we get back after the call indicates success, `pCount` will point to an integer representing the number of `vkStuff` we need to set aside memory for
- set aside appropriate memory for the list of `vkStuff` (now that we know the count), and get a pointer to that list called `pStuff`
- call `getListData(aStruct, pCount, pStuff)`—this time, because `pStuff` is not `NULL`, and the `getListData` will fill out the list pointed to by `pStuff`, instead of writing information about the count

Thus, based on this model, the function prototype for `vkEnumeratePhysicalDevices` should look familiar:

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

Assume we have a `vkInstance` called `instance` at hand and a pointer to memory where we can store an unsigned integer `pPhysicalDeviceCount`. Then, we would:

1. call `vkEnumeratePhysicalDevices(instance, pPhysicalDeviceCount, NULL)`—if `vkResult` indicates success, then `pPhysicalDeviceCount` now points to an integer denoting the number of physical devices available
2. allocate enough memory for a list of `VkPhysicalDevice`s containing `*pPhysicalDeviceCount` items (in C-ish the list would be of type `VkPhysicalDevice*`), and call it `pPhysicalDevices` (the preceding `p` indicates "pointer" in C-ish naming conventions)
3. call `vkEnumeratePhysicalDevices(instance, pPhysicalDeviceCount, pPhysicalDevices`—if `vkResult` indicates success, then our list `pPhysicalDevices` should be properly filled out with the `vkPhysicalDevice`s.

How would we do this in Rust? The `ash::Instance struct` has an `impl` function `ash::Instance::enumerate_physical_devices`, which calls `vkEnumeratePhysicalDevices` for us and returns a vector of `vk::PhysicalDevice`s. Have a look at how it works: gist link for annotated `enumerate_physical_devices`. To find the source for yourself, go to `ash`'s documentation for `ash::Instance`, and click the appropriate `[src]` links. You can see that under the hood, `ash` follows exactly the pattern we noted above.

Have a look at `enumerate-devices.rs`, where we query for the list of the physical devices on our system, and print out how many we found. So, to figure out how many physical devices are on your system run `enumerate-devices.rs`:

```
cargo run --bin enumerate-devices
```

# init-device.rs

## 7.1 init-device-0.rs

So far, we know how to create an instance, which allows our application to communicate with underlying physical devices that provide a Vulkan-compatible driver. In particular, we know how to use the instance to query if there are any such physical devices.

The rest of this tutorial is assuming you found at least one physical device on your system. If you haven't found one, do you have Vulkan-compatible drivers installed for your CPU/GPU, and is your CPU/GPU new enough to support Vulkan functionality? You'll have to do some trouble shooting here.

Assuming that you have found at least one such physical device, we will now learn how to create the logical device abstraction over this physical device. Recall that the logical device allows us to eventually create queues which pass data between your application and the physical device.

For the sake of simplicity, we'll just choose the first (perhaps only) device we've found. Let's ask the physical device what sort of queue families it supports. We do so by by calling the C function `vkGetPhysicalDeviceQueueFamilyProperties` which has prototype:

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

Note that this function has the same organization as a function which can either give information about the number of some objects of interest, or provide a list of the same objects of interest. In the last section, we discussed how such a function would be used, and we simply need to repeat that process in order to get a list of `VkQueueFamilyProperties`.

Again, similar to `ash::Instance::enumerate_physical_devices` discussed in the last section, `ash` makes our life easy by giving us an `impl` function on `ash::Instance` which does performs the details of getting a list of `vk_sys::QueueFamilyProperties`: `ash::Instance::get_physical_device_queue_family_properties`. If you examine its source (recall how we found the source for `ash::Instance::enumerate_physical_devices`) you'll see the similarities between the functions. Upon a successful call to `get_physical_device_queue_family_properties`, we get a `Vec<vk_sys::QueueFamilyProperties>`.

What's inside C-ish `vkQueueFamilyProperties`?

```c
typedef struct VkQueueFamilyProperties {
        VkQueueFlags    queueFlags;
        uint32_t        queueCount;
        uint32_t        timestampValidBits;
        VkExtent3D      minImageTransferGranularity;
} VkQueueFamilyProperties;

typedef enum VkQueueFlagBits {
        VK_QUEUE_GRAPHICS_BIT = 0x00000001,
        VK_QUEUE_COMPUTE_BIT = 0x00000002,
        VK_QUEUE_TRANSFER_BIT = 0x00000004,
        VK_QUEUE_SPARSE_BINDING_BIT = 0x00000008,
} VkQueueFlagBits;
```

We'll only look at some of the members in detail for now:

- `queueCount`: unsigned integer specifying how many queues this device has
- `vkQueueFlags` is a bit mask of one or more `VkQueueFlagBits` specifying the capabilities of the queues in this family (e.g. graphics queues specialize in handling graphics-related operations, while transfer queues specialize in transferring data between application and device).

The bit mask stuff is also important in Rust-land. `ash` has a special type for the bits, `ash::types::QueueFlags`, which allow bitwise operations on them too. So, here's what's going on with the bit mask stuff:

- converting from hexadecimal to binary:

| | |
|---|---|
| `0x1` $\to 000\underline{1}$ | Graphics |
| `0x2` $\to 00\underline{1}0$ | Compute |
| `0x3` $\to 0\underline{1}00$ | Transfer |
| `0x4` $\to \underline{1}000$ | Sparse |

- so if I gave you a flag `0xF`, in binary that is 1111, and this would mean that that queue family supports Graphics, Compute, Transfer, and Sparse operations
- but if I gave you `0x5`, in binary that is 0101, this would mean that the queue family only supports Graphics and Transfer operations

To test whether a particular bit is set (i.e. is 1), we can use the `subset impl` for `ash::types::QueueFlags`. See how `get_queue_family_supported_ops` function tests for this in `init-device-0.rs`. Also, note that before the program panics intentionally, care is taken to ensure that any instances we create are destroyed with the help of the `unsafe` function `destroy_instance_and_panic`.

When you run `init-device-0.rs`, you should see a print out of all the queue families supported by the first physical device in the list of physical devices available to you.

## 7.2 `init-device-1.rs`

In the last section, we described `init-device-0.rs`, which allowed us to print out a list of queue families available for one physical device on our system. In `init-device-1`, we will iterate upon `init-device-0`, and choose a physical device which has a queue family with graphics capability (recall that the ultimate aim of the samples progression is to display a cube, which is undoubtedly a graphics operation).

Once we choose the appropriate physical device, and the appropriate queue family from the device, we will set about creating a *logical device*, which will have a queue from the queue family of interest. Here's a broad outline of the steps we'd follow if we were using the C-ish API:

1. fill out a `VkDeviceQueueCreateInfo struct` based on queue family selected for each queue we want to create (i.e. we would form a list of such `struct`s)

2. fill out a `VkDeviceCreateInfo struct`, which amongst other members, has `pQueueCreateInfos`: a pointer to a list of `VkDeviceQueueCreateInfo struct`s (one for each queue we want to create)

3. call `vkCreateDevice`, with the `VkDeviceCreateInfo struct` we filled out as one of the `struct`s, to create a `vkDevice` upon success

4. perform any tasks we need to with the resulting `vkDevice`

5. destroy the device

From now on in this tutorial, whenever you read "device" without qualification, assume it means "logical device". I will always refer to a physical device as "physical device".

Let's have a look at `VkDeviceQueueCreateInfo`:

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

Relevant members of this `struct` are:

- `flags`: for future (as in, future versions of Vulkan) use, set to 0 in the present

- `queueFamilyIndex`: the index of the queue's queue family properties in the list of `vkQueueFamilyProperties` for the physical device we will create a logical device for;

- `queueCount`: the number of queues we'd like to create (shouldn't be greater than the number of queues supported by the physical device for this queue's queue family properties!)

- `pQueuePriorities`: a list of `queueCount` normalized floating point values (i.e. `floats` between 0.0 and 1.0), denoting the relative priority of each queue we're creating. Higher values indicate a higher priority (0.0 is lowest, 1.0 is highest), and within the physical device, queues with higher priority will have a higher chance of being allotted more processing time than queues with lower priority; more on this later

Let's have a look at the `VkDeviceCreateInfo struct`:

```
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;
```

Relevant members are:

- `queueCreateInfoCount`: how many queue groups are going to be associated with this logical device (each group will have its own `vkDeviceQueueCreateInfo struct`)
- `pQueueCreateInfos`: list of `vkDeviceQueueCreateInfo structs` (count given by `queueCreateInfoCount`)
- `enabledLayerCount` and `ppEnabledLayerNames`: deprecated and ignored
- `enabledExtensionCount` and `ppEnabledExtensionNames`: to be discussed later, for now we'll put in `0` and `NULL` respectively
- `pEnabledFeatures`: to be discussed later, `NULL` for now

Already we are getting hints of more advanced features available to us, such as Vulkan API extensions which can add functionality to the API in order to handle specialized tasks, or physical device features (e.g. 64 bit capability). We won't be worrying about these advanced features for now.

The analogues of `VkDeviceQueueCreateInfo` and `VkDeviceCreateInfo` in `ash` are `vk::DeviceQueueCreateInfo` (docs) and `vk::DeviceCreateInfo` (docs) respectively. In `init-device-1.rs`, we select a physical device which has a queue family with graphics capability, fill out `vk::DeviceQueueCreateInfo` and `vk::DeviceCreateInfo` respectively, and then call `ash::Instance::create_device` to create a device. Upon exit, we clean up by destroying the created device, and the instance.

Do note one important thing: we are only making one queue in the example, but if we were making more than one queue, we would provide either a Rust array or vector of `vk::DeviceQueueCreateInfo` with `.as_ptr`. This provides a C-ish style list of objects, which can be accessed by offsetting the pointer.

Make sure to study the code in `init-device-1.rs` and then run it:

```
cargo run --bin init-device-1
```

## `init-command-buffer.rs`

Unlike other APIs which send commands (memory transfer commands, draw commands, etc.) directly to the physical device's drivers, Vulkan records commands in a command buffer, which are then submitted using queues to the physical device's driver, which in turn handles the organization of the execution of instructions in the command buffer—in other words, Vulkan splits the recording of commands from the submission of commands. The benefit of such a system is that it allows drivers to better manage resources, since it has some idea of the work it is going to have to do.

Creating and destroying individual command buffers can be expensive, Vulkan utilizes the abstraction of "command buffer pools", which manage the creation and destruction of command buffers more efficiently through the use of specialized pool allocators. Furthermore, pools can manage sending large groups of (small) command buffers with increased efficiency, as there are inefficiencies in sending small command buffers. It is important to note that a command buffer pool is associated with one particular queue family on some physical device, since the driver allocates pools based on details specified by queue family.

To create a command buffer pool, we need to fill out a `vkCommandPoolCreateInfo struct`, and then submit it to a `vkCreateCommandPool` function. Let's have a look at `vkCommandPoolCreateInfo`:

```
typedef struct VkCommandPoolCreateInfo {
        VkStructureType             sType;
        const void*                 pNext;
        VkCommandPoolCreateFlags    flags;
        uint32_t                    queueFamilyIndex;
        } VkCommandPoolCreateInfo;
```

- `sType`: set to `VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO`
- `pNext`: typical info `struct` member
- `flags`: a bitmask of `VkCommandPoolCreateFlagBits` indicating pool usage and behaviour attributes. Possible bits are:
  - `VK_COMMAND_POOL_CREATE_TRANSIENT_BIT` (0x1): command buffers allocated from the pool will be short-lived
  - `VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT` (0x2): when a command buffer is first allocated is in the initial state, and command pools with the reset bit set allow buffers from the pool to be reset to their initial state

- `queueFamilyIndex`: index of the queue family that the command pool will be associated with (queue families are associated with a particular device, and the device in question is specified when calling `vkCreateCommandPool`)

Let's have a look at the function prototype for `vkCreateCommandPool`:

```
VkResult vkCreateCommandPool(
        VkDevice                                    device,
        const VkCommandPoolCreateInfo*              pCreateInfo,
        const VkAllocationCallbacks*                pAllocator,
        VkCommandPool*                              pCommandPool)
```

- `device`: logical device that will create the command pool
- `pCreateInfo`: a pointer to a filled out `vkCreateCommandPool` `struct`
- `pAllocator`: standard input described in earlier sections
- `pCommandPool`: a pointer to a `vkCommandPool`, which `vkCreateCommandPool` will associate with the newly created pool

In a previous section, we identified a graphics capable queue family, and will associate our command pool with it.

Once we have a command pool we store it in a `VkCommandBufferAllocateInfo` `struct`, which we can use along with a `vkAllocateCommandBuffers` call to create (allocate) command buffers. Let's have a look at `VkCommandBufferAllocateInfo`:

```
typedef struct VkCommandBufferAllocateInfo {
        VkStructureType         sType;
        const void*             pNext;
        VkCommandPool           commandPool;
        VkCommandBufferLevel    level;
        uint32_t                commandBufferCount;
        } VkCommandBufferAllocateInfo;
```

- `sType`: set to `VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO`
- `pNext`: standard info `struct` member
- `commandPool`: command pool that we created
- `level`: the `enum` `VkCommandBufferLevel` has values `VK_COMMAND_BUFFER_LEVEL_PRIMARY` and `VK_COMMAND_BUFFER_LEVEL_SECONDARY`, which specify whether this command buffer is primary or secondary. A primary command buffer can be submitted to a queue but cannot be called by another command buffer, while a secondary command buffer cannot be submitted to a queue, but can be called by another command buffer.
- `commandBufferCount`: how many buffers to make with these settings

Let's have a look at the function pointer for `vkAllocateCommandBuffers`:

```
VkResult vkAllocateCommandBuffers(
        VkDevice                                    device,
        const VkCommandBufferAllocateInfo*          pAllocateInfo,
        VkCommandBuffer*                            pCommandBuffers);
```

- `vkDevice`: the device with which the command pool is associated
- `pAllocateInfo`: pointer to a filled out `VkCommandBufferAllocateInfo` struct
- `VkCommandBuffer`: a pointer to an array of `VkCommandBuffers` where the newly created command buffers will be stored (so, the array should have length at least `commandBufferCount` set in the `VkCommandBufferAllocateInfo struct`)

Note that once a command pool is created, it must also be destroyed.

Let's now turn our attention to Rust and `ash`, where we'd follow (similar to C-ish) the following steps to create a command pool, and then command buffers associated with that pool:

1. we store info used to create a command pool in a `vk::CommandPoolCreateInfo struct`
2. a logical device (with type `Device<V1_0>` in our tutorial so far) has an `impl create_command_pool` for creating a command pool
3. we store info used to allocate command buffers in a `vk::CommandBufferAllocateInfo struct`
4. a logical device (with type `Device<V1_0>` in our tutorial so far) has an `impl allocate_command_buffers` for allocating buffers
5. use command pool as necessary
6. destroy command pool with a logical device's (with type `Device<V1_0>` in our tutorial so far) `impl destroy_command_pool`

In `init-command-buffer.rs`, you can see how these steps are straightforwardly executed using `ash`. However, note that our clean ups are becoming more complicated and have a look at the new function `clean_up`.

In coming sections, we will record commands to command buffers, but note that recording commands in the buffer does not make the GPU do anything until the command buffer is submitted using a `vkQueueSubmit` call. We have much to do before we will make this call, which will be made in the last part of this tutorial series.