Using Vulkan with Rust via `ash`

Brian Merchant

0.1

Contents

Abstract

This document provides a Rust-based introduction to using Vulkan to create graphics. `ash`, a Rust wrapper around the C Vulkan API, is used to build a series of small programs ("samples") which iterate upon one another towards a final goal of displaying a 3D cube.

Setup

Necessary: install Vulkan SDK.

Optional: read history of Vulkan on Wikipedia.

"C-ish"

C-ish is this document's shorthand for C/C++. Since the Vulkan API is written in C, being faimilar with certain features of C-ish is useful. Additionally, many existing Vulkan tutorials use C++.

However, this document does not assume intimacy with C-ish: in fact, the author has only a passing familiarity. The following is a complete list of concepts that will be directly or indirectly referenced within the document:

- syntax and concept differences between C-ish pointers and Rust references
- C-ish arrays
- relationship between pointers and arrays in C-ish; why will one sometimes see `int*` given as the type for an array of `int`s?
- C-ish `struct`s and `enum`s
- the keyword `typdef`
- the keyword `void`
- the keyword `NULL`
- function definition syntax; what is a "function prototype"?
- what a bit mask is, and how it's used
-
- what does `\0` do in a string?

Introduction to Vulkan

Vulkan provides an interface between software applications and a Vulkan-compatible physical devices. Vulkan compatible physical devices can be:

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

A physical device is Vulkan compatible if someone has written a "driver" meant to organize communication between the physical device and Vulkan's internals. Thus, Vulkan standardizes communication between software and a wide variety of hardware.

This interface between application and hardware drivers is called the "loader". Thus, to begin using Vulkan, a programmer initializes the loader. Vulkan's specification calls for both the loader and the drivers it communicates with to be highly specialized to facilitate communication between software and silicon, and extraneous capabilities (such as debugging conveniences) are provided through optional (chosen by the programmer) "layers" which are placed between the loader-driver conduit.

Thus, to begin, a programmer creates an "instance" of the loader. Through this loader instance, they search system for physical devices with Vulkan-compatible drivers. They choose specific physical devices to work with, and set which optional features of the physical devices should be used (e.g. use 64-bit floats, or 32-bit floats?). For each physical device, the programmer defines one or more "logical devices": abstractions representing subsets of a physical device's resources. Each logical device is used to create "queues", which are selected from several types of "queue families". Not all physical devices may implement every type of queue family, as different queue families are specialized for transferring particular types/formats of information. Finally, the programmer enables Vulkan API "extensions" which provide convenient functions to handle certain tasks they might be interested in (e.g. graphics display).

> Vulkan exposes one or more devices, each of which exposes one or more queues which may process work asynchronously to one another. The set of queues supported by a device is partitioned into families. Each family supports one or more types of functionality and may contain multiple queues with similar characteristics. Queues within a single family are considered compatible with one another, and work produced for a family of queues can be executed on any queue within that family. This Specification defines four types of functionality that queues may support: graphics, compute, transfer, and sparse memory management.

0: `init-instance.rs`

A Vulkan *instance* has C-ish type `vkInstance`. One could create many `vkInstance`s, if it is helpful for their task, but we only need to create one. In 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);
```

- 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 us examine the `struct vkInstanceCreateInfo`, since we need to provide `vkCreateInstance` with one:

```
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;
```

- `sType`: a member common to all Vulkan "info `struct`"s, it 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 common info `struct` member, usually set to `NULL` unless API extensions (and only extensions) require one to pass additional `struct`s

- `flags`: for future Vulkan versions, 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 thus should be zero since we have `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

This `struct` has which has some members typical to many other Vulkan info `struct`s, as the reader will see. `vkInstanceCreateInfo struct` requires as a member an instance of the `VkApplicationInfo struct` meant to provide some information regarding the application initializing the loader:

```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;
```

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

- `pApplicationName`, `applicationVersion`, `pEngineName`, `engineVersion`: for general use, e.g. sometimes drivers may be designed to execute special behaviour for certain application—this can be used to let them know which application they are dealing with

- `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`).

In `/init-instance/src/main.rs` we initialize a loader. Points to note and questions to consider:

- comments explaining `use` statements

- `unsafe` block: `ash` an almost one-to-one interface to the Vulkan C API, thus many of Rust's safety features have to be disabled

- for fields like `pApplicationName` we create a `std::ffi::CString` and then pass it "raw" (i.e. as a pointer) by calling `as_ptr`, rather than using the standard Rust `String`

- `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?)

- that 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>`?)

- 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 `vkPhysicalDevices`.

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::PhysicalDevices`. 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

6.1 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:

```c
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`?

```
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` → 0001	Graphics
`0x2` → 0010	Compute
`0x3` → 0100	Transfer
`0x4` → 1000	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.

6.2 `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 `structs`)

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

3. call `vkCreateDevice`, with the `VkDeviceCreateInfo struct` we filled out as one of the `structs`, 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;
```

```c
        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`:

```c
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`

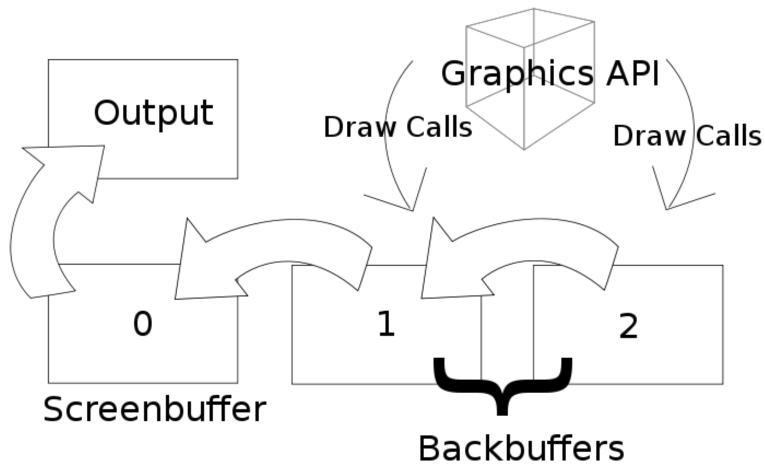

Figure 1: An abstract depiction of the swap chain. Source.

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.

`init-swap-chain.rs`

In this section, we'll learn how to initialize the "swap chain", sometimes also written as "swapchain". What is a swap chain? Well, Wikipedia says it best[1]:

> In computer graphics, a swap chain is a series of virtual framebuffers utilized by the graphics card and graphics API [in our case, Vulkan] for frame rate stabilization and several other functions. The swap chain usually exists in graphics memory, but it can exist in system memory as well. The non-utilization of a swap chain may result in stuttering rendering, but its existence and utilization are required by many graphics APIs...In every swap chain there are at least two buffers. The first framebuffer, the screenbuffer, is the buffer that is rendered to the output of the video card. The remaining buffers are known as backbuffers. Each time a new frame is displayed, the first backbuffer in the swap chain takes the place of the screenbuffer, this is called presentation or swapping...(see also Figure 1)

So, the output of the graphics device is buffered through the use of swap chains, in order to give the end-user the illusion that rendering is smooth, by making the frames appear one after the other at a constant, rather than variable rate.

The core Vulkan API is platform (e.g. Windows, GNU/Linux, MacOS) and physical device (see Introduction) agnostic. In order to expose capabilities provided by a particular platform or physical device, the core API has to be extended by special extensions which take into account details relevant to the particular platform or physical device. Many of these extensions are provided and maintained by the Khronos Group itself, and we will now augment the core API with an extension that will allow it to deal with physical devices' specific implementations of swap chains.

So, how does one enable the relevant extension? Recall the info `struct` we had to fill out in order to create a logical device:

```
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;
```

When we last looked at this `struct`, we ignored `enabledExtensionCount` and `ppEnabledExtensionNames`, but in this section, we'll fill these out. For creating and managing swap chains, we are interested in the extension `VK\_KHR\_swapchain`, so we'd pass a list containing that string, and the count of that list.

With `ash`, we'd do this as follows:

1. `use ash::extensions::Swapchain`: loads the module which contains swap chain extension related stuff. In particular, the `ash::extensions::{DebugReport, Surface, Swapchain}` `struct` has function pointer members (i.e. members which are basically functions), `impl`s that will be of great use to us

2. Create an array with the right string, by using the `name` static `impl` of `Swapchain` (track down the source for `Swapchain::name` as an exercise):

   ```
   let device_extension_names_pointers = [Swapchain::name().as_ptr()];
   ```

3. set the `enabled_extension_count` and `pp_enabled_extension_names` members of the `vk::DeviceCreateInfo` `struct` we fill out using `device_extension_names_pointers.len() as u32` and `device_extension_names_pointers` respectively

Thus, when we create `device` in `init-swap-chain.rs`, we'll have enabled the swap chain extension.

In C-ish our next step will be to set up the `VkSwapchainCreateInfoKHR` info `struct`:

```c
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;
```

- `sType` and `pNext` are standard info `struct` fields, while `flags` is for future use and set to 0
- `surface`: surface to which this swap chain will present to (more about surfaces coming up right after)
- `minImageCount`: how "deep" the swap chain should be (how many buffers it should have between input and output)
- `imageFormat`: format of the data representing the images (i.e. pixel encoding format) the swap chain will be handling, a value from the `VkFormat enum`
- `imageColorSpace`: specifies which colour space the presentation engine will interpret each pixel (recall that pixels store color information) `VkColorSpaceKHR`
- `imageExtent`: maximum size of the images the swap chain will handle
- `imageArrayLayers`: relevant for applications that will be presenting monitors that take advantage of the stereoscopic effect to provide the illusion of 3D images—we'll just set the value to 1, as we are making an application that will display on standard monitors
- `imageUsage`: a bitmask of VkImageUsageFlagBits specifying how the application will use the images

- `imageSharingMode`: a value from the VkSharingMode `enum`, specifying whether access to data in the swap chain is exclusive to queues from one family, or whether access is open to queues from multiple queue families:
  - `queueFamilyIndexCount` and `pQueueFamilyIndices`: if concurrent access between queue families is allowed, which families have permission?
- `preTransform`: a bitmask of VkSurfaceTransformFlagBitsKHR bits, which specifies how to transform (e.g. rotate, mirror, etc.) the images in the swap chain relative to the "natural" orientation of the presentation device (e.g. a monitor); to understand this better, consider this contrived example: hanging upside down while using a computer is a new health fad, but your monitor is locked to the table its on, so your application sets the `preTransform` bitmask to rotate the image by 180° so that the image appears right side up while you're upside down
- `compositeAlpha`: a bitmask of VkCompositeAlphaFlagBitsKHR bits, indicating how the images should be alpha composited together
- `presentMode`: value from `VkPresentModeKHR` `enum`; how will the presentation engine ask for data from the swap chain?
- `clipped`: should the underlying infrastructure care about pixels that are outside the image extents relevant for this swap chain?
- `oldSwapchain`: pointer to the old swap chain that this new swap chain is going to replace (`NULL` usually)
-

As you can see, there's a lot of infrastructure involved in taking image data from a physical device, and then displaying it to a user. The rest of this section is going to go over how to fill out this structure.

It is the "windowing system" (e.g. X Window System, MS Windows, Wayland) which is responsible for displaying the data stored in a swap chain, by providing us with a "surface" abstraction onto which data is drawn. Note that the windowing system is separate from the core Vulkan API, but there are Khronos Group maintained extensions to Vulkana (different ones for different windowing systems) which allow it to integrate with the windowing system (Window System Integration, or WSI). The relevant extension in this case will be specified in the info `struct` used to create the instance.

Getting the name of the relevant extension using `ash`needs a bit more care this time, because the extension to be used depends upon which platform our application is on. First, we should load the relevant extension modules: `use ash::extensions::{Surface, Win32Surface, XlibSurface}`, then we write functions that will conditionally compile based on which platform we are on:

```
#[cfg(all(unix, not(target_os = "android")))]
fn extension_names() -> Vec<*const i8> {
        vec![
```

```
                Surface::name().as_ptr(),
                XlibSurface::name().as_ptr(),
        ]
}

#[cfg(all(windows))]
fn extension_names() -> Vec<*const i8> {
        vec![
                Surface::name().as_ptr(),
                Win32Surface::name().as_ptr(),
        ]
}
```

Now we have surface related extensions loaded, but we still can't create a surface, until there is a window which can be associated with the surface. In C-ish, there are various windowing libraries, but in Rust, we'll use `extern crate winit`.

8.1 Creating a window

The following code in `init-swap-chain.rs` is relevant to creating a window:

```
let events_loop = winit::EventsLoop::new();

let window = winit::WindowBuilder::new()
.with_title("Ash - Example")
.with_dimensions(window_width, window_height)
.build(&events_loop)
.unwrap();
```

Let's understand what is going on here. First of all, we create an event loop. An event loop essentially handles messages sent between the user interface (the window), and our application. For instance, if the user presses a key, the event loop will notify our application of this, and allow it to choose what to do. It is worth taking a look at the documentation for `winit::EventsLoop`. Once we have the events loop set up, we build a window, using several pretty self-explanatory calls which provide the title, definition and so on. This tutorial will not spend much time discussing the internals of the window system, which depend heavily upon the particular platform the application is running on. Now that we have a `window`, we can create a surface. Again, these functions are going to be platform dependent: have a look at them (`create_surface`) within the source code for this example.

Find a queue that is capable of presentation.

The gist of what is happening in these functions is that

Thanks

The information in this tutorial was derived from multiple sources ("`<search engine> is your best friend!`"), but some stand prominent amongst them. Grateful thanks is owed to:

1. LunarG's Vulkan Samples Progression, whose progression structure I found comforting, and thus stole, along with some conceptual explanations.

2. Alexander Overvoorde's Vulkan Tutorial, whose rich coverage often provided some clues on things other's assumed a beginner would know—this tutorial does not even come close to covering the range of material Overvoorde's tutorial does.

3. https://github.com/MaikKlein, who began writing `ash`, and is its prime contributor. Also, Maik and msiglreith on `ash`'s Gitter were happy to answer my questions.

4. The many people behind open software projects such as Vulkan and Rust, who build vast infrastructure to provide ease that we take for granted.

References

[1] Wikipedia contributors. *Swap Chain — Wikipedia, The Free Encyclopedia.* [Online; accessed 13-January-2018]. 2015. URL: `https://en.wikipedia.org/w/index.php?title=Swap_Chain&oldid=689198445`.