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.

Vulkan commands that form the interface between application and hardware drivers are brought into action by a “loader”. Vulkan’s specification calls for its core commands and the drivers they communicate with to be highly specialized for software-silicon crosstalk, so 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).

From now on in this document, “device” without qualification shall refer to a “logical device” while a physical device will only be referred to in full as “physical device”.

> 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 `vkInstances`, 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 `vkCreat`⌋
`eInstance` 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 `structs`

- `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 `structs`, 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:

```
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
```

1: `enumerate-devices.rs`

Obtaining a list of information from the loader is a common operation, and in the course of this tutorial, we will note that there is a pattern to how the API handles these operations. The pattern is well demonstrated by the next task: querying the loader for a list of available Vulkan-compatible physical devices on the host system. We call the function `vkEnumeratePhysicalDevices`, which has prototype

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

The specification explains the function's arguments succinctly:

> If `pPhysicalDevices` is `NULL`, then the number of physical devices available is returned in `pPhysicalDeviceCount`. Otherwise, `pPhysicalDeviceCount` must point to a variable set by the user to the number of elements in the `pPhysicalDevices` array, and on return the variable is overwritten with the number of handles actually written to `pPhysicalDevices`. If `pPhysicalDeviceCount` is less than the number of physical devices available, at most `pPhysicalDeviceCount` structures will be written. If `pPhysicalDeviceCount` is smaller than the number of physical devices available, `VK_INCOMPLETE` will be returned instead of `VK_SUCCESS`, to indicate that not all the available physical devices were returned.

Assume we have a `vkInstance` called `instance` created, as explained in the last section, and a pointer to memory where we can store an unsigned integer `pPhysicalDeviceCount`. Then, for our purposes, we would:

1. call `vkEnumeratePhysicalDevices(instance, pPhysicalDeviceCount, NULL)` since we do not have a pre-existing array of devices, and are interested in finding out how many such devices exist; if `vkResult` indicates success (`VK_SUCCESS` is returned), then `pPhysicalDeviceCount` now points to an integer counting the number of physical devices available otherwise the specification states we get errors `VK_ERROR_OUT_OF_HOST_MEMORY`, `VK_ERROR_OUT_OF_DEVICE_MEMORY`, `VK_ERROR_INITIALIZATION_FAILED`

2. assuming success, create an empty array of `VkPhysicalDevices` containing enough space for `*pPhysicalDeviceCount` items `pPhysicalDevices` (`p` in the variable name stands for "pointer")

3. call `vkEnumeratePhysicalDevices(instance, pPhysicalDeviceCount, pPhysicalDevices`—if `vkResult` indicates success, then our list `pPhysicalDevices` should be filled out with the `vkPhysicalDevices`

In Rust-land, the `ash::Instance` `struct` has an `impl` function `ash::Instance::enumerate_physical_devices`, which calls `vkEnumeratePhysicalDevices` and returns a vector of `vk::PhysicalDevices`. The reader should look at the source code of this function: gist link for annotated `enumerate_physical_devices`. To find the source, go to `ash`'s documentation for `ash::Instance`, and click the appropriate `[src]` links. You can see that under the hood, `ash`performs the operations we outlined above.

`enumerate-devices.rs` contains the code to enumerate physical devices on a system and print out how many were found. 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`. Run the program with the command:

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

2: `init-device.rs`

The rest of this document assumes the at least one physical device exists on the host system.

6.1 `init-device-0.rs`

Creation of logical device abstractions over found physical devices will now be explored. Apart from organizing available physical resources, a logical device allows for the creation of queues which pass data between applications and physical devices.

For the sake of simplicity, let us select the first (perhaps only) physical device found. What queue families (types of queues) does it support? The list of available queue families can be obtained by calling `vkGetPhysicalDeviceQueueFamilyProperties`, which has prototype:

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

This function has the same organization as `vkEnumeratePhysicalDevices` seen in the previous section: after all, it too returns a list of data. Thus, similar to `ash::Instance::enumerate_physical_devices`, `ash`wraps the querying process in the `impl` function `ash::Instance::get_physical_device_queue_family_properties`. Examining this function's source will reveal its similarities to `ash::Instance::enumerate_physical_devices`. A successful call to `get_physical_device_queue_family_properties` provides a list: `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;
```

- `queueCount`: unsigned integer specifying how many queues this device has
- `vkQueueFlags`: a bit mask of one or more `VkQueueFlagBits` specifying the capabilities of queues in this family (some or all of Graphics, Compute, Transfer or Sparse).

`ash`has a special type for these flag bits, `ash::types::QueueFlags`, which also allow bitwise operations. Thus, we can convert the returned flag from hexadecimal to binary to determine the queue family's capabilities:

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

For example, the flag `0xF` is 1111 in binary, meaning that the queue family supports Graphics, Compute, Transfer, and Sparse operations, while flag `0x5`, in binary is 0101, meaning that only Graphics and Transfer operations are supported.

To test whether a particular bit is set (i.e. is 1), the `subset impl` for `ash::types::QueueFlags` can be used, as demonstrated by `get_queue_family_supported_ops` in `init-device-0.rs`. This program prints out queue families supported by the first physical device in the list of physical devices available on the host system.

6.2 init-device-1.rs

In `init-device-1`, `init-device-0` is iterated upon by by adding functionality to choose a physical device which has a queue family with graphics capability, since the tutorial ultimately aims to display a cube. One an appropriate physical device has been chosen, a *logical device* is created on it, which will be used to create a queue from the graphics family. A broad outline for executing this procedure using the C-ish API is:

1. fill out a `VkDeviceQueueCreateInfo struct` based on the queue family selected for each queue that will be created, and store each such `struct` in an array, perhaps called `pQueueCreateInfos`
2. fill out a `VkDeviceCreateInfo struct`, which amongst other members, requires `pQueueCreateInfos`

3. call `vkCreateDevice`, which takes as an argument `VkDeviceCreateInfo`, and will create `vkDevice` upon success

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

5. destroy the 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;
```

- `flags`: for use by future Vulkan versions, set to 0
- `queueFamilyIndex`: index of the queue's queue family properties in the array `vkQueueFamilyProperties`
- `queueCount`: the number of queues to be created (shouldn't be greater than the number of queues supported by the physical device, as specified in the queue family's properties!)
- `pQueuePriorities`: an array 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: 0.0 is lowest priority, 1.0 is highest; 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

The `struct VkDeviceCreateInfo` is defined like:

```
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;
```

- queueCreateInfoCount: how many queue groups are 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 in the document, for now 0 and NULL respectively
- pEnabledFeatures: to be discussed later in the document, for now NULL

Hints of more advanced features available, such as Vulkan API extensions or optional physical device features are beginning to appear, but we ignore these for the time being.

The analogues of VkDeviceQueueCreateInfo and VkDeviceCreateInfo in ashare vk::DeviceQueueCreateInfo (docs) and vk::DeviceCreateInfo (docs) respectively. init-device-1.rs selects a physical device graphics capable queue families, fills out vk::DeviceQueueCreateInfo and vk::DeviceCreateInfo, and then calls ash::Instance::create_device. Before exiting, it cleans up by destroying the created device, and the underlying loader instance. This example only creates one queue, but if more were to be created, we would provide a Rust array or vector of vk::DeviceQueueCreateInfo in raw pointer form through .as_ptr. This provides a C-ish style list of objects, which can be accessed through offsetting of the pointer.

Study the code in init-device-1.rs and then run it:

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

init-command-buffer.rs

Using Vulkan, application records commands in a command buffer, which are then transmitted to underlying drivers Since instructions are given in batches from a buffer, drivers are able to further optimize execution, as they have some fore-knowledge of future instructions.

However, creating and destroying individual command buffers manually can be expensive, so Vulkan utilizes the abstraction of “command buffer pools”, which manage the creation and destruction of command buffers using “pool allocators”. In addition, pools manage sending large groups of (small) command buffers with increased efficiency. Every command buffer pool is associated with a particular queue family on the physical device, since drivers allocate pools based on the type of queue family.

To create a command buffer pool, a vkCommandPoolCreateInfo struct is submitted to the vkCreateCommandPool function:

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

```
        uint32_t                    queueFamilyIndex;
        } VkCommandPoolCreateInfo;

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

- `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
- `device`: logical device which will manage creating 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

Once we have a command pool, we store it in a `VkCommandBufferAllocateInfo struct`, which is submitted to a call of `vkAllocateCommandBuffers`:

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

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

- `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
- `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 `VkCommandBuffer`s where the newly created command buffers will be stored (so, the array should have length at least `commandBufferCount` set in the `VkCommandBufferAllocateInfo struct`)

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.

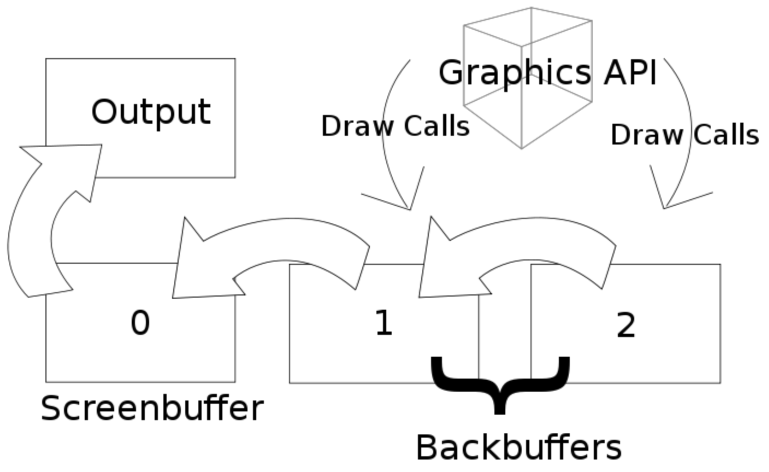

Figure 1: An abstract depiction of the swap chain. Source.

`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`:

```
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.