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 `structs` and `enums`
- 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?

Motivation

This tutorial assumes that the reader is somewhat interested in displaying computationally generated images on a physical screen: a task that is evidently easy to state. Attempting to execute this task however, will bear testament to the vast number of interconnected systems (developed over decades) which must be marshalled to create applications using electronic computing devices.

To display a graphical image on an electronic device, one must:

1. have some system (mathematics, "geometry") for describing images without ambiguity that requires human interpolation to decipher (i.e. without words)
2. have some system ("raster graphics") for digitally representing image data

3. have a computing device capable of *efficiently* converting mathematical descriptions of images into digital representations; these tasks are very computationally intensive, since each point (pixel) of raster image must be specified, and the number of pixels in any interesting picture tend to be very large—this computational difficulty spawned a whole family of computing devices geared specifically towards performing many small calculations in parallel ("graphics cards")

4. have an electronic device "CRT screen, LCD screen" capable of understanding/presenting digital image data: i.e. capable of converting image data into a physical format our eyes can interpret

The hardware (physical) software (digital, programmatic) systems required are less standardized than the mathematical systems involved. There is more than one vendor of Graphical Processing Units (GPUs), Central Processing Units (CPUs), screens, operating systems for managing these devices etc. and they are often engaged in economic competition, so standardization has been a slow, but steady process. Vulkan is the result of this process of standardization: it provides specification documents and software-side infrastructure to facilitate software-hardware communication by presenting a standardized interface joining software developers from one side with hardware developers on the other.

People began to notice that there is no reason why one should *only* perform graphics related computations in parallel on a graphics card, as there are many applications that can take advantage of the highly parallel nature of such physical devices. In fact, as Moore's prediction of exponentially increase processing speeds of electronic processors begins to tangle with engineering/physical limits, people are increasingly putting what were traditionally single processors into groups that work together.

The marketing for Vulkan API is in many ways, is a product of this trend as it portrays Vulkan as not only a graphics API, but also a general "compute" API. While it is true that compared to previous APIs such as OpenGL (highly specialized for graphics), Vulkan is designed with a far more abstract view of the computations it facilitates, the OpenCL API (developed by the same organization behind Vulkan) is a better choice for handling general (usually scientific) highly-parallelized and/or heterogeneous system computations. Still, there is something to be said for Vulkan's abstracted organization, as it makes possible future convergence of OpenCL with Vulkan.

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 somewhere, sometime, effort was put into writing a "driver" meant to organize transfer of data between hardware and software as per the Vulkan specification. 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 specific tasks (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 `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 `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 `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:

```
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 `vkEnumerat` ⌋ `ePhysicalDevices`, 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. 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::type` `s::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.

7.1 `init-device-and-queue.rs`

`init-device.rs` will now be 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 us 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 `ash`are `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-and-queue.rs` and then run it:

```
cargo run --bin init-device-and-queue
```

3: `init-command-buffer.rs`

In the Vulkan model of execution, an application records commands into a buffer, which is passed to physical device drivers Since instructions are received in batches, drivers are able to use knowledge of upcoming instructions to optimize execution of the entire batch.

Since aive implementations for creating and destroying individual command buffers can be very inefficient, Vulkan provides "command buffer pools", which manage the creation and destruction of command buffers using "pool allocators". Another source of inefficiency can be the processing of many command buffers, each with a small number of instructions, so pools also provide solutions here. Since pools are allocated based on the type of queue family their commands are intended for, every command buffer pool is associated with a single queue family available to the physical device.

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 a command pool is created, it is stored 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 was 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, while a secondary command buffer cannot be submitted to a queue, but can be executed by a primary command buffer; primary command buffers cannot be executed by secondary ones
- `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`)

Using Rust, assuming a logical device has been created:

1. use logical device's `impl create_command_pool` for creating a command pool
2. store info used to create a command pool in a `vk::CommandPoolCreateInfo struct`
3. store info used to allocate command buffers in a `vk::CommandBufferAllocateInfo struct`
4. use logical device's `impl allocate_command_buffers` for allocating buffers
5. use command pool as necessary
6. destroy command pool with logical device's `impl destroy_command_pool`

These steps are implemented in `init-command-buffer.rs`. Note that our clean ups are becoming more complicated, necessitating a new function `clean_up`, which can handle clean up of instances, devices, and pools.

In the last sample of this tutorial series, we will record commands to command buffers, and submit them to a physical device using Vulkan-provided `vkQueueSubmit`'s Rust analogue. Between now and then, necessary preparatory steps are undertaken.

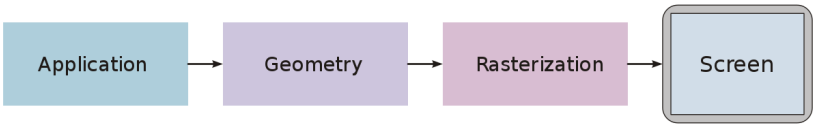

Figure 1: A diagram of a graphics pipeline showing only its most fundamental components. By PaterMcFly / Vierge Marie.

The Graphics Pipeline

In the coming sections, we will familiarize ourselves with Vulkan extensions that are gearted towards real time computer graphics applications. Since the Vulkan specification often makes references to concepts related to this process, having an understanding of how it works will be helpful.

The creation ("rendering") of 3D images using a computer can be broadly separated into categories:

1. ahead-of-time rendering: these rendering processes are well suited towards creating 3D images which require time-intensive computations (e.g. for lighting, see: Wikipedia article on Ray Tracing) to render highly realistic scenes, detailed animations, or other advanced effects; used for movies, game cut scenes, 3D art, etc.

2. on-demand rendering: these rendering processes are well suited towards creating 3D images for interactive applications, such as games or user interfaces, which require something like at least 25 frames per second to produce the illusion of smooth animation; clever techniques are used to minimize rendering cost (for example, reflections on glossy surfaces, but instead the scene from the perspective of the surface would be painted on as an approximation), at the cost of losing realistic detail, or other advanced features

The graphics pipeline is a conceptual organization of computing resources to handle the second case of on-demand rendering. APIs like Vulkan and its extensions are designed along the concepts of the graphics pipeline, which represent a series of computations (perhaps with loops?) data is passed through, with the end product being an image which can be displayed to the user. Vulkan allows the user to build and order the operations within the pipeline as required. The ordering of operations within a pipeline is referred to as `synchronization`. We now look at an example of a typical Vulkan pipeline.

The application provides computations that are run external to the pipeline. It selects which scenes need to be rendered, takes into account user input, collisions between physical object models, networking, and so on; in other words, it runs all the computations needed to provide the program of interest as a whole, and delegates graphics rendering to the graphics pipeline—when using Vulkan, the application does so by issuing drawing commands. This quote from the Vulkan specification shows how objects we have seen such as queues and command buffers come together to deliver drawing commands:

> Drawing commands (commands with Draw in the name) provoke work in a graphics pipeline. Drawing commands are recorded into a command buffer and when

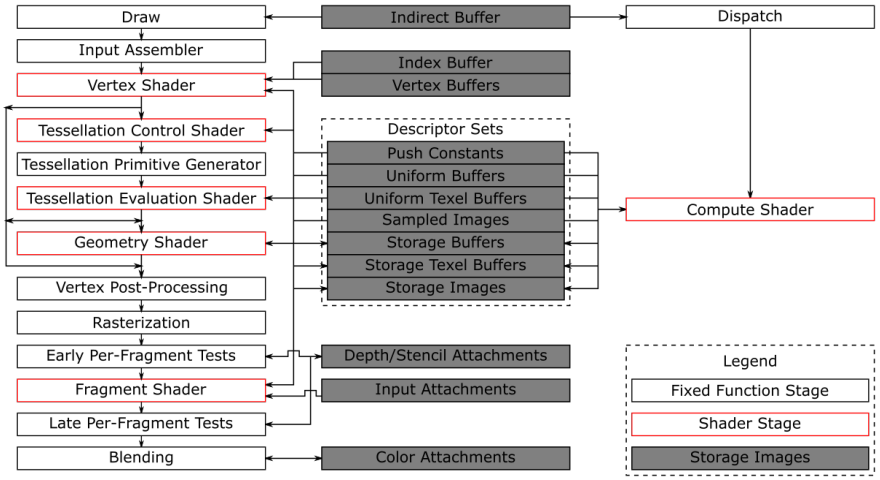

Figure 2: A Vulkan graphics pipeline, showing all the different components that could be utilized. From the Vulkan specification.

> executed by a queue, will produce work which executes according to the bound graphics pipeline. A graphics pipeline must be bound to a command buffer before any drawing commands are recorded in that command buffer.

After the application has issued graphics instructions using draw commands, these commands are transferred to the "Input Assembler", which puts together triangles, lines and points (geometric primitives) based on the requested commands.

Once these primitives have been generated, they are passed onto the Vertex Shader, which every Vulkan graphics pipeline must contain. The "Vertex Shader" ("shader" is simply jargon for a program for executing highly parallelizable tasks: for example, executing some instruction on a large number of vertices as input) computes the positions of the points and vertices assembled by the Input Assembler; one can think of the Input Assembler as defining the relationships between the various geometric primitives, based on drawing commands, but these primitives are not yet placed concretely in space, as this will depend upon the viewing angle, the zoom level, and so on. Having generated vertex data using the Vertex Shader, one can choose to pass this data through optional "Tesselation" and "Geometry" Shaders, which can do things like create more advanced shapes programmatically from primitives.

The vertex data is then passed through the Vertex Post-Processing step, where vertices which will not fit into the viewing area (see related Wikipedia article) are "clipped": that is, they will be essentially removed, since they will not be viewed within the scene. This way, computationally intensive operations do not need to be run on objects which will not be viewed in the first place. The clipped vertex data is then passed to the "Rasterization" stage.

Rasterization refers to the process of converting a mathematical data representing an image (as we have at this stage of the pipeline, where our scene is made of primitives defined by their coordinates) into a 2D pixel image representation. Pixels are stored in memory using "framebuffers" ("frame" for an image frame, and "buffer" for storage memory). The data Rasterization produces for a given pixel (its (x, y) coordinates, address within the framebuffer, colour, depth, etc.) is called a `fragment` (see related Wikipedia article). The

next steps in the pipeline perform calculations which may modify the fragment. For instance, in the Early Per-Fragment tests stage, a technique called z-buffering (see related Wikipedia article) is used to figure out how objects in the scene occlude each other, depending on their "depth" within the scene. This technique requires some additional memory per pixel, called the "depth/stencil attachment" to keep track of the depth of various objects. Essentially, an attachment is a piece of memory, capable of being addressed in a per-pixel manner, that shaders can refer to (and even modify) as necessary. So, additional shaders and tests can be conducted, to perform a variety of operations needed to determine what data each fragment should finally contain (see the Wikipedia article on Fragments for more examples of such operations).

There is some general vocabulary to talk about this stage of rendering, where each fragment is being updated based on the colours attached to objects, lighting, depth, and so on. A "subpass" is a phase of rendering that reads and writes into certain attachments. Rendering commands, provided by the user are recorded in association with a particular subpass. Subpasses make up a "render pass", which is a collection attachments, subpasses, and "subpass descriptions". A subpass description describes which attachments are associated with a subpass (and whether they are read/write).

Some common type of attachments that are referred to in the specification are:

- color attachments: these include color information (perhaps also including how these colours should be blended) regarding various scene objects
- depth/stencil attachments: these are typically used for z-buffering (recall the Wikipedia article on this technique), and may be more specially called `depth attachments` in that case; otherwise, `stencil attachments` could refer to stenciling, which is a simple way of defining what parts of the screen are finally visible to the user (conceptually: if for example, if one wants to model looking through the keyhole of a door, one could `apply` a keyhole shaped stencil, which would block rendering of all pixels that are not within the keyhole)
- resolve attachments: useful in more advanced techniques, they store output data from programs (shaders?) which take into account multiple input attachments, or take into account how the data in a fragment might be influenced by its neighbours

Once a framebuffer is filled out with the finalized data in each fragment, it can be displayed to the user. Assuming the application is running on a personal computer, this would be done with the help of the PC's OS and accompanying "window system". However, the core Vulkan API doesn't assume this is a universal intention, so functionality relevant to this end is packaged separately in a Khronos (i.e. "Official") extensions, such as: `Swapchain`, `Surface`, `Win32Surface` or `XlibSurface`.

The Vulkan specification defines "presentation engine" as:

> The presentation engine is an abstraction for the platformâĂŹs compositor or display engine.

Vulkan provides an API for interaction with the presentation engine, which is an abstraction for all the pieces of hardware and software which take in graphics information, and then display it.

Importantly, the presentation owns "presentable images". One can think of these as blank slates which the application borrows from the presentation engine in order to fill it with relevant information (e.g. some graphics) (from framebuffers?). Once the application has filled in the presentable image, it returns it to the presentation engine, which will manage the display of the image. In particular, the presentation image doesn't simply paint the image onto a display as soon as the application hands it back, but rather, buffers presentable images through an abstraction called the "swap chain". In some sense, swap chains are an abstraction that facilitate the illusion of real-time, as Wikipedia explains:

> In computer graphics, a swap chain is a series of virtual framebuffers utilized by the graphics card and graphics API 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...

The Vulkan specification writes:

> A swap chain is an abstraction for an array of presentable images that are associated with a surface. The presentable images are represented by 'VkImage' objects created by the platform. One image (which can be an array image for multiview/stereoscopic-3D surfaces) is displayed at a time, but multiple images can be queued for presentation. An application renders to the image, and then queues the image for presentation to the surface.

The presentation engine is exposed via Vulkan through its relevant extensions. The key components of the presentation engine are:

- windowing system integration (WSI): a process responsible for organizing the display of data emitted by a phsyical device. A `platform` is an abstraction for a window system (e.g. X Window System, MS Windows, Mir, Wayland, Android), an OS, or some combination of the above (e.g. MS Windows, Wayland, Android)—basically, whatever the "platform" needs to display graphics data. Not all physical devices will have WSI support, and not all queue families on a physical device might support "presentation" of images to a specified surface. Thus, the application will need to find a suitable physical device, and relevant queue families on it which support a specified surface.
- "surface": an abstraction of a display surface. In particular, the surface abstracts away a lot of the platform specific details of what windowing systems may call a "window".
- "swap chain": abstraction providing buffering of presentable images.

We will now learn how to initialize a swap chain.

4: init-swap-chain.rs

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;
```

Previously, `enabledExtensionCount` and `ppEnabledExtensionNames` were ignored, but now we will specify extension `VK_KHR_swap chain` as enabled. The following steps are necessary, using `ash`:

1. `use ash::extensions::swap chain` loads the module which contains swap chain extension related stuff

2. create an array with the string `VK_KHR_swap chain` by using the `name` static `impl` of `swap chain` (track down the source for `ash::extensions::swap chain::name` as an exercise):

   ```
   let device_extension_names_pointers = [swap chain::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_poi⌋ nters.len() as u32` and `device_extension_names_pointers` respectively

Now, the `device` created in `init-swap-chain.rs` will have the swap chain extension enabled. Note that all the work was essentially done by the platform dependent conditionally compiled function `get_extension_names`.

We eventually want to create a surface, but a surface needs a "window" associated with it. In Rust as in C-ish, there are various windowing libraries, but this tutorial uses `extern crate winit`. This document will not discuss in depth the internals of a window system, which depend heavily upon the particular platform the application is running on. These details are explained in documentation for the `winit` crate, amongst other sources.

The following code in `init-swap-chain.rs`, within the function `create_events_loop⌋ _and_window` creates an event loop and 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();
```

The event loop handles messages sent between the user interface (the window), and our application (e.g. user input). Further details are presented in the documentation for `winit::EventsLoop`. A window is created using self-explantory calls setting the title, dimensions and event loop. Given the `window` from a windowing system, we create an associated surface using a platform dependent conditionally compiled function `create_surface`, which is defined within `init-swap-chain.rs`.

We now have a surface, but we have not yet found a physical device on which we can create logical devices. Previously, we found a physical device supporting graphics on our system. However, just like in the case of graphics, not all physical devices support presentation so this time we must find a physical device which supports both graphics and presentation. Essentially, a physical device supports graphics and presentation if it has queue families capable of supporting graphics and presentation operations. Some queue families may support both, but this is not always the case, so we will also need to determine which queue families on the physical device are of interest to us. All of this logic is handled in the function `find_pdevice_with_queue_family_supporting_graphics_and_presentation`. Note that in order to call this function, we need to pass a reference to an instance of a surface extension loader provided through the `Surface` Khronos extension. This extension allows us to do things like check if a physical device supports presentation to a specified surface; essentially, it contains all sorts of utilities for dealing with surfaces. To use the capabilities of the extension, we load in an instance of its loader, which handles our requests for various utilities. In particular, if we were using C-ish, we would query for a queue's presentation support relative to a surface using the function:

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

- `VkPhysicalDevice`: the physical device containing the queue
- `queueFamilyIndex`: index of some queue family offered by the physical device
- `surface`: the surface we created
- `pSupported`: a pointer to `bool32` sized memory; `vkGetPhysicalDeviceSurfaceSupportKHR` will set its value depending on whether the queue family supports presentation to the surface (True for supported, False for not supported)

In Rust-land, Ash's wrapper around the `Surface` extension allows us to call the function `get_physical_device_surface_support_khr` through an instance of the surface extension loader.

Having determined which physical device is of interest, and which queue families on it are relevant, we can now create logical device and set up queues on it, and then we can set up command buffers and pools managing those queues.

In C-ish, to create a swapchain, one would begin by setting up `struct VkSwapchainCreateInfoKHR`. We will explore the fields of this `struct` in detail later, and for now we learn how to query the physical device for information about the surface we have associated with it, so that we can use this information to fill in `VkSwapchainCreateInfoKHR` later. To query the physical device, we use the function:

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

This function fills out the following `struct`:

```
typedef struct VkSurfaceCapabilitiesKHR {
    uint32_t                        minImageCount;
    uint32_t                        maxImageCount;
    VkExtent2D                      currentExtent;
    VkExtent2D                      minImageExtent;
    VkExtent2D                      maxImageExtent;
    uint32_t                        maxImageArrayLayers;
    VkSurfaceTransformFlagsKHR      supportedTransforms;
    VkSurfaceTransformFlagBitsKHR   currentTransform;
    VkCompositeAlphaFlagsKHR        supportedCompositeAlpha;
    VkImageUsageFlags               supportedUsageFlags;
} VkSurfaceCapabilitiesKHR;
```

- `minImageCount`: minimum number of images physical device supports for a swap chain interfacing with surface (will be at least one)
- `maxImageCount`: maximum number of images physical device supports for a swap chain interfacing with surface (will be either 0, meaning that there is no upper limit (apart from memory) or greater than or equal to `minImageCount`)
- `currentExtent`: current width and height of the surface, or special value ($0xFFFFFFFF$, $0xFFFFFFFF$) indicating that surface size is governed by the extent of a swap chain interfacing with it
- `minImageExtent`: smallest valid swap chain extent for surface—width and height of the extent will be less than or equal to width and height of `currentExtent` (unless `currentExtent` is the special value)
- `maxImageExtent`: largest valid swap chain extent for surface—width and height given by `maxImageExtent` will be greater than or equal to width and height of `minImageExtent` and greater than or equal to width and height of `currentExtent` (unless `currentExtent` is the special value)

- `maxImageArrayLayers`: maximum number of layers presentable images can have for a swap chain created for this device and surface, and will be at least one—"image layers" are views in a multiview/stereo surface for stereoscopic-3D applications

- `supportedTransforms`: bitmask of `VkSurfaceTransformFlagBitsKHR` indicating presentation transforms supported for the surface (at least one bit will be set); `VkSurfaceTransformFlagBitsKHR` is an `enum`:

```
typedef enum VkSurfaceTransformFlagBitsKHR {
    VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR = 0x00000001,
    VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR = 0x00000002,
    VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR = 0x00000004,
    VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR = 0x00000008,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_BIT_KHR = 0x00000010,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_90_BIT_KHR = 0x00000020,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_180_BIT_KHR = 0x00000040,
    VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_270_BIT_KHR = 0x00000080,
    VK_SURFACE_TRANSFORM_INHERIT_BIT_KHR = 0x00000100,
} VkSurfaceTransformFlagBitsKHR;
```

  - `VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR`: specifies that the "identity" transformation (in mathematics, an identity operation returns its input unaffected) is applied to images being presented (i.e. no transformation is applied)
  - `VK_SURFACE_TRANSFORM_ROTATE_90_BIT_KHR`: specifies that presented images are rotated 90 degrees clockwise
  - `VK_SURFACE_TRANSFORM_ROTATE_180_BIT_KHR`: specifies that presented images are rotated 180 degrees clockwise
  - `VK_SURFACE_TRANSFORM_ROTATE_270_BIT_KHR`: specifies that presented images are rotated 270 degrees clockwise
  - `VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_BIT_KHR`: specifies that presented images are mirrored horizontally
  - `VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_90_BIT_KHR`: specifies that presented images are mirrored horizontally, then rotated 90 degrees clockwise
  - `VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_180_BIT_KHR`: specifies that presented images are mirrored horizontally, then rotated 180 degrees clockwise
  - `VK_SURFACE_TRANSFORM_HORIZONTAL_MIRROR_ROTATE_270_BIT_KHR`: specifies that presented images are mirrored horizontally, then rotated 270 degrees clockwise
  - `VK_SURFACE_TRANSFORM_INHERIT_BIT_KHR`: specifies that presentation transform is not specified, and is determined by mechanisms outside of Vulkan

- `currentTransform`: current `VkSurfaceTransformFlagBitsKHR` value for surface

- `supportedCompositeAlpha`: bitmask of `VkCompositeAlphaFlagBitsKHR`, representing alpha compositing modes supported by the surface on this physical device (at least one bit will be set) where is an enum:

```
typedef enum VkCompositeAlphaFlagBitsKHR {
    VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR = 0x00000001,
    VK_COMPOSITE_ALPHA_PRE_MULTIPLIED_BIT_KHR = 0x00000002,
    VK_COMPOSITE_ALPHA_POST_MULTIPLIED_BIT_KHR = 0x00000004,
    VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR = 0x00000008,
} VkCompositeAlphaFlagBitsKHR;
```

  - `VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR`: alpha channel, if it exists, of images is ignored and assumed to have value 1.0 (note that opaque composition can be achieved regardless of alpha composition mode by either using image formats with no alpha component, or by ensuring that all pixels in the presentable images have an alpha of 1.0)
  - `VK_COMPOSITE_ALPHA_PRE_MULTIPLIED_BIT_KHR`: alpha channel, if it exists, of images is respected by the compositing process; non-alpha channels of images are expected to already be multiplied by the alpha channel by the application (see "pre-multiplication" in Wikipedia's Alpha compositing article)
  - `VK_COMPOSITE_ALPHA_POST_MULTIPLIED_BIT_KHR`: alpha channel, if it exists, of images is respected by the compositing process; compositor will multiply the non-alpha channels of the image by the alpha channel during compositing
  - `VK_COMPOSITE_ALPHA_INHERIT_BIT_KHR`: treatment of images' alpha channel is unknown to the Vulkan API; application is responsible for setting composition mode using native window system commands, otherwise a platform-specific default will be used

- `supportedUsageFlags`: bitmask of `VkImageUsageFlagBits` representing how application can use presentable images of a swap chain associated with surface on the physical device. Recall that attachments are simply "image-shaped" pieces of memory. Thus, any image can serve as storage for an attachment, and the words image and attachment in fact, start to blur in meaning, their exact purpose becoming clear depending on how they are used. For `VkImageUsageFlagBits` settings to be used, `VkPresentModeKHR` of the swap chain must be set to `VK_PRESENT_MODE_IMMEDIATE_KHR`, `VK_PRESENT_MOD` ⌋ `E_MAILBOX_KHR`, `VK_PRESENT_MODE_FIFO_KHR` or `VK_PRESENT_MODE_FIFO_RELAXED_` ⌋ `KHR` (more on these modes later). `VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT` must be included in the set but additional usage may also be supported.

```
typedef enum VkImageUsageFlagBits {
    VK_IMAGE_USAGE_TRANSFER_SRC_BIT = 0x00000001,
    VK_IMAGE_USAGE_TRANSFER_DST_BIT = 0x00000002,
    VK_IMAGE_USAGE_SAMPLED_BIT = 0x00000004,
    VK_IMAGE_USAGE_STORAGE_BIT = 0x00000008,
    VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT = 0x00000010,
```

```
    VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT = 0x00000020,
    VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT = 0x00000040,
    VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT = 0x00000080,
} VkImageUsageFlagBits;
```

- VK_IMAGE_USAGE_TRANSFER_SRC_BIT specifies that image can be used as the source in a transfer command
- VK_IMAGE_USAGE_TRANSFER_DST_BIT specifies that image can be used as the destination in a transfer command
- VK_IMAGE_USAGE_SAMPLED_BIT specifies that the image can be sampled by a shader
- VK_IMAGE_USAGE_STORAGE_BIT specifies that the image can be used as for general purpose storage
- VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT specifies that the image can be used as a color or resolve (multi-sampling) attachment
- VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT specifies that the image can be used as a depth/stencil attachment
- VK_IMAGE_USAGE_TRANSIENT_ATTACHMENT_BIT specifies that the memory bound to this image will have been allocated with the VK_MEMORY_PROPERTY_LAZIL⌋ Y_ALLOCATED_BIT, which specifies that memory for the image is located on the physical device—this bit can be set for any image meant for use as a color, resolve, depth/stencil, or input attachment
- VK_IMAGE_USAGE_INPUT_ATTACHMENT_BIT specifies that the image can be read from a shader as an input attachment and be used as an input attachment in a framebuffer

This info struct describing surface properties will be used to fill VkSwapchainCreateIn⌋ foKHR, which is now introduced in full:

```
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`: type of this structure, which is `VkSwapchainCreateInfoKHR`
- `pNext`: set to `NULL` for now
- `flags`: bitmask of `VkSwapchainCreateFlagBitsKHR` indicating swapchain creation settings

  ```
  typedef enum VkSwapchainCreateFlagBitsKHR {
      VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR = 0x00000001,
      VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR = 0x00000002,
  } VkSwapchainCreateFlagBitsKHR;
  ```

  - `VK_SWAPCHAIN_CREATE_SPLIT_INSTANCE_BIND_REGIONS_BIT_KHR`: when set, specifies that images created from the swapchain can be stored in pieces of memory across a number of physical devices (an advanced feature we will not be using)
  - `VK_SWAPCHAIN_CREATE_PROTECTED_BIT_KHR`: when set, specifies that images created from the swapchain are protected (i.e. they are inaccessible to the application)
- `surface`: surface onto which the swapchain will present images
- `minImageCount`: is the minimum number of presentable images that the application needs
- `imageFormat`: `VkFormat` value specifying format of swapchain images (there are many different types of image formats supported and one fairly common one is `B8G8R8\_UNORM` which stores each pixel into memory by providing 8 bits of space for the B value in the first component, 8 bits of space for the G value in the second component, and 8 bits of space for the R value in the third component; 8 bits of memory can store integers from 0 to 255 inclusive (256 values total), but one can also store such an integer as a float by normalizing it (dividing it) by 255, hence `UNORM` for "unsigned norm")
- `imageColorSpace`: `VkColorSpaceKHR` value specifying how the presentation engine interprets each pixel in an image as a color (currently only `VK_COLOR_SPACE_SRGB_N` ⌋ `ONLINEAR_KHR` is supported)
- `imageExtent`: size (in pixels) of the swapchain images
- `imageArrayLayers`: number of views in a multiview/stereo surface (always set to 1 for us, since we do not use stereo surfaces)

- `imageUsage`: bitmask of `VkImageUsageFlagBits` describing the intended usage of swapchain images (`VkImageUsageFlagBits` were described earlier, as the `supportedUsages` member of `VkSurfaceCapabilitiesKHR` also returns a bitmask of `VkImageUsageFlagBits`, describing what sort of images a surface associated with the physical device in question can support)

- `imageSharingMode`: specifies how swapchain images can be shared

```
typedef enum VkSharingMode {
    VK_SHARING_MODE_EXCLUSIVE = 0,
    VK_SHARING_MODE_CONCURRENT = 1,
} VkSharingMode;
```

  - `VK_SHARING_MODE_EXCLUSIVE`: image access is exclusive to one queue family at a time
  - `VK_SHARING_MODE_CONCURRENT`: image access from multiple queue families at the same time (concurrently) is allowed

- `queueFamilyIndexCount`: number of queue families having access to swapchain images when `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`

- `pQueueFamilyIndices`: array of queue family indices having access to swapchain images when `imageSharingMode` is `VK_SHARING_MODE_CONCURRENT`

- `preTransform`: a `VkSurfaceTransformFlagBitsKHR` value (described earlier in the document) describing which transform should be performed on images before the presentation engine performs its final transformations

- `compositeAlpha`: a `VkCompositeAlphaFlagBitsKHR` value indicating the alpha compositing mode to use when surface associated with swapchain is composited with other surfaces

- `presentMode`: presentation mode swapchain will use which determines the order in which incoming present requests will be processed

```
typedef enum VkPresentModeKHR {
    VK_PRESENT_MODE_IMMEDIATE_KHR = 0,
    VK_PRESENT_MODE_MAILBOX_KHR = 1,
    VK_PRESENT_MODE_FIFO_KHR = 2,
    VK_PRESENT_MODE_FIFO_RELAXED_KHR = 3,
    VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR = 1000111000,
    VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR = 1000111001,
} VkPresentModeKHR;
```

  - `VK_PRESENT_MODE_IMMEDIATE_KHR`: presentation engine does not wait for monitor to finish updating image on screen—presentation is performed immediately, but this can lead to problems (called "tearing") where the user sees multiple frames

at the same time (as the monitor might still be in the process of clearing the last frame while the next one is already being drawn)

  - `VK_PRESENT_MODE_MAILBOX_KHR`: presentation engine waits for the next vertical blanking period to update the current image (thus taking care of tearing); single-entry queue is used to hold presentation requests while waiting; if queue is full, the new request replaces the (last?) existing entry, and images associated with the replaced entry become available for acquisition by the application
  - `VK_PRESENT_MODE_FIFO_KHR`: presentation engine waits for the next vertical blanking period to update the current image; "first-in-first-out" (FIFO) queue is used to hold pending presentation requests, i.e. new requests go to the end of the queue, and one request is removed from the front of the queue and processed during each vertical blanking period
  - `VK_PRESENT_MODE_FIFO_RELAXED_KHR`: presentation engine generally waits for the next vertical blanking period to update the current image, unless a vertical blanking period has already passed since last update (this can result in tearing); useful for reducing visual stutter (noticeably slow/stuttering animation) when one knows that application that will usually present a new image before the next vertical blanking period, but may occasionally be late; FIFO queue is used to hold presentation requests
  - `VK_PRESENT_MODE_SHARED_DEMAND_REFRESH_KHR`: presentation engine and application have concurrent access to a single image, called the "shared presentable image"; presentation engine only updates current image after a new presentation request is received from the application, i.e. presentation requests are not processed per the rate of monitor vertical blanking, thus, this can result in visual tearing if the application sends update requests too fast
  - `VK_PRESENT_MODE_SHARED_CONTINUOUS_REFRESH_KHR`: presentation engine and application have concurrent access to a single shared presentable image; presentation engine periodically updates the current image (the image displayed) per its own refresh cycle, without prompting from application through a presentation request, i.e. application only needs to to make one initial presentation request; the application can indicate the image contents have been updated by making a presentation request, but no guarantees are provided as to when the displayed image will be updated (again, tearing is possible if presentation engine's refresh period is faster than the vertical blanking period)

- `clipped`: specifies whether Vulkan is allowed to discard rendering operations for regions of the scene that are not visible (e.g. because viewing area is blocked by another window); clipped area of images will hold garbage data upon which shaders cannot peform meaningful operations; the Vulkan specification recommends:

> Applications should set this value to `VK_TRUE` if they do not expect to read back the content of presentable images before presenting them or after reacquiring them, and if their pixel shaders do not have any side effects that require them to run for all pixels in the presentable image.

- `oldSwapchain`: `VK_NULL_HANDLE` (i.e. no old swapchain), or the existing non-retired swapchain associated with surface; providing the old swapchain will allow the application to present any images acquired from it, and will also help with resource re-use

`init-swapchain.rs` shows how one would fill out the swapchain create info structure, and then create a swapchain, in a step-by-step fashion which should have no surprises. We finish this section by acquiring "views" of presentable images from the swapchain we created.

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 think is good for learning, and thus re-used.

2. 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.

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