Using Vulkan with Rust via Ash: A Samples Tutorial

Brian Merchant

0.1

1 Setup

Make sure:

1. you have the rust compiler installed: https://www.rust-lang.org (I am going to assume you've gone through the basics of Rust);
2. you have the Vulkan SDK installed: https://www.lunarg.com/vulkan-sdk/.

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

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

- how `struct`s and `enum`s defined in C-ish, and what `typdef` does;
- what a pointer is, and how it is related to C-ish arrays;
- the syntax of a C-ish functions;
- what a bit mask is, and how it's used in C-ish.

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

- what is `void`?
- why is a C-ish pointer evil compared to a Rust reference?
- what is `NULL`?
- why will you sometimes see `int*` written as the type for an array of `int`s?
- what is a function prototype?
- where is a function's return type and the type of its arguments, indicated in its prototype?

- do you know what a bit mask is in a broad sense (no need for details)?

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

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

3 Introduction

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

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

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

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

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

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

4 An Overview of Vulkan's Abstraction System: Instances, Devices and Queues

A Vulkan *instance* is a software object which your application uses to interact with Vulkan device(s). The physical device(s) in your computing system that one can use Vulkan to interact with are represented as members of the instance. Commands from your application are passed through a Vulkan instance into a *queue* (each associated with a logical device abstracting a subset of the physical device's resources) which pipes to and from the physical device.

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

5 `init-instance.rs`

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

```
VkResult vkCreateInstance(
const VkInstanceCreateInfo*                    pCreateInfo,
```

```
const VkAllocationCallbacks*                pAllocator,
VkInstance*                                 pInstance);
```

Let's take this apart:

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

Let's now have a closer look at `vkInstanceCreateInfo struct` definition:

```
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;
```

A brief description of the members:

- `sType`: indicates the type of the structure, in this case `VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_` 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`: usually set to `NULL`, and can be used to pass additional structures (whose type will be defined by the `sType` field), sometimes needed by API extensions;
- `flags`: currently no flags defined, set to `0`;
- `pApplicationInfo`: pointer to a `VkApplicationInfo` structure which we will study next;
- `ppEnabledLayerNames`: in this tutorial, we will not be using layers, so we can set this to `NULL`;

- `enabledLayerCount`: length of the `ppEnabledLayerNames` list, should be zero if `ppEnabledLayerName`

- `ppEnabledExtensionNames`: at this point in the tutorial, we are not using extensions, so we'll be setting this to `NULL`;

- `enabledExtensionCount`: length of the `ppEnabledExtensionNames` list;

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

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

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

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

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

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

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

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

- note the `unsafe` block;

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

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

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

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

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

- note how we have to manually destroy the instance.

To see everything in action:

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

6 enumerate-devices.rs

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

- `getListData` will have prototype:

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

- get a pointer to some memory set aside for an unsigned 32 bit integer, and call it `pCount`;

- call `getListData(aStruct, pCount, NULL)`—the `NULL` pointer for `pStuff` indicates that we don't yet know how many things will be in the list, so we want that information to be put in the memory pointed to by `pCount`, and if the `vkResult` we get back after the call indicates success, `pCount` will point to an integer representing the number of `vkStuff` we need to set aside memory for;

- set aside appropriate memory for the list of `vkStuff` (now that we know the count), and get a pointer to that list called `pStuff`;

- call `getListData(aStruct, pCount, pStuff)`—this time, because `pStuff` is not `NULL`, and the `getListData` will fill out the list pointed to by `pStuff`, instead of writing information about the count.

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

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

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

1. call `vkEnumeratePhysicalDevices(instance, pPhysicalDeviceCount, NULL)`—if `vkResult` indicates success, then `pPhysicalDeviceCount` now points to an integer denoting the number of physical devices available;

2. allocate enough memory for a list of `VkPhysicalDevice`s containing `*pPhysicalDeviceCount` items (in C-ish the list would be of type `VkPhysicalDevice*`), and call it `pPhysicalDevices` (the preceding `p` indicates "pointer" in C-ish naming conventions);

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

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

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

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

7 `init-device.rs`

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

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

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

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

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

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

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

What's inside C-ish `vkQueueFamilyProperties`?

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

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

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

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

Here's what's going on with the bit mask stuff:

- converting from hexadecimal to binary:

`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;

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