Using Vulkan with Rust via Ash: A Samples Tutorial

Brian Merchant

0.1

1 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).

2 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.

3 `init-instance`

If we were using C, 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 `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 (a C pointer is similar to a Rust reference, but with some important differences) 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 concept which provides a pointer to the "tip" of some data structure: i.e. it gies you a handle to some data structure, whose internals aren't transparent to you).

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`! Some points for your consideration:

- note comments explaining what `use` statements;
- 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 `FunctionPointers` and their supporting infrastructure into 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.

Don't forget to call `cargo run` for `init-instance` to make sure that everything compiles.

4 enumerate-devices

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:

1. Application calls the function with a valid pointer to an integer for the count argument and a `NULL` for the pointer argument.
2. The API fills in the count argument with the number of objects in the list.
3. The application allocates enough space to store the list.
4. The application calls the function again with the pointer argument pointing to the space just allocated.

Alright, so what does the function prototype for `vkEnumeratePhysicalDevices` look like?

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

In Rust, `ash::Instance` object has an `impl` function which calls `vkEnumeratePhysicalDevices` and returns the result. Let's examine how it works (see documentation for `ash::Instance`, and click `src`):

```
fn enumerate_physical_devices(&self) -> VkResult<Vec<vk::PhysicalDevice>> {
        unsafe {
                let mut num = mem::uninitialized();
                // call enumerate_physical_devices (through function pointers to under
                self.fp_v1_0().enumerate_physical_devices(
                        self.handle(),                  // handle to an integer
```

```
                &mut num,                           // pointer to where we are goi
                ptr::null_mut(),          // null pointer, since we are querying
        );

        // result of first call is now stored in `num`
        // allocate the vector for physical devices based on count
        let mut physical_devices = Vec::<vk::PhysicalDevice>::with_capacity(num

        // call `vkEnumeratePhyiscalDevices` again, with a pointer to the vec
        let err_code = self.fp_v1_0().enumerate_physical_devices(
                self.handle(),
                &mut num,
                physical_devices.as_mut_ptr(),
        );

        physical_devices.set_len(num as usize);
        match err_code {
                vk::Result::Success => Ok(physical_devices),
                _ => Err(err_code),
        }
    }
}
```

So under the hood, Ash follows exactly the pattern we noted above.