# TP02
# Vulkan setup
# Instance, Physical devices and Queue families

Robin Faury

01/08/20

**Abstract**

In this practical work, we will see how to setup Vulkan and how to create a Vulkan Instance, Physical devices and Queue families.

## 1 Driver

To use the latest API of Vulkan (or any graphic API) you must have the latest graphic driver. First, we need to find the name of your device. You can use the "run" application on Windows ('Windows key'+ 'R') and write "msinfo32". On the "Components" section, select "Display" and check the name of your GPU. In the case of Nvidia card you can go on their website (https://www.nvidia.fr/Download/index.aspx?lang=en) to download the driver. On the "Download Type" section chose "Game Ready Driver". This driver allows you to use latest GPU features.

## 2 Vulkan

You can find the Vulkan SDK on the vulkan lunarg (the developer of the Windows and Linux Vulkan's SDK) website (https://vulkan.lunarg.com/home/welcome). Choose the version 1.1.130.0 and install it. On this git repository you can find the CMakeList.txt to create project linked to Vulkan.

```
https://github.com/robinfaurypro/GPGPU_ISIMA_2019-2020.git
```

Vulkan can be seen as a set of structs identify by the sType member. This member must be filled every time you create a Vulkan object. It always starts with "VK_STRUCTURE_TYPE_" and the name of the struct. The second member is pNext. It allows a linked list of structures to be passed to functions. For this practical pNext will be set to nullptr for each struct created.

## 2.1 Instance

For this practical we will create only one Vulkan Instance for our application. The Instance object store all tracked states (The current buffer, program, pipeline, etc...).
The prototype of the function for instance creation is:

```
VkResult vkCreateInstance(
        const VkInstanceCreateInfo*,
        const VkAllocationCallbacks*,
        VkInstance*);
```

We must instantiate a VkInstanceCreateInfo object, but this object need a VkApplicationInfo. We first fill this object like this:

```
VkApplicationInfo applicationInfo = {};
applicationInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
applicationInfo.pApplicationName = "Vulkan lesson";
applicationInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
applicationInfo.pEngineName = "No Engine";
applicationInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
applicationInfo.apiVersion = VK_API_VERSION_1_0;
```

We can now fill the create info struct:

```
VkInstanceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &applicationInfo;
createInfo.enabledExtensionCount = 0;
createInfo.enabledLayerCount = 0;
```

And create the instance:

```
VkInstance instance;
VkResult result = vkCreateInstance(&createInfo, nullptr, &instance);
```

This function uses the createInfo to fill the instance handle. In Vulkan a handle is 64 bits wide.
Finally, we must check the error returned:

```
if (result != VK_SUCCESS) {
        throw std::runtime_error("Error during the Vulkan Instance creation.");
}
```

Don't forget to destroy the instance at the end of your application using:

```
vkDestroyInstance(instance, nullptr);
```

## 2.2 Physical devices

Vulkan deal with two types of devices, the physical one and the logical one. Physical devices are hardwares. They can be graphics card, DSP (Digital Signal Processor), etc... Each system have a fixed number of physical devices. Logical devices are abstractions of physical devices. It is configured in a way that is specified by the application. The first step is to choose the correct device for our application. First, ask Vulkan how many devices are there in our system using:

```
uint32_t physicalDevicesCount = 0;
vkEnumeratePhysicalDevices(instance, &physicalDevicesCount, nullptr);
```

vkEnumeratePhysicalDevices will fill physicalDevicesCount with the number of physical devices and fill the last parameter with handles to physical devices. For the first run of this function we don't know how many devices are connected. Now we have this number (feel free to throw an error if the number is 0) so we can call the function a second time:

```
std::vector<VkPhysicalDevice> physicalDevices(physicalDevicesCount);
vkEnumeratePhysicalDevices(
        instance, &physicalDevicesCount, physicalDevices.data());
```

Then, we need to select the best candidate for our application. For each device we need to check its properties and features.

```
VkPhysicalDeviceProperties physicalDeviceProperties;
vkGetPhysicalDeviceProperties(physicalDevice, &physicalDeviceProperties);

VkPhysicalDeviceFeatures physicalDeviceFeatures;
vkGetPhysicalDeviceFeatures(physicalDevice, &physicalDeviceFeatures);
```

Feel free to read all properties and features stored in these object. You must select the one match for the best to your program. In this case, the property "deviceType" will help us. Its value can be:

- VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU = 1
- VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU = 2
- VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU = 3
- VK_PHYSICAL_DEVICE_TYPE_CPU = 4
- VK_PHYSICAL_DEVICE_TYPE_OTHER = 0

Integrated GPUs are embedded devices coupled with the host, virtual GPU is run in virtual environment and "CPU" is running on the same processors as the host. The discrete GPU means the device is separate processor connected to the host. We must choose this one.

## 2.3 Queue families

Every work submitted to queues will be executed by Vulkan devices. Physical devices have several queue families. A queue family can be identified by its flag and the number of queue that can be run in parallel.
Using the same way that vkEnumeratePhysicalDevices get the number of queue families using vkGetPhysicalDeviceQueueFamilyProperties and fill a vector of VkQueueFamilyProperties.
In this practical we will use the compute feature provide by the graphic card. For each queue family, check if the queueFlags contain VK_QUEUE_COMPUTE_BIT.

```
if (queueFamily.queueFlags & VK_QUEUE_COMPUTE_BIT) {
        // select this queue family
}
```