Introduction to Vulkan using Python

Miguel Monteiro, miguel.ecm@ua.pt

***Abstract*—A guide to getting started with the Vulkan graphics library using the Python programming language.**

This article explains how to setup the environment required to work with the library and gives an overview of how Vulkan works.

Included in this article is a tutorial that helps the reader become familiarized with Vulkan by going over some characteristics of its usage.

***Index Terms*—Vulkan, graphics library, Python, tutorial**

I. Introduction

Vulkan is an open-source graphics library developed by the Khronos Group as the successor to OpenGL (Open Graphics Library).
Some of the benefits of Vulkan include being cross-platform, having low-overhead, incorporating multi-threading support, providing improved shader debugging and being highly customizable.

Such benefits should be considered when choosing the graphics library to use when building an application. However, it should also be noted that Vulkan is significantly harder to use when compared to other libraries, in part due to its explicit nature. This complexity will often result in longer development times.

In this article it will be provided an overview of Vulkan and some of its characteristics, along with a tutorial that will help the reader take their first steps into Vulkan.

II. Vulkan Overview

In this section it is given an overview of the main components that are used in a Vulkan program and that are implemented in the files of the tutorial.

A. Instance

The first thing to do when setting up Vulkan is creating an instance. This will be the root of our program where we choose the minimum API (Application Programming Interface) version to support and create a context where our application will run.

When creating the instance, it must be specified what instance extensions will be enabled, as Vulkan will not do this by default, this way it saves resources by avoiding enabling extensions that will not be used when running the program. To find out which extensions are required to create the window we can query the library used to create it which, in the case of this tutorial, is GLFW (Graphics Library Framework).

B. Physical and Logical Devices

After an instance has been created, we can use it to find a physical device on which to run the program, which will be one of the GPUs (Graphics Processing Units) in our system. Using “VkPhysicalDevice”, which is the handle the Vulkan graphic library uses to represent a GPU, we can query many of its features, such as the amount of memory it has available or which queue families it supports.

With this information we will then create a logical device (using the “VkDevice” handle) by requesting which features of the physical device will be used in the program. This will ensure that the driver does not waste time performing checks on features we do not need.

Once this device is created, it can then be used in the various operations of the program.

C. Surface and Swapchain

When it comes to rendering to a screen, there are two components required, a surface and a swapchain.
Starting with the surface, (for which the respective Vulkan handle is “VkSurfaceKHR”) it serves the purpose of abstracting the native window handle of the windowing library (GLFW in this example) which is necessary as Vulkan supports multiple platforms.

Moving on to the swapchain, it is the component that enables presenting rendering results to a surface. It is, essentially, a collection of render targets that allow the image being rendered to be separate from the one being displayed on the screen, in other words, creating a buffering system. The logic the swapchain follows is defined by its “present mode”, for which there are six options (described in detail in the “VkPresentModeKHR” page on Vulkan’s documentation [1]) that basically specify the relationship between the moment the image is presented to the screen and the screen’s refresh rate.

D. Graphics Pipeline

In Vulkan the graphics pipeline is not ready-made, almost every stage of it needs to be configured. This also means that when you want to make any change to the pipeline, you need to recreate it. To do so, you can prepare a “VKPipeline” object for each different rendering scenario. Note that some basic configuration, like altering the viewport size or the clear color, can be done dynamically.

This particularity of Vulkan, results in more optimization opportunities as the large changes being applied to the pipeline in runtime are explicitly detailed in advance.

E. Shader Modules

Another important aspect of Vulkan is its approach to shader modules. While the various shaders, such as the vertex and fragment shaders, are still written in a high-level language such as GLSL (OpenGL Shading Language) Vulkan does not read GLSL directly. Instead, it uses SPIR-V (Standard Portable Intermediate Representation).

As defined in the official specification [2], SPIR-V is "a simple binary intermediate language for graphical shaders and compute kernels". In practice, it is a bytecode that converts code written in high level languages into optimized binary code.

The use of SPIR-V allows for shaders to be written in an array of different languages as they are then converted into the same standard. It also provides better debuggability in the case of a shader compilation failure.

III. TUTORIAL

In this tutorial, the following topics will be covered:

- configuring a python environment for Vulkan development
- creating the instance Vulkan component
- compiling shaders using the SPIR-V format
- rendering a triangle

A. Environment Setup

To start with, it is required to have the Vulkan SDK installed (available on the "LunarG" website [3]). It is also necessary a text editor compatible with the Python programming language and the Python packages "vulkan"[4] and "glfw"[5] (both available on the "Python Package Index" website).

B. Application Overview

The starting point for this tutorial is the code provided in the repository mentioned in the appendix of this article. Note that there are two project folders, one named "vulkan-tutorial" which contains the code to be used as base for this tutorial and another one named "vulkan-tutorial-completed" which contains the expected result of this tutorial.

Start by opening the "main.py" file, in the code there is already an "Engine" class which implements a graphics engine that manages the Vulkan components implemented in this project. There is also an "App" class where the graphics engine is being run and a window is being created, using the GLFW library.

If you try to run the code now, all that will be visible is an empty window that quickly closes, and an error message will be printed to the debug console where it is mentioned that the Vulkan instance is missing an extension. This is because we have yet to create a Vulkan instance, which we will be doing next.

C. Creating an Instance

First, create a new Python file in the project folder named "instance.py" and open it.

We start by importing the "vulkan" package. This will give access to many Vulkan functions, although not all, some will still need to be manually accessed. We will also import the "glfw" package as we will need to query it later. Add this as the first lines of code:

```
from vulkan import *
import glfw
```

If we look at Vulkan's documentation [1], instance creation is done using the "vkCreateInstance" method which requires a "VkInstanceCreateInfo" structure with information on instance creation which in turn requires a "VkApplicationInfo" structure with information on application name and versions.

To do all this we will create a method named "create_instance" that will have a parameter for the application name. In this method, we will define the Vulkan API version we are going to use (I suggest version 1.0.0 as it has everything required and will increase compatibility), and then create the "VkApplicationInfo" structure mentioned above, passing in the application name and API version. This can be accomplished with the following code:

```
def create_instance(applicationName):

    vulkanVersion = VK_MAKE_VERSION(1, 0, 0)

    appInfo = VkApplicationInfo(
            pApplicationName = applicationName,
            apiVersion = vulkanVersion
    )
```

With this done we move on to the "VkInstanceCreateInfo" that will receive our application information and, since everything in vulkan is opt-in, which extensions will be enabled.

To find out what extensions we need we will ask GLFW which extensions it needs to run, using its "get_required_instance_extensions()" method.
Then add a *for loop* to print out each extension name so that we can see what the required extensions are:

```
requiredExtensions =
glfw.get_required_instance_extensions()
print("------------- Required Extensions ------------
-")
for extension in requiredExtensions:
    print(f"{extension}")
```

One more thing we should do is check if these extensions are supported so, for that purpose, we will create a method named "check_extension_support" that will receive extensions to check.

To find which extensions are supported, we call the Vulkan function "vkEnumerateDeviceExtensionProperties()" passing in None as we don't want to specify any layer. This returns an array with the extensions provided by the Vulkan implementation. We then insert this array in a for loop so that for each extension in this array its name is added to another array where the names of the supported extensions will be saved. As done with the required extensions, add a *print* for these too so we can check the supported extension names in the debug console later.

Next, add another for loop to find out if each of our required extensions are in the supported extensions array, returning False in we find one that is not.

To conclude this method, add a return True at the end as, if the program reached this point, it means that all the extensions we wanted to check are supported. The code for this method should look something like this:

```
def check_extension_support(extensionsToCheck):

    supportedExtensions = []
    for extension in
vkEnumerateInstanceExtensionProperties(None):
        supportedExtensions.append(
extension.extensionName)

    print("------------- Supported Extensions -------
------")
    for extension in supportedExtensions:
        print(f"{extension}")

    for extension in extensionsToCheck:
        if extension not in supportedExtensions:
            return False
    return True
```

With this done we can now instantiate the "VkInstanceCreateInfo()" passing in the application info created previously, the number of extensions that are required and their names.

At last we call the vulkan function "vkCreateInstance()" passing in our creation info and a null for the allocator. Add the following to the code of the "create_instance" method:

```
if check_extension_support(requiredExtensions):
    createInfo = VkInstanceCreateInfo(
        pApplicationInfo = appInfo,
        enabledExtensionCount =
len(requiredExtensions),
        ppEnabledExtensionNames = requiredExtensions
    )

    return vkCreateInstance(createInfo, None)
```

To check if the code is working, we can go back to the "main.py" file, add the import for the instance script, and call the "create_instance" method we created in the first line of the "make_instance" method of the "Engine" class:

```
import instance

(...)

def make_instance(self):
        self.instance =
instance.create_instance(self.appName)
```

If we run the code, we will still have an empty window, but it now has red background, which means the Vulkan components are operational, and on the debug console we can read the names of the required extensions and supported extensions.

D. Creating Shaders

In the shaders folder within the project folder there are files named "shader.frag" and "shader.vert", that we will open. In these files, we already have code for simple fragment and vertex shaders where we define the position and color of three vertices that form a triangle. Note that the code inside of the main function of each shader is commented, so it is not running. To get it to run, remove the forward slashes.

However, if we run the project now, we still get the same empty window with the red background.That is because even though the code for the shaders is written in the GLSL language in these "shader.frag" and "shader.vert" files, for Vulkan to use them we must compile them to SPIR-V files.

To that end we can create a ".bat" file inside the shaders folder (call it compile_shaders.bat) where we will write a line for each shader to be compiled using the VulkanSDK installation in our computer. To do this write the path to the glslc.exe file in the "Bin" folder of the Vulkan SDK (the path should be something like "C:\VulkanSDK\<vulkan-version-number>\Bin\glslc.exe") followed by these lines of code:

```
C:\VulkanSDK\1.3.231.1\Bin\glslc.exe  shader.vert -o
vert.spv
C:\VulkanSDK\1.3.231.1\Bin\glslc.exe  shader.frag -o
frag.spv
```

Note that while the file names for our shaders can be different, the file extensions must be ".frag" and ".vert" otherwise the Vulkan SDK will not recognize them.
Run the .bat file and you will notice new "vert.spv" and "frag.spv" files were generated, these are the SPIR-V format files that Vulkan works with and that are being loaded in the "Engine" class.

F. Hello Triangle

Go back to the "main.py" file, run it, and you will finally see the triangle being rendered.

IV. Conclusions

Having reached the end of this article and the included tutorial, the reader should now begin to understand the overall concepts of Vulkan and the application of its various components.

The information provided will also aid in deciding whether the use this graphics library could be beneficial in the development of an application.

Appendix

The companion code to this tutorial is available at: https://github.com/Exodus09/Vulkan-Tutorial. This code was based on the code developed in Andrew Mengede's "Vulkan with Python" video series [6] which is recommended for further learning.

Also recommended are Victor Blanco's "Vulkan Guide" [7] and Alexander Overvoorde's "Vulkan Tutorial" [8] although it should be noted that the code provided with both tutorials is written in the C++ programming language and, as such, having prior experience with the language is advised.

REFERENCES/BIBLIOGRAPHY

[1] N/D, “Khronos Vulkan Registry”, *Khronos Group*, The Khronos Group Inc., N/D, https://registry.khronos.org/vulkan/

[2] Abbott, Connor, et al. “SPIR-V Specification”, *Khronos Group*, The Khronos Group Inc., 21/06/2022, https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html

[3] N/D, “Vulkan SDK”, *LunarG*, LunarG, Inc, N/D, https://www.lunarg.com/vulkan-sdk/

[4] N/D, “vulkan”, *Python Package Index*, Python Software Foundation, 20/06/2019, https://pypi.org/project/vulkan/

[5] Rhiem, Florian, “glfw”, *Python Package Index*, Python Software Foundation, 07/09/2022, https://pypi.org/project/glfw/

[6] Mengede, Andrew, “Vulkan with Python”, *YouTube*, Google LLC, 06/10/2022, https://www.youtube.com/playlist?list=PLn3eTxaOtL2M4qgHpHuxY821C_oX0GvM7

[7] Blanco, Victor, “Vulkan Guide”, *Vulkan Guide*, N/D, 24/10/2020, https://vkguide.dev/

[8] Overvoorde, Alexander, “Vulkan Tutorial”, *Vulkan Tutorial*, N/D, Apr. 2022, https://vulkan-tutorial.com/Introduction