2

Getting Started with Vulkan

Vulkan is a newer generation of cross-platform high-performance 3D graphics and compute API from Khronos Group. Released in February 2016, the API is especially designed to take the advantage of modern GPU hardware pipeline to meet the demanding requirement of modern applications such as games, interactive data visualization and interactive media applications. Unlike its predecessor OpenGL and other traditional APIs, Vulkan allows a direct access to the underlying GPU. Its explicit nature of programming model provides fine grain control over the application in the hands of programmer guarantying predictability and lag-free experience.

This chapter provides an overview of Vulkan and its differentiating features over OpenGL. We will learn to setup the Vulkan SDK and create our first Vulkan application from scratch. As a modern API, Vulkan brings many new jargons and distinctive programming model, we will learn these in a step-by-step manner and as we proceed through the chapter.

In this chapter, we will cover the following topics and by the end of the chapter you should be able to run your first Vulkan application to render a triangle on your system.

* Introduction to Vulkan
* Vulkan Setup
* Vulkan Programming Model
* Hello World in Vulkan - Rendering a Simple Triangle
* Summary

# Introduction to Vulkan

Vulkan is a new high-performance 3D graphics and compute API for modern GPU pipeline architectures. Designed from scratch, the API promises to meet-up long time demanding requirement of the community. Vulkan give programmers a low level accessibility of the hardware offering extensive capabilities to exploit and exhaust the GPU. The API comes in a full SDK bundle form, providing tools, debugger, profiler, tracer and many more utilities.

Vulkan is not alone in the race, it’s competing with Microsoft’s Direct-X 12 and Apple’s Metal. Both these API’s are platform specific limiting to Windows variant and Mac platform respectively, whereas, Vulkan is the first choice of many programmers due to it cross platform nature supporting Windows, Linux, Tizen, Steam OS and Android.

Unlike its predecessor OpenGL, Vulkan is verbose in nature which is infact a boon as it provides full control over the rendering pipeline making it free from rendering hitches and unpredictable behavior, in the next section will discuss Vulkan and OpenGL differences in detail. The driver layer of Vulkan is thin therefore it puts more responsibilities on the shoulders of an application programmer. That means jobs like resource management, presentation and synchronization are not performed by the driver and remain with application, therefore, as a graphics or compute programmer you have to know your job.

## Vulkan vs OpenGL

Here are the features/improvements in Vulkan that give it an edge over OpenGL:

1. **Driver overhead and CPU usage:** Vulkan offers reduced driver overhead compare to OpenGL, it is designed to avoid error checking and validation at runtime. These are optionally done by application at the development stage. The programmer has explicit control over the resources, the control logics are directly in the hands of programmer rather embedded into underlying drivers, this makes the driver layer thinner, resulting it much closer to underlying graphics hardware.

In contrast, OpenGL has heavy driver overhead due to extensive error checking to prevent system crash.

1. **Programming Model:** OpenGL is a global state machine, users do not have direct control over the graphics pipeline and it is created behind the curtains. These binary states (on/off) are used to build the dependency mapping in the driverand applied on the underlying pipeline.Since the states are global, they can potentially affect other modules unintentionally leaving behind the programmer spending precious time and efforts fixing such bugs.

Vulkan on the other hand is more object based API, here states are directly associated with graphics pipeline. User can build numerous graphics pipeline objects programmatically with different states and flags. This programming model excludes overhead of managing complex dependency mapping, also states meant for one type of graphics pipeline does not affects the other.

1. **Multi-Thread scalability:** In OpenGL, the rendering jobs submitted to the GPU are blocking in nature that means the CPU has to block its work until the previous submitted work is not completed. The GPU does not provide synchronization mechanism indicating the states of these underlying work blocks. This mean effectively only a single thread is communicating to the GPU and this way OpenGL cannot take the full advantage of the multi-threading.

Vulkan is specially design to encourage multi-threading, unlike the OpenGL where work request is submitted implicitly behind the curtains, in Vulkan it is explicitly submitted by the programmer in the form of command buffers. A command buffer is a small unit of work that is requested by the application. Each command buffer can be tracked using synchronization. In addition, there are my other many synchronization mechanism that provide a higher and lower level of synchronization means. This mean multiple thread can do respective work simultaneously while keeping an eye on the previous work in parallel using these synchronization mechanism exploiting GPU to its limit.

1. **Explicit API:** OpenGL is an implicit API, resource management is driver’s responsibility. Driver takes application hints and track the resources which is unnecessary overhead.

Vulkan is an explicit API; here, the driver is no longer responsible for tracking resources and their relationships – this is moved to the application. This clean approach is more predictable, the driver is not doing gymnastics behind the curtains to manage resources (like in OpenGL). As a result, the job processing is streamlined and straightforward result optimal performance and predictable behavior.

1. **Shading Language:** In OpenGL, the driver converts GLSL shaders to GPU shader instruction at run-time causing slo wdown in application load time. Vulkan supports a new type intermediate shading language representation called SPIR-V (**S**tandard **P**ortable **I**nte**r**mediate **R**epresentation – **V**ulkan). This representation can take input in standard available shading languages like GLSL, HSL shader precompile into intermediate binary form.
2. **Memory Management:** Vulkan offers an application to choose from different types of memory. Vulkan advertises these memory types available on the system, so that it can be used appropriately intended resource type.

OpenGL do not have a concept where an application can specify its interest in the kind of memory it prefers to use. The OpenGL driver manages memory implicitly with internal control logics, now this heuristics approach may vary from one driver to another which may be not optimal for every hardware.

1. **Predictable:** The explicit nature of Vulkan implements controller logic at the application level keeping the driver light weight. Also, the render jobs are explicitly submitted by the application and executed upfront. This makes Vulkan is extremely predictable as no hidden premature optimizations carrying out at the driver level. As a result, you experience no lags or hitches.

OpenGL jobs are performed in batches and controlled by the driver making it less predictable.

1. **Unified API:** Vulkan is a single API for all different platforms, this goes well with desktop and mobile platforms. However, this is not the case with OpenGL it has two different version catering desktop and mobile version. With Vulkan no need to spinoff different version between desktop and mobile version, also no more worrying about whether an OpenGL API is supported in OpenGL ES.
2. **Error Check and Validation:** In OpenGL, error checking and validation are an integral part of the execution, this is an overhead at runtime.

On the other hand, these checks are optional and the validation are managed through validation layers. This allows the application to enable selective layers of interest.

## Is OpenGL Dead?

Featured as a next generation API, Vulkan’s roll out in market raises an important question –*Is OpenGL dead*? This is not true, while Vulkan offers many shiny features ahead of OpenGL, it learning curve is steeper compare to OpenGL. OpenGL is easy to learn.

Vulkan’s verbose nature requires the application programmer to know his job, in OpenGL the driver takes care of many complex logics behind the curtains. Also, for smaller application where performance does not matter OpenGL wins the argument. OpenGL is still evolving and supports are available by vendors, new specs are release time-to-time with newer features.

Vulkan is the primary choice for modern graphics enthusiast and tech savvy’s, who wants to remain at the cutting edge of the technologies, harnessing and exploiting the power of GPU with no compromise. It’s best suitable for gaming applications, heavy visualization and scientific simulations.

## Why Vulkan?

In the last few decades OpenGL from Khronos group and DirectX (up to DirectX 11) from Microsoft are the primary API for many developers for developing graphics and gaming application. These APIs evolved over time as well as introduced complexity to support backward compatibility, better validation support etc. and eventually hit a roadblock where the API became too heavy for high throughput graphics and gaming applications.

Vulkan API is designed from ground up to deliver high throughput graphics application. The API also provided support to maximize full potential of the recent CPU and GPU architectural changes over the past few decades. A graphics or gaming applications performance can be primarily categorized into 2 categories: CPU bound or GPU bound.

**CPU bound application**: An application is CPU bound when the application takes too much time to send the commands to the GPU, causing GPU to go idle intermittently. The frame rate of the application can be measured by the time taken by the CPU for each frame.

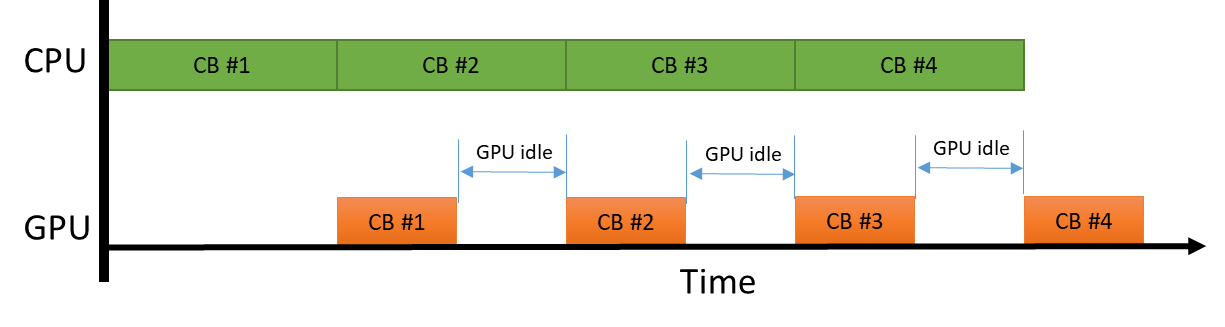


Figure : CPU Bound Application

In the above diagram, CB refers to the command buffer and the length of the block represent the time taken by CPU to create the command buffer and submit to GPU and the length of the block in the GPU side represent the time taken to execute the command buffer in GPU. Since the time taken by the CPU to generate and submit the command buffer is greater than the time taken by the GPU to process the command buffer this scenario is called “CPU bound” case.

**GPU bound application**: An application is GPU bound when the performance of the application is based on how fast the GPU executes the commands from the application. The frame rate of the application can be measured by the time taken by the GPU for each frame.

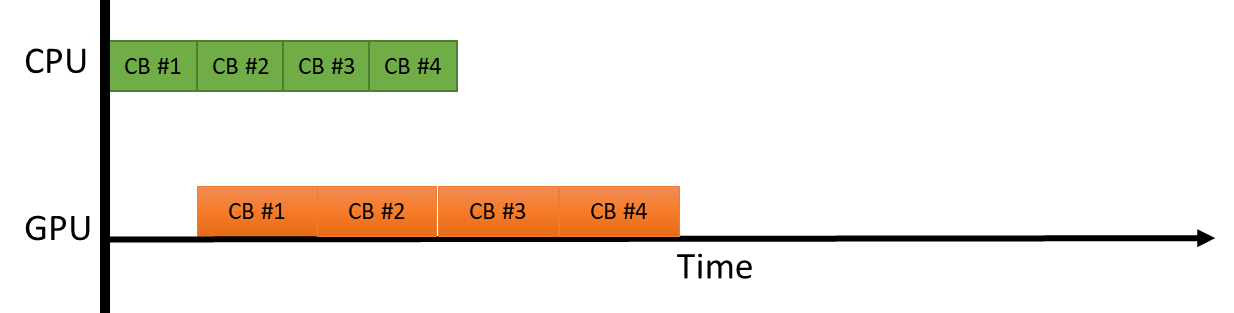


Figure : GPU Bound Application

In the above diagram, when the GPU is busy from time the first command buffer (CB#1) is executed till the last command buffer (CB #4). This is an ideal use case for the GPU where the GPU is kept busy with no internment idle states.

Vulkan APIs are designed to utilize the CPU & GPU cycles efficiently thus by providing maximum throughput in the graphics/gaming applications.

# Installing Vulkan

This section describes the step-by-step process to install Vulkan SDK. This includes minimum system requirements, software components and post installation analysing and testing.

## System Requirements

The following are the minimum system configuration:

* Microsoft Windows OS (Windows 7, 8, 8.1 or 10), preferably 64bit OS
* 8GB of system memory should suffice better throughput.
* Graphics driver supporting Vulkan API.

## Installation

Please follow the below instructions for Vulkan installation:

1. **Install CMake:** Next, install CMake 3.0 or newer from <https://cmake.org/>. While installing select “*Add CMake to the system PATH for all users*”. We will use CMake to build some additional optional libraries and tools in Vulkan SDK, additionally examples followed in this book are also based on CMake build.
2. **Install Python:** Install python and make sure you add it to the path. This can be done by simply checking the “*Add Python <version> to PATH*” in the install dialog’s check box.
3. **Install SDK:** Download and install the open-source LunarG Vulkan SDK from <https://vulkan.lunarg.com>, the default location suggested by the installer (‘C:\VulkanSDK’) should be fine. Upon successful installation, the Lunar-G SDK contains multiple folders in it, such as:-
   1. **Documentation:** Consists of Vulkan specification, manual pages and tutorial.
   2. **Bin:** Contains demo binaries and other tools such as shader validator, Vulkan tracer and replayer, and SPIR-V shader conversion executables.
   3. **glslang:** This contains source code for frontend parser for GLSL, building this project with CMake creates and a standalone wrapper tool for shader validation called glslangValidator. In addition, it produces OGLCompiler.dll which is helpful in programing it helps converting glsl shaders directly into SPIR-V binary representation.
4. **Vulkan driver:** Based on the video card in your system, you can download the latest graphics driver from <https://www.khronos.org/vulkan/> under the “Vulkan Driver” section.
5. **IDE:** Download Visual Studio Community edition (2015 or newer) on Windows.
6. **Qt SDK:** Qt 5.6 or greater SDK for user interface.

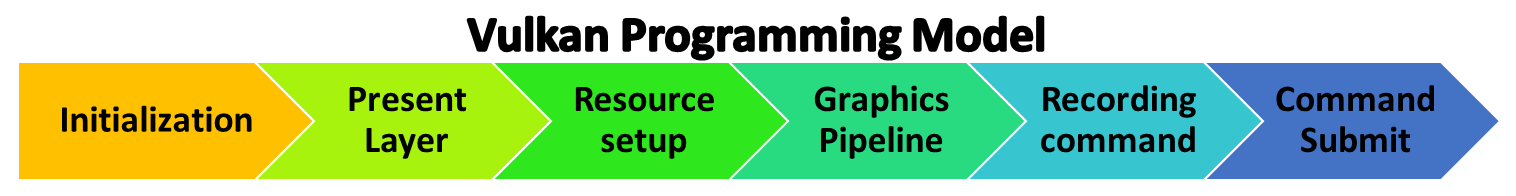
## Testing Installation

At this stage your system should be ready to run a Vulkan sample application.

* To check this open a new command prompt and type “via” in the command prompt to run the *Vulkan Install Analyzer* (via). This will execute couple of graphics application and outputs via.html in the same folder. Open “via.html” to review the hardware, Vulkan driver info, runtime info and other Vulkan specific details. If the Vulkan driver is up to date or if your run into issues running the via application please reinstall the latest graphics driver and try “via” app again.
* There is also an application called “vulkaninfo” which provides more detailed overview of installed Vulkan components.

# Vulkan Programming Model

Before we delve into Vulkan concepts deeper and dirty our hands with programming, let’s see its programming model. This section gives a taste to new OpenGL or DirectX programmers who are curious to know what is looks like programming in Vulkan, how it is different from existing APIs. The below image shows a pictorial representation of Vulkan programming model, let take a look into each section briefly:-

****

**Initialization:** When Vulkan application starts it creates a Vulkan instance at the initialization. Vulkan offers add-on services like validation layers (helpful in debugging) and extensions, these can be queried programmatically and passed into Vulkan instance metadata right before creating it.

Next, find an appropriate device on your system. Vulkan allows querying the existing system offering to choose best device (GPU) as per application requirement. Each device consist of one or more type of queues, if you are coming from OpenGL background this term would appear new to you. A Queue is a bridge/channel/interface between the CPU and GPU. All the work requested from CPU to GPU are sent through the queues in the form of small work units called command buffers. There are various types of Queues – Graphics, Compute and Transfer (for DMA) depending upon the type of GPU it may one or all type in it.

After the choosing a suitable device, choose a suitable queue that support Graphics queue and also has presentation capabilities.

**Presentation Layer:** In OpenGL, the presentation layer is inbuilt and catered through EGL, it provides - context management, surface binding, and rendering synchronization.

In Vulkan, presentation is served through **W**indow **S**ystem **I**ntegration (WSI). Presentation in Vulkan comprises of two steps:

1. **Display Window:** Create a UI display window, this window would contain the window surface in it, into which the final output of drawn primitives from the graphics pipeline will be drawn.
2. **Window Surface Binding:** WSI provide an abstract surface object, bind the above created window with this object. The surface object advertises all its supported formats, capabilities and the number of surface it can support, this information is used to create the swapchain. A swapchain comprises a set of images are used to draw the rendering output.
3. **Frame buffer:** The created swapchain image associated with the special object in Vulkan called Frame buffer object.In Vulkan programing the swapchains are used through frame buffer object. Later we will discover how this frame buffer object are associated with Render pass.

**Resource setup:** Resource setup involves vertex buffer and texture allocation into memory regions. Vulkan is highly flexible in terms of memory management unlike OpenGL, it divides memory into two types –*host(CPU)* and *device(GPU)* memory. These memories are further categories into various heaps for most suitable resource usage type like host local, *device local*, *device local host visible*, *host local host visible*.

Resource setup in Vulkan comprise of the following steps:-

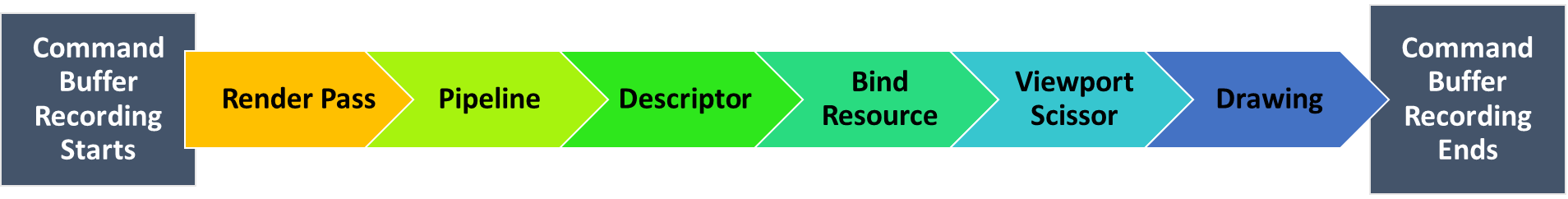
* **Resource creation:** Creates a Vulkan buffer or texture object, at this point there is not physical allocation associated with this object.
* **Gather memory requirements:** Before the physical allocation get the Buffer/Image resource requirements, create memory allocation metadata information and get the compatible type of memory.
* **Allocate and bind:** Finally, allocate the physical backing and copy storage data and bind this memory with the create resource object.

**Pipeline Setup:**During the pipeline setup, shaders module is created, the allocated resource and interfaced with various shader stages like (vertex and fragment). Finally, the graphics pipeline line object are created.

* **Shader module creation:**Unlike OpenGL which expects shader input in the GLSL form, the shaders in Vulkan needs to be converted into SPIR-V format before they are used in Vulkan. Vulkan SDK providesglslangValidator.exetool (<Vulkan\_SDK>/Bin) to convert a GLSL shader into respective format. This different shader are used to create a single shader module object (similar to OpenGL shader program).
* **Descriptors:** Descriptors in Vulkan are interface between resources and shaders.
* **Building pipeline:**Vulkan can create two type of pipelines graphics and compute. A pipeline determines how the submitted input data needs to interpreted and draw. A pipeline states in Vulkan is a collection of settings that controls the hardware. Unlike global states in OpenGL, the states in Vulkan are specific in pipeline object through pipeline states. Some of the example of states are rasterizer state, blend state, and depth-stencil state, specifying primitive topologies for geometry and the shader used in the drawing process.

**Command Recordings:**In Vulkan, rendering of computejobs are submitted from CPU to GPU in the form of command buffers. Command buffers are special objects that records the command or action in it and submitted into the Queues for processing.

Command buffer creation is performance critical path, command buffer can be recorded, cached and reused again and again.



The above image shows the command buffer recording scope, it is effectively be used for drawing some primitive on the frame buffer surface. Let’s go through each step quickly:

* **Render Pass:** indicates what attachments and sub-passes currently involved in this execution process. It also specifies the any dependencies between those sub-passes. For example: an attachment refers to depth buffer for depth testing or color image for performing drawing. In the case of sub-pass, an attachment can follow another route performing some post operations on it for instance image can sub-passed for multisampling resolve. Render pass can controls the frame buffer how it will be begin and end of the render pass. For example, at the beginning of the render pass it needs to be clear or not. Similarly, will it be stored or discarded end of execution process.
* **Bind the pipeline:** Specifies what pipeline will be used, here we will pass the graphics pipeline object that we created in last stage during pipeline setup.
* **Descriptor:** If you using uniforms in your application, then here you need to specify the descriptors. The descriptor contains layout information indicatingof how resource inputs are interfaced within the shader. Remember, the shader information is embedded in the graphics pipeline object
* **Bind Resource:** Specify the allocated geometry or images information.
* **Viewport/Scissor:** Specify the rectangular viewport and scissor region.
* **Drawing:** Drawing command interprets how the geometry buffer is read from geometry data stored in the memory regions.

**Queue Submission:** A command buffer once recorded is ready for processing on the GPU, they need to be submitted into an appropriate queue where they are executed in an asynchronous fashion. It’s an application programmer’s responsibility to recognize the job type associate with the command buffer and chose the correct queue for submission. For example, a drawing related command buffer must be submitted into the graphics queue, whereas the compute command buffer belongs to the compute queue.

Vulkan provides various synchronization mechanism,itcan be performed at command buffer level, between queues or even across device and host,we would learn more about these in upcoming chapters.

# Before We Start

Vulkan- By Example is a quick reference guide for programmer who have some experience on OpenGL or Direct-X want to learn this new graphics API with several real time Vulkan examples. This book start from a beginner level and quickly paces to intermediate and advance level Vulkan concepts. This book builds a project over several intermittent small example in quick successions, with each example cover the flow of the implementation, Vulkan concept and related API used in it. The detail description of Vulkan APIs is out of the scope of this book, if you are looking for in-depth details of Vulkan API and want to learn over many simple step-by-step example, please refer to Learning Vulkan, by Parminder Singh from Packting publishing.

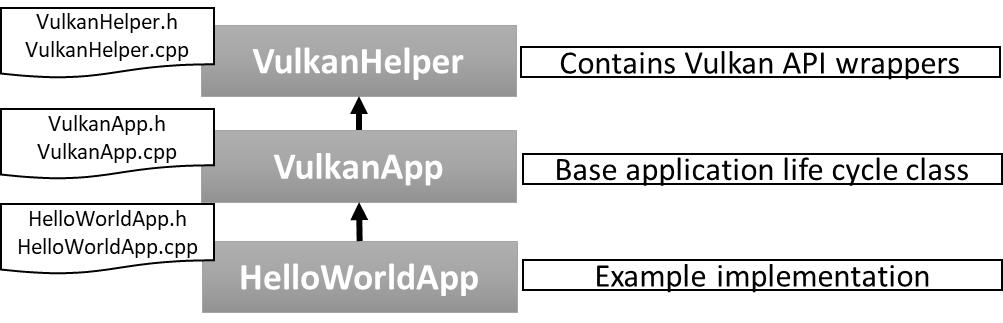
The scope of this book is not to cover the low-level details of the API, it

This book is intended to provide a quick reference to our reader who wants to hands on experience with real time Vulkan examples What we expect from the users, give reference to other book of packt for API. Tell the user about the nature of book – for example this book will be at a high pace and going through each API specification is out the scope of this book.

## Program structure

All the examples in this this book will share the same program structure, there are two basic common classes across:

* **VulkanHelper:** This helper class contains reusable utility wrappers over Vulkan API. It contains wrappers for validation layers, resource like buffer and image, memory, swapchain, command buffer, shaders and many more. This class is the base class of VulkanApp.
* **VulkanApp:** This class defines the application life cycle, it initializes the Vulkan application and runs it. This class provide basic skeleton for a Vulkan application, all the example implementation must be derived from this class.



We will implement these classes from scratch grow it over the time, the following code shows a minimal C++ example using CMake. In the next section, we will learn to use CMake and build our project solution.

**// Vulkan wrappers -VulkanHelper.h / VulkanHelper.cpp**

class VulkanHelper

{

public:

VulkanHelper();

virtual ~VulkanHelper();

};

**// Base Vulkan applicationclass - VulkanApp.h / VulkanApp.cpp**

class VulkanApp : public VulkanHelper

{

public:

VulkanApp();

virtual ~VulkanApp();

**// Initialize the Vulkan application**

void Initialize(){ cout << “Hello from Vulkan!”}

};

**// Example inherits VulkanApp -HelloVulkanApp.h / HelloVulkanApp.cpp**

class HelloVulkanApp : public VulkanApp

{

public:

HelloVulkanApp();

virtual ~HelloVulkanApp();

};

int main(int argc, char \*\*argv)

{

VulkanApp\* helloVulkanApp = new HelloVulkanApp();

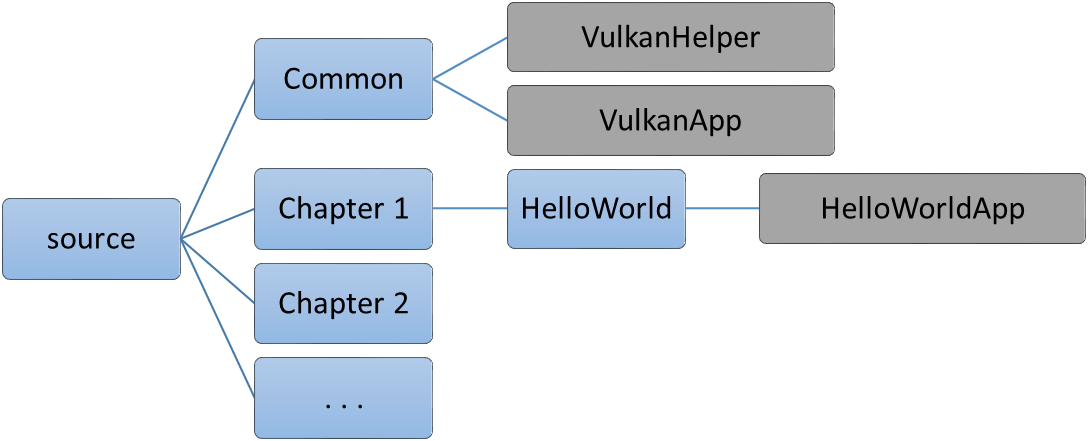
helloVulkanApp->Initialize();

delete helloVulkanApp;

return 0;

}

The below image provides an overview of the folder structure used in the source code of this book. The entire source code is present in the source folder, this folder contain sample from various chapters and a common folder. The common folder contains VulkanHelper and VulkanApp classes as we proceed through this book these classes would grow over the time. Each chapter may contain one or more sample examples in it. For instance, the very first example in this book contains the HelloWorldApp class (derived from *VulkanApp*) in HelloWorld in Chapter 1.



## First handshake with Vulkan

In this section, we will write a small code to query validation layers in your Vulkan system. Validation layers may vary from one system to another, also based on the capabilities of device and Vulkan SDK version. Validation layer are highly important to point out coding error, their correct usage etc., some layers like VK\_LAYER\_LUNARG\_object\_trackeralso indicates memory leaks of the Vulkan API. We will learn more about these validation layers in detail in the upcoming chapters. (Todo: Pls add a reference where user can read layer and externsions.)

The following code introduction a new function GetValidationlayers() in VulkanHelper class prints the name of all validation layer present in existing system. This function is called in HelloVulkanApp class constructor.

class VulkanHelper

{

. . .

VkResult GetValidationLayers();

};

VkResult VulkanHelper::GetValidationLayers()

{

uint32\_t instanceLayerCount;

std::vector<VkLayerProperties>layerProperties;

**// Query all the layers**

VkResult result = vkEnumerateInstanceLayerProperties(

&instanceLayerCount, NULL);

layerProperties.resize(instanceLayerCount);

result = vkEnumerateInstanceLayerProperties(

&instanceLayerCount, layerProperties.data());

if (result) return result;

**// Print all layers**

std::cout <<"\n \*\*\*\*\*Instance Layers \*\*\*\*\*" << std::endl;

std::cout <<"------------------------------" << std::endl;

for (auto globalLayerProp : layerProperties)

{

std::cout << "\n" << globalLayerProp.description <<

"\n\t|---[Layer Name]--> " << globalLayerProp.layerName;

}

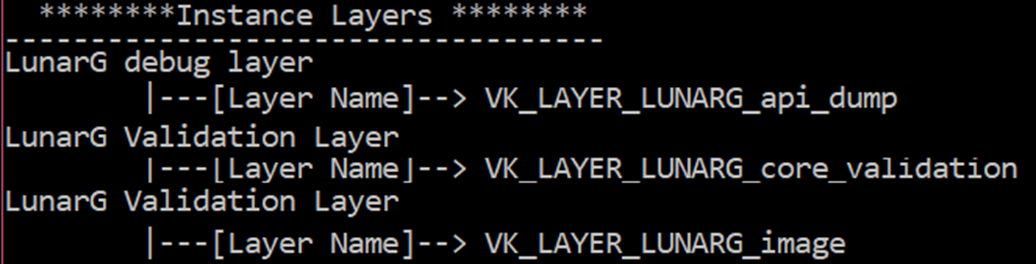
}

Vulkan layers can exist at global level(per instance) and device level(per gpu). This section will only query and print layers at global level, device level layers will be discussed in the (given section name, chanpter name).

The application starts from the main() creating an object of HelloVulkanApp, the constructor of this class calls GetValidationLayers(). This function has two variable instanceLayerCount and layerProperties, the First store total number of layers support on existing system, whereas the second is vector of layers of data structure typeVkLayerProperties.First, the number of layers are retrieved by calling vkEnumerateInstanceLayerProperties Vulkan API. In order to retrieved layer count in instanceLayerCount, the second argument must be as NULL. For queries the layer information the same API is called again but this time the second argument is passed with vector in which we want to retrieve the layer information.

In this code you have noticed that we called a same API twice for two different functionalities. The vkEnumerateInstanceLayerProperties API queries number of layers (by passingNULL argument),also it queries the layer property information (by passing vector of layer properties). This is a very common practice in Vulkan enumerated APIs where passing an API with two different arguments changes its nature of functionality.

Following output is producedwhen the program executes, there are many more layers, however, the output is only for demonstration purpose.



## Using CMake

This book uses CMake build process management tool to build our sample examples in a compiler-independent way.In this section, we set-up the very first CMake configuration fileCMakeLists.txt and build the above sample code.

1. Create an empty CMakeLists.txtin your sample directory(say Hello World).The very first line the in this configuration file specifies the minimum supported version (cmake\_minimum\_required) for CMake compatibility purposes.In CMake user defined variables can be specified using setproperty, for instance you can store the name of your sample application in Example\_Name variable as shown below:

cmake\_minimum\_required(VERSION 3.7.1)

set(Example\_Name "HelloWorld")

1. CMake is also capable of finding installer package, which is awesome. It can auto locate the Vulkan SDK using find\_package CMake property,you must need to passpackage name into it, in our case it must be ‘Vulkan’. If CMake located Vulkan SDK successfully,it provide${VULKAN\_PATH} environmentvariable specifying the complete Vulkan SDK path into it.

In the case, find\_packagefails to locate Vulkan SDK then you can manually provide the Vulkan SDK in the CMakeList.txt. For more information, please refer to accompanying source code of this example.

find\_package(Vulkan)

Next, specify Vulkan system header files path using include\_directoriesproperty and link directories (link\_directories) to locate path for Vulkan libraries.

if(${CMAKE\_SYSTEM\_NAME} MATCHES "Windows")

include\_directories(AFTER ${VULKAN\_PATH}/Include)

link\_directories(${VULKAN\_PATH}/Bin)

link\_directories(${VULKAN\_PATH}/Lib)

endif()

1. CMake can read all the files present in folder using file(GLOB\_RECURSEproperty, we will take the advantage of this property and read ‘source’ folder (contains HelloWorldApp.h /.cpp) and ‘common’ folder(contains VulkanHelper, VulkanApp.h/.cpp). The read file names from the respective folder arestored inuser defined variablesSOURCE\_FILE and COMMON\_FILES.Use these to building project withadd\_executable CMake property.

file(GLOB\_RECURSE SOURCE\_FILES

${CMAKE\_CURRENT\_SOURCE\_DIR}/source/\*.\*)

file(GLOB\_RECURSE COMMON\_FILES

${CMAKE\_CURRENT\_SOURCE\_DIR}/../../common/\*.\*)

add\_executable(${Example\_Name} ${SOURCE\_FILES}

${COMMON\_FILES})

The Vulkan libraries (vulkan-1.dll) can be linked with the target solution by specifying the in the target\_link\_librariescommand.

target\_link\_libraries( ${Recipe\_Name} vulkan-1)

1. The output directory for of the executable can be specified based of the build targetsuch as Debug, Release etc.

set\_property(TARGET ${Recipe\_Name}

PROPERTY RUNTIME\_OUTPUT\_DIRECTORY

${CMAKE\_CURRENT\_SOURCE\_DIR}/binaries)

set\_property(TARGET ${Recipe\_Name}

PROPERTY RUNTIME\_OUTPUT\_DIRECTORY\_DEBUG

${CMAKE\_CURRENT\_SOURCE\_DIR}/binaries)

set\_property(TARGET ${Recipe\_Name}

PROPERTY RUNTIME\_OUTPUT\_DIRECTORY\_RELEASE ${CMAKE\_CURRENT\_SOURCE\_DIR}/binaries)

Specify C/C++ version of your choiceto compile the project using C/CXX\_STANDARD flags in set\_propertyof CMake property. The other flag C/CXX\_STANDARD\_REQUIREDindicate whether the C/C++ standard value are required or not.

set\_property(TARGET ${Recipe\_Name}

PROPERTY CXX\_STANDARD 11)

set\_property(TARGET ${Recipe\_Name}

PROPERTY CXX\_STANDARD\_REQUIRED ON)

set\_property(TARGET ${Recipe\_Name} PROPERTY C\_STANDARD 99)

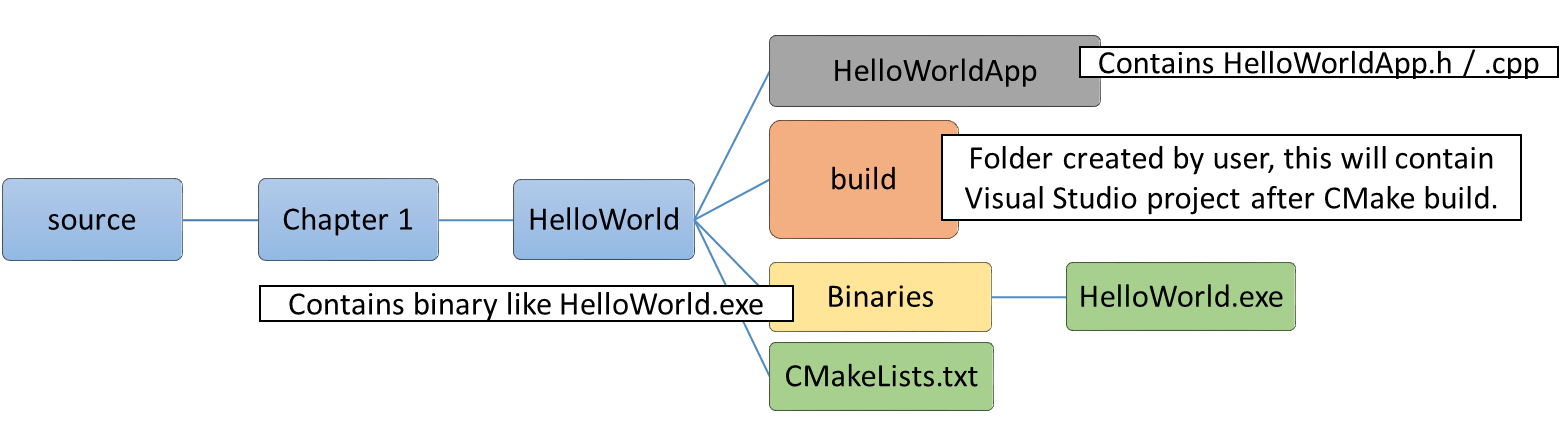
set\_property(TARGET ${Recipe\_Name} PROPERTY

C\_STANDARD\_REQUIRED ON)

**Building target solution:** Goto your sample directory, create an empty build folder next to CMakeLists.txt, this folder will contain the CMake auto-generated visual studio project solution.

Now, open terminal and go this builddirectory and execute the following command on Windows 64bit machine, for 32 bit use Win32 instead.

CMake –G “Visual Studio 14 2015 Win64” ..



Here is the command line interface looks like:



# Hello World Program in Vulkan

In this section, we will render a Hello World! Program in Vulkan, this program will draw a simple triangle on the display window. Here we will understand what it takes to properly initialize a Vulkan application, we implement and learn unique concepts of Vulkan that are very different from traditional API. For example, we will interact with GPU installed on your system through Vulkan instance and query the physical devices and their capabilities. We will also delve with queues which is an interface to communicate with the physical device. Finally, we will close this section with a detail introduction to and layer and extensions understanding there critical role in Vulkan application development process.

## Vulkan Instance

The first step in the Vulkan initialization process is to create a Vulkan instance object (VkInstance). Like a main() function is an entry program of a program and Vulkan Instance is also an entry point for an Vulkan application. It is a highest level of object in a Vulkan application house keeping all essential states in it.

The Vulkan instance (VkInstance) is created in CreateVulkanInstance() using vkCreateInstance() API, it uses a VkInstanceCreateInfo which contains application specific information such as name, version of application and it also indicates what Vulkan version it expects to run upon. If this version of application found incompatible with the underlying driver then validation layer should report an error. The Vulkan instance also help us enabling set of validation layer and extension APIs.

We added a new function in VulkanApp class CreateVulkanInstance()to create Vulkan instance.

class VulkanApp : public VulkanHelper{

**voidCreateVulkanInstance();**

**VkInstance m\_hInstance;**

};

std::vector<const char \*> validationLayers =

{"VK\_LAYER\_LUNARG\_standard\_validation"};

void VulkanApp::CreateVulkanInstance()

{

VkApplicationInfo **appInfo** = {};

appInfo.sType = VK\_STRUCTURE\_TYPE\_APPLICATION\_INFO;

appInfo.pNext = NULL;

appInfo.pApplicationName = m\_appName.c\_str();

appInfo.applicationVersion = VK\_MAKE\_VERSION(1, 0, 0);

appInfo.pEngineName = "VulkanApp";

appInfo.engineVersion = VK\_MAKE\_VERSION(1, 0, 0);

appInfo.apiVersion = VK\_API\_VERSION\_1\_0;

VkInstanceCreateInfo **createInfo** = {};

createInfo.sType = **VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_INFO**;

createInfo.pNext = nullptr;

createInfo.flags = 0;

createInfo.pApplicationInfo= &**appInfo**;

createInfo.enabledLayerCount = validationLayers.size();

createInfo.ppEnabledLayerNames = validationLayers.data();

createInfo.enabledExtensionCount =0;

createInfo.ppEnabledExtensionNames= nullptr;

// Create the Vulkan Instance

**vkCreateInstance**(&**createInfo**, nullptr, &**m\_hInstance**);

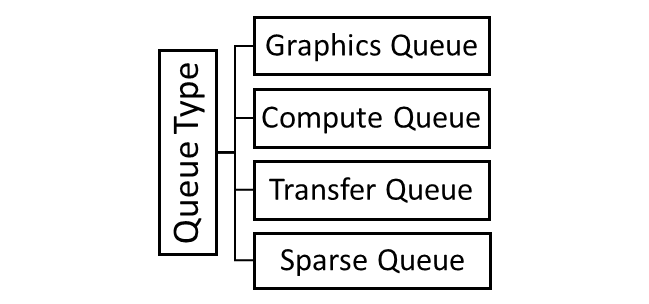
}

## Device and Queues

Vulkan API allows an application to peek into the system exposing GPU devices and associated properties. Vulkan categories devices in two types – *Physical device* and *Logical device*.

* **Physical device (**VkPhysicalDevice**):** This refers to an actual GPU hardware units on your system, the number of physical device are fixed.
* **Logical device (**VkDevice**):** A logical device is a software construct and provide a logical view of the actual device. The number of logical devices is in the hand of the application, it can create as many views of the actual device and use it for intended purposes.

Each physical device may comprises of 4types of queue. – Graphics, Compute, Transfer and Sparse. Further, these queues belongs to some queue family.



The second step in the initialization process is to create the Vulkan device and queues. This is a multi-step process.

1. First we need to find the appropriate physical device (GPU) that supports Vulkan in the system
2. Then create the handle to Vulkan device
3. Then get the handle to graphics queue and present queue that are required to submit work to the GPU.

A physical device comprises of various properties that would be required for application programming, we will find a suitable physical device and stored it properties in a custom structure PhysicalDeviceInfo which is defined in VulkanHelper class.

**// A structure to store physical device information**

struct PhysicalDeviceInfo**// Defined in VulkanHelper**

{

VkPhysicalDeviceProperties prop = {};

VkPhysicalDeviceFeatures features = {};

VkPhysicalDeviceMemoryProperties memProp = {};

std::vector<VkQueueFamilyProperties> familyPropList;

intgraphicsFamilyIndex = -1;

int presentFamilyIndex = -1;

std::vector<VkExtensionProperties> extensionList;

VkSurfaceCapabilitiesKHR capabilities = {};

std::vector<VkSurfaceFormatKHR> formatList;

std::vector<VkPresentModeKHR> presentModeList;

};

The below code shows the new member variables and functions added to VulkanApp class in order to implement for Devices and Queues.

class VulkanApp : public VulkanHelper

{

. . .

private:

voidCreateNativePlatformSurface();

void CreateVulkanDeviceAndQueues();

**// Helper functions for CreateVulkanDeviceAndQueues()**

void SelectPhysicalDevice();

void GetPhysicalDeviceInfo(VkPhysicalDevice device,

PhysicalDeviceInfo\* pDeviceInfo);

bool IsDeviceSuitable(PhysicalDeviceInfo deviceInfo);

protected:

VkSurfaceKHR m\_hSurface; **// Vulkan native surface**

VkPhysicalDevicem\_hPhysicalDevice;**// Physical device**

PhysicalDeviceInfo m\_physicalDeviceInfo;**// Device attributes**

VkDevice m\_hDevice; **// Logical device**

VkQueue m\_hGraphicsQueue, m\_hPresentQueue;**//Queue**

};

The device and queues are created in CreateVulkanDeviceAndQueues() function.

* The SelectPhysicalDevice() will query all physical devices that are available on the system and choose the one that closely matches the application requirement. For instance, implementing a graphics application requires a graphics queue and presentation capabilities from the physical device. The properties of best selected physical device is stored locally for later use.
* Once a suitable physical device is created, we can create a logical device (VkDevice) and retrieve the Graphics and presentation queue in CreateDeviceAndQueueObjects() function.

void VulkanApp::CreateVulkanDeviceAndQueue()

{

**// Select the physical suitable for Vulkan**

SelectPhysicalDevice();

**// Create the Vulkan device and graphics & present queue**

CreateDeviceAndQueueObjects();

}

The SelectPhysicalDevice()selects the best physical device:-

1. First it enumerates all the available connected physical devices on the system using vkEnumeratePhysicalDevices().
2. For each physical device its properties and capabilities are queried using GetPhysicalDeviceInfo() and stored in PhysicalDeviceInfo structure.
3. Finally, the IsDeviceSuitable() function determines if the device’s property matches to our application need. Let’s look into these function in detail.

void VulkanApp::SelectPhysicalDevice()

{

uint32\_t deviceCount = 0;

vkEnumeratePhysicalDevices(m\_hInstance, &deviceCount, nullptr);

if (deviceCount == 0) return;

std::vector<VkPhysicalDevice>deviceList(deviceCount);

vkEnumeratePhysicalDevices(m\_hInstance,

&deviceCount, deviceList.data());

for (uint32\_t i = 0; i<deviceCount; i++)

{

PhysicalDeviceInfo deviceInfo = {};

GetPhysicalDeviceInfo(deviceList[i], &deviceInfo);

if (IsDeviceSuitable(deviceInfo)){

m\_hPhysicalDevice = deviceList[i];

m\_physicalDeviceInfo = deviceInfo;

return;

}

}

}

### Retrieving Device Properties

Each physical device get device details using the following APIs in the GetPhysicalDeviceInfo() function:

1. vkGetPhysicalDeviceProperties() returns properties of the device, such as device name, device id, vendor id, device limits etc.
2. vkGetPhysicalDeviceFeatures() returns the capabilities of the device such as shader capabilities, rendering capabilities etc.
3. vkGetPhysicalDeviceMemoryFeatures() API exposes one or more heaps and further exposes one or more memory types from these heaps.

void VulkanApp::GetPhysicalDeviceInfo(VkPhysicalDevice device, PhysicalDeviceInfo\* pDeviceInfo)

{

**// Query device properties, features and memory properties**

vkGetPhysicalDeviceProperties(device, &pDeviceInfo->prop);

vkGetPhysicalDeviceFeatures(device, &pDeviceInfo->features);

vkGetPhysicalDeviceMemoryProperties(device,

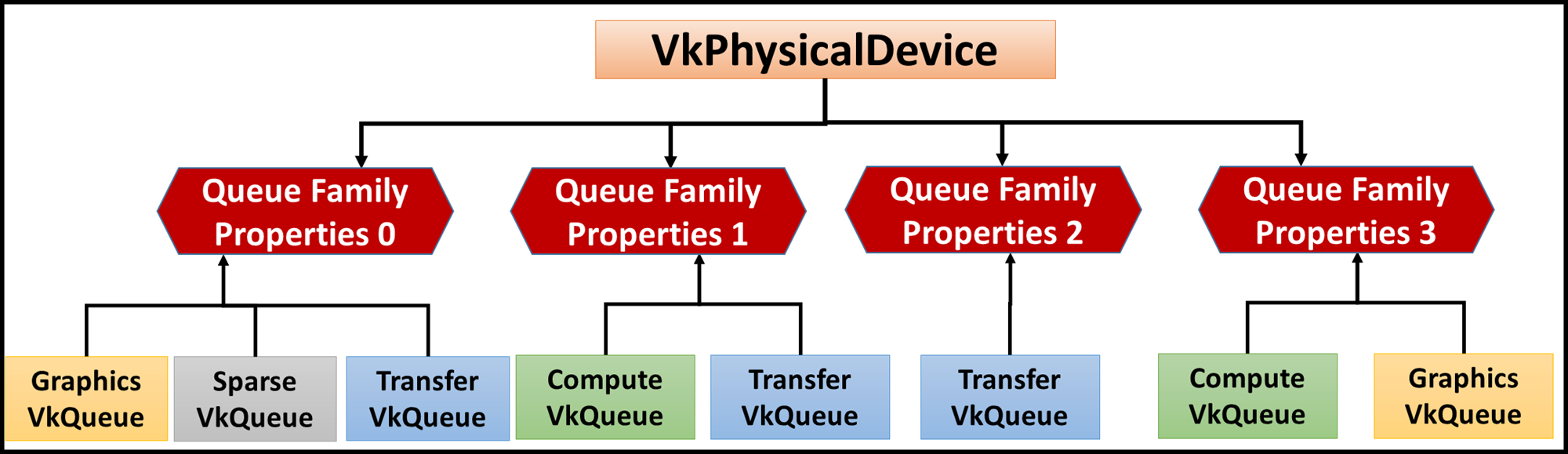
&pDeviceInfo->memProp);

. . . . .

}

1. vkGetPhysicalDeviceQueueFamilyProperties() returns the properties of the queue family this physical device supports. Application can submit workload to the device using the queue. A queue family is collection of queues that have identical capabilities. A device can support multiple queue families, where each queue family can be a combination of graphics, compute, transfer & sparse memory management.

A queue family is collection of queues that have identical capabilities, a physical device may have of one or more queue families.



The below code finds a Queue Family that supports Graphics queue and also have the presentation capabilities.

**// VulkanApp::GetPhysicalDeviceInfo continued . . .**

**// Get the list of queue families support on this physical device**

uint32\_t queueFamilyCount = 0;

vkGetPhysicalDeviceQueueFamilyProperties(device,

&queueFamilyCount, nullptr);

pDeviceInfo->familyPropList.resize(queueFamilyCount);

vkGetPhysicalDeviceQueueFamilyProperties(device,

&queueFamilyCount, pDeviceInfo->familyPropList.data());

**// Find graphics and presentation queue and store its index**

pDeviceInfo->graphicsFamilyIndex = -1;

pDeviceInfo->presentFamilyIndex = -1;

for (int i = 0; i<pDeviceInfo->familyPropList.size(); i++){

if (pDeviceInfo->familyPropList[i].queueCount > 0){

if (pDeviceInfo->familyPropList[i].queueFlags &

VK\_QUEUE\_GRAPHICS\_BIT)

{ pDeviceInfo->graphicsFamilyIndex = i; }

VkBool32 presentSupport = false;

vkGetPhysicalDeviceSurfaceSupportKHR(device, i,

m\_hSurface, &presentSupport);

if (presentSupport){pDeviceInfo->presentFamilyIndex = i; }

if (pDeviceInfo->graphicsFamilyIndex >= 0 &&

pDeviceInfo->presentFamilyIndex >= 0){

break;

}

}

1. Next check what all extension are supported in the for this physical device using GetPhysicalDeviceExtensions() this function inside uses vkEnumerateDeviceExtensionProperties() API to retrieve the list of extension supported by this device.

### Querying surface capabilities

For presentation, its very important to know what surface capability the physical device supports this includes:

1. Querying all surface formats are supported by the physical device.
2. We also want to know what present modes are offered by this device. The presentation modes that can be retrieved using the vkGetPhysicalDevice-SurfacePresentModesKHR() API extension.

The surface capabilities information will be later used in the next section swapchain to produce presentation surfaces with compatible surface formats.

**// VulkanApp::GetPhysicalDeviceInfo continued . . .**

**// Get the list of extension supported by physical device**

GetPhysicalDeviceExtensions(device, pDeviceInfo->extensionList);

vkGetPhysicalDeviceSurfaceCapabilitiesKHR(device,

m\_hSurface, &pDeviceInfo->capabilities);

**// Get the list of supported surface format from the device**

uint32\_t formatCount;

vkGetPhysicalDeviceSurfaceFormatsKHR(device,

m\_hSurface, &formatCount, nullptr);

if (formatCount != 0){

pDeviceInfo->formatList.resize(formatCount);

vkGetPhysicalDeviceSurfaceFormatsKHR(device,

m\_hSurface, &formatCount, pDeviceInfo->formatList.data());

}

**// Get the list of Supported Presentation Modes from the device**

uint32\_t presentModeCount;

vkGetPhysicalDeviceSurfacePresentModesKHR(device,

m\_hSurface, &presentModeCount, nullptr);

if (presentModeCount != 0){

pDeviceInfo->presentModeList.resize(presentModeCount);

vkGetPhysicalDeviceSurfacePresentModesKHR(device, m\_hSurface,

&presentModeCount, pDeviceInfo->presentModeList.data());

}

}

### Selecting the Best Physical Device

From the retrieved physical device information, we will choose a physical device that fits application requirements using IsDeviceSuitable(). The below are criteria use to match the best fit:

1. The physical device must support queues that are capable graphics and presentation capabilities.
2. The device must support the desire extensions for swap chain support. For instance, we supplied the list of extension that application interested in like VK\_KHR\_SWAPCHAIN\_EXTENSION\_NAME, int the deviceExtensionNames vector(from HelloVulkanApp.cpp).

std::vector<const char \*> deviceExtensionNames =

{VK\_KHR\_SWAPCHAIN\_EXTENSION\_NAME};

bool VulkanApp::IsDeviceSuitable(PhysicalDeviceInfo deviceInfo)

{

bool result = false;

if (deviceInfo.graphicsFamilyIndex >= 0 &&

deviceInfo.presentFamilyIndex >= 0)

{

extern std::vector<const char \*> deviceExtensionNames;

std::set<std::string> requiredExtensions

(deviceExtensionNames.begin(), deviceExtensionNames.end());

for (uint32\_t i = 0; i < deviceInfo.extensionList.size()

&& !result; i++){

requiredExtensions.erase

(deviceInfo.extensionList[i].extensionName);

result = requiredExtensions.empty();

}

if (result)

{ result = (!deviceInfo.formatList.empty()) &&

(!deviceInfo.presentModeList.empty()); }

}

return (result);

}

### Creating Device and Queues objects

After selecting the suitable physical device, the logical device (VkDevice) is created with using vkCreateDevice()API. This API intake queues related information like the intended queues it must connect to, extensions to be enabled on this device. The queues are automatically created with logical device behind the curtains using. A queue object can retrieved using vkGetDeviceQueue() APIs passing queue index into it.

The following function CreateVulkanDeviceAndQueues()code the exact process we defined above to create the logical device and queue objects.

void VulkanApp::CreateVulkanDeviceAndQueues()

{

std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;

std::set<int> uniqueQueueFamilies =

{ m\_physicalDeviceInfo.graphicsFamilyIndex,

m\_physicalDeviceInfo.presentFamilyIndex };

float queuePriority[] = { 1.0f };

for (std::set<int>::iterator it = uniqueQueueFamilies.begin();

it != uniqueQueueFamilies.end(); ++it)

{

VkDeviceQueueCreateInfo queueCreateInfo = {};

queueCreateInfo.sType =

VK\_STRUCTURE\_TYPE\_DEVICE\_QUEUE\_CREATE\_INFO;

queueCreateInfo.queueFamilyIndex = \*it;

queueCreateInfo.queueCount = 1;

queueCreateInfo.pQueuePriorities = queuePriority;

queueCreateInfos.push\_back(queueCreateInfo);

}

**// Now fill-in the VkDeviceCreateInfo structure**

VkPhysicalDeviceFeatures deviceFeatures = {};

VkDeviceCreateInfo createInfo = {};

createInfo.sType = VK\_STRUCTURE\_TYPE\_DEVICE\_CREATE\_INFO;

createInfo.queueCreateInfoCount =

static\_cast<uint32\_t>(queueCreateInfos.size());

createInfo.pQueueCreateInfos = queueCreateInfos.data();

createInfo.pEnabledFeatures = &deviceFeatures;

createInfo.enabledExtensionCount =

static\_cast<uint32\_t>(m\_requiredDeviceExtensionList.size());

createInfo.ppEnabledExtensionNames =

m\_requiredDeviceExtensionList.data();

createInfo.enabledLayerCount = 0;

**// Create the Vulkan logical device**

if (vkCreateDevice(m\_hPhysicalDevice, &createInfo,

nullptr, &m\_hDevice) == VK\_SUCCESS)

{

**// Retrieve the graphics queue and present queue**

vkGetDeviceQueue(m\_hDevice, m\_physicalDeviceInfo.

graphicsFamilyIndex, 0, &m\_hGraphicsQueue);

vkGetDeviceQueue(m\_hDevice, m\_physicalDeviceInfo.

presentFamilyIndex, 0, &m\_hPresentQueue);

}

}

## Presentation Layer

In this section, we will implement initialization of presentation layer. Here, we will create a display window. The handle of this display window is used to create a native window surface object. This surface object is then used by the swapchain to create the a set of image that will be used by drawing and presenting on the window screen.

### Creating Display Window Surface

In this section, we will create the simple Qt display window, this window will be used to display the drawing content from the graphics pipeline. We need to append the following code in the CMakeList.txt in order to bring the Qt support in our target project.

include(QtCommon)

fix\_project\_version()

add\_project\_meta(META\_FILES\_TO\_INCLUDE)

find\_package(Qt5Widgets REQUIRED)

qt5\_use\_modules(${PROJECT\_NAME} Widgets)

The following Window class is implemented in VulkanApp.h and provide a custom native window in which the graphics output will appear on the screen. This class implements QWindow which represents a window in the underlying windowing system.

#include <QWindow>

class Window : public QWindow

{

Q\_OBJECT

public:

Window(VulkanApp\* vulkanApp) : m\_VulkanApp(vulkanApp)

{

assert(vulkanApp);

setWidth(800);

setHeight(600);

}

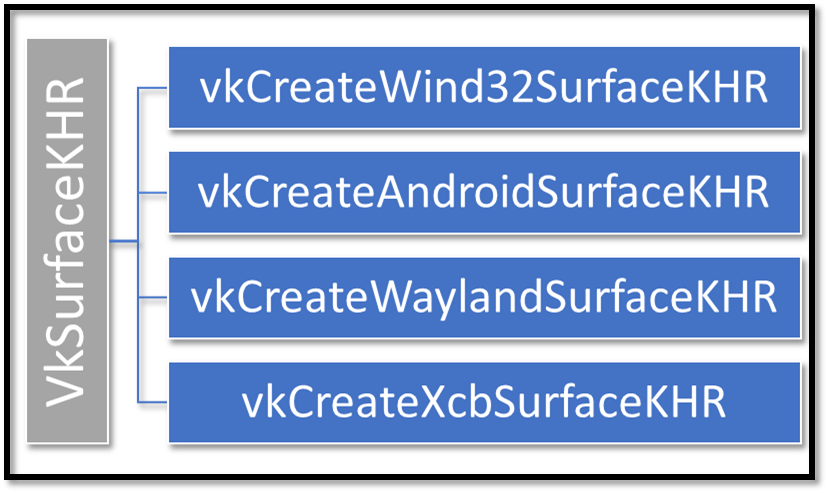
VulkanApp\* m\_VulkanApp;

};

### Creating Native Window Surface

As you must be knowing EGL is used in OpenGL/ES programs to interface with the native windowing platform system which may vary from one platform to another. In Vulkan the Native Windowing support is managed through *Window System Integration* (WSI) which standardize the native window creation for various platforms.

The native window is created in higher level surface abstraction object called VkSurfaceKHR. The native window is created through a platform specific extension whose signatures are of form vkCreate<Platform>SurfaceKHR. The following image shows these APIs for various system.



In this section, we will implement the native surface in CreateWindowAndNativeSurface() of VulkanApp class. The vkCreateWin32SurfaceKHR()API take a create info structure, the most important parameter of this structure is the hwnd, which is the handle of the native window that we create in previous section.

void VulkanApp::CreateWindowAndNativeSurface()

{

**m\_pWindow = new Window(this);// Display window**

m\_pWindow->show();

VkResult result;

#ifdef \_WIN32

**// Native surface**

VkWin32SurfaceCreateInfoKHR createInfo = {};

createInfo.sType = VK\_STRUCTURE\_TYPE\_WIN32\_

-SURFACE\_CREATE\_INFO\_KHR;

createInfo.pNext = NULL;

createInfo.hinstance = GetModuleHandle(nullptr);

createInfo.hwnd = **m\_pWindow->winId();**

result = **vkCreateWin32SurfaceKHR**(m\_hInstance,

&createInfo, NULL, &**m\_hSurface**);

#endif

}

### Swap chain

Swap chains are collections of images that are used to render and present the content on the screen. In this section, we will learn the step-by-step procedure to create the swap chain surface images, these are consumed by the graphics pipeline to paint the drawing primitives and later this surface image used by presentation engine to swap on the screen’s display window.

During initialization we cached swap chain support properties and surface capabilities from our physical device information in the PhysicalDeviceInfoobject m\_physicalDeviceInfo in GetPhysicalDeviceInfo() function. In this section, we will use this information thoroughly to build the swap chain.

The swap chain is implemented in the CreateSwapChain() function of VulkanApp class. Following are the new member variables and function added for the swapchain support.

class VulkanApp : public VulkanHelper{

**void CreateSwapChain();**// Function to create swapchain

// Swap chain specific member variable

**VkSwapchainKHR m\_hSwapChain;**

**VkFormat m\_hSwapChainImageFormat;**

**VkExtent2D m\_swapChainExtent;**

**std::vector<VkImageView> m\_hSwapChainImageViewList;**

};

Below are the step-by-step process to implement the swapchain in Vulkan.

1. **Select the best surface format:** Iterate through all the physical device formatList and choose the one with surface format VK\_FORMAT\_B8G8R8A8\_UNORMand having color space withVK\_COLOR\_SPACE\_SRGB\_NONLINEAR\_KHR.

The surface format and colorspace is very specific to your own application requirement. You can choose it as per your choice and see if the physical device surface format is compatible against the requested properties.

**// Pseudocode VulkanHelper::SelectBestSurfaceFormat**

foreach (m\_physicalDeviceInfo.formatList){

if (m\_physicalDeviceInfo.formatList.format ==

VK\_FORMAT\_B8G8R8A8\_UNORM &&

m\_physicalDeviceInfo.formatList.colorSpace ==

VK\_COLOR\_SPACE\_SRGB\_NONLINEAR\_KHR){

bestFormat found;

break;

}

}

1. **Select the best presentation mode:** When a present request comes to the presentation engine takes intended swapchain image from the graphics pipeline and present to the user. There are different schemes in which the synchronization between window system and present rate(rate at which images are displayed)can be controlled these are called presentation modes. These schemes determine how incoming present requests will be processed and queued internally. The VkPresentModeKHR supports four types of presentation modes:

* VK\_PRESENT\_MODE\_IMMEDIATE\_KHR: This mode displays the present request as soon as possible it is triggered, it does not wait for the vertical blank signal making the image highly susceptible to the tearing.
* VK\_PRESENT\_MODE\_MAILBOX\_KHR: In this mode, when the present request is made it is pushed into a single entry queue, any other entry into this queue will queued after the older one. In case, the queue is full the latest presentation request replaces the previous one. The present requests added into this queue are not immediately displayed but deferred until next vertical blanking event is signaled. This scheme ensures only the latest present request is displayed to the user and guarantees no tearing effect.
* VK\_PRESENT\_MODE\_FIFO\_KHR: In this presentation mode, the present request are queued into a single entry queue and fetched at the regular vertical blanking interval from the front of the queue.This mode does not discard any presentation request and display them in order without causing any tearing effect.
* VK\_PRESENT\_MODE\_FIFO\_RELAXED\_KHR:This mode operates in the same manner as VK\_PRESENT\_MODE\_FIFO\_KHR but handle a special case where at the time of vertical blank event, the present request queue may goes empty, in thatevent the scheme fall backs to VK\_PRESENT\_MODE\_IMMEDIATE\_KHR wherethe next present request is displayed immediately irrespective of next vertical blank wait.

The SelectBestPresentMode() function from VulkanHelperclass aids the swapchain to find the best support presentation mode. In our application we gave VK\_PRESENT\_MODE\_MAILBOX\_KHR a higher priority over VK\_PRESENT\_MODE\_IMMEDIATE\_KHR as it is not susceptible to tearing effects.

1. **Get swap chain images dimensions:**Retrieve the correct dimension of the swap chain image from VkSurfaceCapabilitiesKHR’s currntExtent field using VulkanHelper’s SelectBestExtent() function.

For more information onSelectBestSurfaceFormat(), SelectBest-PresentMode(), SelectBestExtent() please refer to the accompanying source of this chapter. <Provide a link>.

1. **Get number of swap chain images:** The capabilities also contain the minimum and maximum number of swapchain surfaces it can support.
2. **Creating the swap chain:** In addition to following the above steps(1 to 4) in the CreateSwapChain(), this function also creates the swapchain object (m\_hSwapChain) using vkCreateSwapchainKHR().When swapchain object is created it silently creates the swap images(of type VkImage) behind the curtains. The swapchain image are retrieved using vkGetSwapchainImagesKHR () API and stored inswapChainImageList (vector of VkImage).

void VulkanApp::CreateSwapChain()

{

VkSurfaceFormatKHR surfaceFormat=SelectBestSurfaceFormat();

VkPresentModeKHR presentMode = SelectBestPresentMode(..);

VkExtent2D extent = SelectBestExtent(..);

uint32\_t imageCount = capabilities minImageCount + 1;

VkSwapchainCreateInfoKHR createInfo = {};

createInfo.sType = VK\_STRUCTURE\_TYPE-

\_SWAPCHAIN\_CREATE\_INFO\_KHR;

createInfo.surface = m\_hSurface;

createInfo.minImageCount = imageCount;

createInfo.imageFormat = surfaceFormat.format;

createInfo.imageColorSpace = surfaceFormat.colorSpace;

createInfo.imageExtent = extent;

createInfo.imageArrayLayers = 1;

createInfo.imageUsage = VK\_IMAGE\_USAGE\_-

COLOR\_ATTACHMENT\_BIT;

createInfo.imageSharingMode = VK\_SHARING\_MODE\_EXCLUSIVE;

createInfo.presentMode = presentMode;

createInfo.clipped = VK\_TRUE;

**// Create the Swap chain**

if (vkCreateSwapchainKHR(m\_hDevice, &createInfo,

nullptr, &m\_hSwapChain) != VK\_SUCCESS){

LogError("vkCreateSwapchainKHR() failed!");

assert(false);

}

**// Get the count of swap chain images**

vkGetSwapchainImagesKHR(m\_hDevice,

m\_hSwapChain, &imageCount, nullptr);

std::vector<VkImage> swapChainImageList;

swapChainImageList.resize(imageCount);

**// Get the swap chain images**

vkGetSwapchainImagesKHR(m\_hDevice, m\_hSwapChain,

&imageCount, swapChainImageList.data());

}

1. **Creating image views:** The swap chain images (VkImage) object cannot be directly used in the Vulkan application instead they are used in the form of image views (VkImageView) and hence finally converted into image view form. Let’s walk through the below code.

for (size\_t i = 0; i < swapChainImageList.size(); i++)

{

**// Fill in VkImageViewCreateInfo**

VkImageViewCreateInfo createInfo = {};

createInfo.sType = VK\_STRUCTURE\_TYPE\_IMAGE-

\_VIEW\_CREATE\_INFO;

createInfo.image = swapChainImageList[i];

createInfo.viewType = VK\_IMAGE\_VIEW\_TYPE\_2D;

createInfo.format = m\_hSwapChainImageFormat;

createInfo.components = VkComponentMapping{};

createInfo.subresourceRange.aspectMask =

VK\_IMAGE\_ASPECT\_COLOR\_BIT;

createInfo.subresourceRange.baseMipLevel = 0;

createInfo.subresourceRange.levelCount = 1;

createInfo.subresourceRange.baseArrayLayer = 0;

createInfo.subresourceRange.layerCount = 1;

**// Create image view(VkImageView) from image(VkImage)**

if (vkCreateImageView(m\_hDevice, &createInfo, nullptr,

&m\_hSwapChainImageViewList[i]) != VK\_SUCCESS)

{

LogError("vkCreateImageView() failed!");

result = false;

}

}

## Building the Graphics pipeline

In this section, we will create the

### Render Pass

A Render pass comprises of *attachments* and *sub-passes* and dependencies between the those sub-passes. It defines an execution process for underlying driver indicating how to treat attachment and sub-passes. Usually, a graphics scene comprises of many passes before the finally rendered or also image posting processing to apply various filters is a good example of passes. In Vulkan, all such passes are defined using the render pass.

A render pass creates a flow in which it indicate the number of sub-passes and there order of execution. Also, what attachments will be used during those subpasses, what to do with the attachment at the beginning or end of subpass. For example, you may like to clear the surface of an attachment in the beginning of the render pass, similarly for post processing process you may like to store results for the next sub-pass to follow-up.

In this example, we will create a very simple render pass using CreateRenderPass() with single sub-pass and attachment. In the upcoming chapters we will learn how to create multiple passes <specific chapter name later here>. The below implementation has single sub-pass with one color attachment which is - one of our swapchain image with the same format. At the render pass begin the attachment is cleared, the render pass will ensures that the final layout of the attachment must be same as required by the presentation.

void VulkanApp::CreateRenderPass()

{

bool result = true;

**// Fill in the color attachment**

VkAttachmentDescription colorAttachment = {};

colorAttachment.format = m\_hSwapChainImageFormat;

colorAttachment.samples = VK\_SAMPLE\_COUNT\_1\_BIT;

colorAttachment.loadOp = VK\_ATTACHMENT\_LOAD\_OP\_CLEAR;

colorAttachment.storeOp = VK\_ATTACHMENT\_STORE\_OP\_STORE;

colorAttachment.stencilLoadOp = VK\_ATTACHMENT\_LOAD\_OP\_DONT\_CARE;

colorAttachment.stencilStoreOp = VK\_ATTACHMENT\_STORE\_OP\_DONT\_CARE;

colorAttachment.initialLayout = VK\_IMAGE\_LAYOUT\_UNDEFINED;

colorAttachment.finalLayout = VK\_IMAGE\_LAYOUT\_PRESENT\_SRC\_KHR;

**// Fill in the color attachment reference**

VkAttachmentReference colorAttachmentRef = {};

colorAttachmentRef.attachment = 0;

colorAttachmentRef.layout = VK\_IMAGE\_LAYOUT\_COLOR\_ATTACHMENT\_OPTIMAL;

**// Fill in the sub pass**

VkSubpassDescription subpass = {};

subpass.pipelineBindPoint = VK\_PIPELINE\_BIND\_POINT\_GRAPHICS;

subpass.colorAttachmentCount = 1;

subpass.pColorAttachments = &colorAttachmentRef;

**// Fill in the sub pass dependency**

VkSubpassDependency dependency = {};

dependency.srcSubpass = VK\_SUBPASS\_EXTERNAL;

dependency.dstSubpass = 0;

dependency.srcStageMask = VK\_PIPELINE\_STAGE\_COLOR\_ATTACHMENT\_OUTPUT\_BIT;

dependency.srcAccessMask = 0;

dependency.dstStageMask = VK\_PIPELINE\_STAGE\_COLOR\_ATTACHMENT\_OUTPUT\_BIT;

dependency.dstAccessMask = VK\_ACCESS\_COLOR\_ATTACHMENT\_READ\_BIT | VK\_ACCESS\_COLOR\_ATTACHMENT\_WRITE\_BIT;

**// Now fill in the render pass info with all the above details**

VkRenderPassCreateInfo renderPassInfo = {};

renderPassInfo.sType = VK\_STRUCTURE\_TYPE\_RENDER\_PASS\_CREATE\_INFO;

renderPassInfo.attachmentCount = 1;

renderPassInfo.pAttachments = &colorAttachment;

renderPassInfo.subpassCount = 1;

renderPassInfo.pSubpasses = &subpass;

renderPassInfo.dependencyCount = 1;

renderPassInfo.pDependencies = &dependency;

**// Create the render pass**

if (vkCreateRenderPass(m\_hDevice, &renderPassInfo, nullptr,

&m\_hRenderPass) != VK\_SUCCESS){

LogError("vkCreateRenderPass() failed!"); assert(false);

}

}

### Graphics Pipeline

A graphics pipeline object informs Vulkan about the entire graphics pipeline setup to draw 3D objects. This includes setting up the fixed function units & programmable shader units in the hardware. The graphics pipeline object is created using vkCreateGraphicsPipelines(), this uses VkGraphicsPipelineCreateInfo structure requires the following pipeline stages to be initialize

* **Shader stages**: In our example, we need a vertex shader and a fragment shader to render the triangle.
* **Vertex Input state:** In our first example we will not be using a vertex buffers hence this state (VkPipelineVertexInputStateCreateInfo) is initialized with zero count. This will be covered in the next upcoming section<Section name> in detail.
* **Input Assembly:** We need to initialize the primitive topology. We will be using VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_STRIP to render our triangle.
* **Viewport:** The view port (VkViewport) is set to the maximum rectangular bound of the swap chain dimension.
* **Scissor Rect:** The scissor rect is also set to the maximum rectangular bound of the swap chain dimension.
* **Rasterizer State:** The rasterizer state such as polygon fill mode, culling mode, front face etc are set in this section using VkPipelineRasterizationStateCreateInfo structure.
* **Multi-Sampling:** Multi-sampling is disabled for our first example. We will be covering more about multi-sampling in the following chapters.
* **Color output masks:** The color output mask is set to output all 4 channels (Red, Green, Blue, Alpha) using VkPipelineColorBlendAttachmentState structure.
* **Color Blending:** Color blending details are specified using VkPipelineColorBlendStateCreateInfo. Blending is disabled in our first example.

We will be using a simple vertex shader and fragment shader to render our first triangle. Store the following vertex shader code to Triangle.vert file. The shader uses fixed position and color values for each of the 3 vertex. In our example we will be calling vkCmdDraw() function to draw 1 triangle using 3 vertex inputs. The gl\_VertexIndex denotes the index of the active vertex shader invocation and the value ranges from 0 to 2. The shader output the vertex position and vertex color based on the gl\_VertexIndex value.

**// Filename: Triangle.vert**

#version 450

#extension GL\_ARB\_separate\_shader\_objects : enable

out gl\_PerVertex { vec4 gl\_Position; };

layout(location = 0) out vec3 fragColor;

vec2 positions[3] = vec2[]

(

vec2(-1.0, 1.0), // Bottom left

vec2(0.0, -1.0), // Center top

vec2(1.0, 1.0) // Bottom Right

);

vec3 colors[3] = vec3[]

(

vec3(1.0, 0.0, 0.0), // Red

vec3(0.0, 1.0, 0.0), // Green

vec3(1.0, 1.0, 0.0) // Yellow

);

void main()

{

**// gl\_VertexIndex contains the zero based vertex invocation index**

gl\_Position = vec4(positions[gl\_VertexIndex], 0.0, 1.0f);

fragColor = colors[gl\_VertexIndex];

}

Similarly, store the following fragment shader code to Triangle.frag file. Once the triangles are rasterized the following pixel shader is executed for each of the interpolated samples in the triangles. Here we have a pass-through shader that just outputs the interpolated input color as output color.

**// Filename: Triangle.frag**

#version 450

#extension GL\_ARB\_separate\_shader\_objects : enable

layout(location = 0) in vec3 fragColor;

layout(location = 0) out vec4 outColor;

void main()

{

// Pass through fragment color input as output

outColor = vec4(fragColor, 1.0);

}

Now we need to compile these shaders into SPIR-V bytecode to pass as input to the graphics pipeline object. Use the glslangValidator.exe to compile as shown below.

C:/VulkanSDK/<version no>/Bin32/glslangValidator.exe -V Triangle.vert –o TriangleVert.spv

C:/VulkanSDK/<version no>/Bin32/glslangValidator.exe -V Triangle.frag –o TriangleFrag.spv

Executing the above commands generate 2 new files TriangleVert.spv and TriangleFrag.spv. The following function create the graphics pipeline object using the shaders and fixed function setups. vkCreatePipelineLayout() is used to create the pipeline layout followed by vkCreateGraphicsPipelines() to create the graphics pipeline object.

bool VulkanApp::CreateGraphicsPipeline()

{

bool result = true;

// Compile the vertex shader

VkShaderModule vertShader = CreateShader("QuadVert.spv");

// Setup the vertex shader stage create info structures

VkPipelineShaderStageCreateInfo vertShaderStageInfo = {};

vertShaderStageInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_SHADER\_STAGE\_CREATE\_INFO;

vertShaderStageInfo.stage = VK\_SHADER\_STAGE\_VERTEX\_BIT;

vertShaderStageInfo.module = vertShader;

vertShaderStageInfo.pName = "main";

**// Compile the fragment shader**

VkShaderModule fragShader = CreateShader("QuadFrag.spv");

**// Setup the fragment shader stage create info structures**

VkPipelineShaderStageCreateInfo fragShaderStageInfo = {};

fragShaderStageInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_SHADER\_STAGE\_CREATE\_INFO;

fragShaderStageInfo.stage = VK\_SHADER\_STAGE\_FRAGMENT\_BIT;

fragShaderStageInfo.module = fragShader;

fragShaderStageInfo.pName = "main";

VkPipelineShaderStageCreateInfo shaderStages[] = { vertShaderStageInfo, fragShaderStageInfo };

**// Setup the vertex input**

VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};

vertexInputInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_VERTEX\_INPUT\_STATE\_CREATE\_INFO;

vertexInputInfo.vertexBindingDescriptionCount = 0;

vertexInputInfo.vertexAttributeDescriptionCount = 0;

**// Setup input assembly**

**// We will be rendering 2 triangles for the quad using triangle strip topology**

VkPipelineInputAssemblyStateCreateInfo inputAssembly = {};

inputAssembly.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_INPUT\_ASSEMBLY\_STATE\_CREATE\_INFO;

inputAssembly.topology = VK\_PRIMITIVE\_TOPOLOGY\_TRIANGLE\_STRIP;

inputAssembly.primitiveRestartEnable = VK\_FALSE;

**// Setup viewport to the maximum widht and height of the window**

VkViewport viewport = {};

viewport.x = 0.0f;

viewport.y = 0.0f;

viewport.width = (float)m\_swapChainExtent.width;

viewport.height = (float)m\_swapChainExtent.height;

viewport.minDepth = 0.0f;

viewport.maxDepth = 1.0f;

**// Setup scissor rect**

VkRect2D scissor = {};

scissor.offset = { 0, 0 };

scissor.extent = m\_swapChainExtent;

**// Setup view port state**

VkPipelineViewportStateCreateInfo viewportState = {};

viewportState.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_VIEWPORT\_STATE\_CREATE\_INFO;

viewportState.viewportCount = 1;

viewportState.pViewports = &viewport;

viewportState.scissorCount = 1;

viewportState.pScissors = &scissor;

**// Setup the rasterizer state**

VkPipelineRasterizationStateCreateInfo rasterizer = {};

rasterizer.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_RASTERIZATION\_STATE\_CREATE\_INFO;

rasterizer.depthClampEnable = VK\_FALSE;

rasterizer.rasterizerDiscardEnable = VK\_FALSE;

rasterizer.polygonMode = VK\_POLYGON\_MODE\_FILL;

rasterizer.lineWidth = 1.0f;

rasterizer.cullMode = VK\_CULL\_MODE\_BACK\_BIT;

rasterizer.frontFace = VK\_FRONT\_FACE\_CLOCKWISE;

rasterizer.depthBiasEnable = VK\_FALSE;

**// Setup multi sampling. In our first example we will be using single sampling mode**

VkPipelineMultisampleStateCreateInfo multisampling = {};

multisampling.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_MULTISAMPLE\_STATE\_CREATE\_INFO;

multisampling.sampleShadingEnable = VK\_FALSE;

multisampling.rasterizationSamples = VK\_SAMPLE\_COUNT\_1\_BIT;

**// Setup color output masks.**

**// Set to write out RGBA components**

VkPipelineColorBlendAttachmentState colorBlendAttachment = {};

colorBlendAttachment.colorWriteMask = VK\_COLOR\_COMPONENT\_R\_BIT | VK\_COLOR\_COMPONENT\_G\_BIT | VK\_COLOR\_COMPONENT\_B\_BIT | VK\_COLOR\_COMPONENT\_A\_BIT;

colorBlendAttachment.blendEnable = VK\_FALSE;

**// Setup color blending**

VkPipelineColorBlendStateCreateInfo colorBlending = {};

colorBlending.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_COLOR\_BLEND\_STATE\_CREATE\_INFO;

colorBlending.logicOpEnable = VK\_FALSE;

colorBlending.logicOp = VK\_LOGIC\_OP\_COPY;

colorBlending.attachmentCount = 1;

colorBlending.pAttachments = &colorBlendAttachment;

colorBlending.blendConstants[0] = 0.0f;

colorBlending.blendConstants[1] = 0.0f;

colorBlending.blendConstants[2] = 0.0f;

colorBlending.blendConstants[3] = 0.0f;

**// Create pipeline layout**

VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};

pipelineLayoutInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_LAYOUT\_CREATE\_INFO;

pipelineLayoutInfo.setLayoutCount = 0;

pipelineLayoutInfo.pushConstantRangeCount = 0;

VkResult vkResult = vkCreatePipelineLayout(m\_hDevice, &pipelineLayoutInfo, nullptr, &m\_hPipelineLayout);

if (vkResult != VK\_SUCCESS)

{

LogError("vkCreatePipelineLayout() failed!");

result = false;

}

else

{

**// Create graphics pipeline**

VkGraphicsPipelineCreateInfo pipelineInfo = {};

pipelineInfo.sType = VK\_STRUCTURE\_TYPE\_GRAPHICS\_PIPELINE\_CREATE\_INFO;

pipelineInfo.stageCount = 2;

pipelineInfo.pStages = shaderStages;

pipelineInfo.pVertexInputState = &vertexInputInfo;

pipelineInfo.pInputAssemblyState = &inputAssembly;

pipelineInfo.pViewportState = &viewportState;

pipelineInfo.pRasterizationState = &rasterizer;

pipelineInfo.pMultisampleState = &multisampling;

pipelineInfo.pColorBlendState = &colorBlending;

pipelineInfo.layout = m\_hPipelineLayout;

pipelineInfo.renderPass = m\_hRenderPass;

pipelineInfo.subpass = 0;

pipelineInfo.basePipelineHandle = VK\_NULL\_HANDLE;

vkResult = vkCreateGraphicsPipelines(m\_hDevice, VK\_NULL\_HANDLE, 1, &pipelineInfo, nullptr, &m\_hGraphicsPipeline);

if (vkResult != VK\_SUCCESS)

{

LogError("vkCreateGraphicsPipelines() failed!");

result = false;

}

}

**// Cleanup**

vkDestroyShaderModule(m\_hDevice, fragShader, nullptr);

vkDestroyShaderModule(m\_hDevice, vertShader, nullptr);

return (result);

}

## Drawing Setup

In this section, will implement the required steps to render a simple triangle on window output. This includes frame buffer setup, recording rendering command in command buffers, implementing render loop for drawing and presentation output.

### Frame buffer

A frame buffer(VkFrameBuffer) comprises of image views created from swapchain, In this section, we are going to create a frame buffer object for every image in the swap chain. vkCreateFramebuffer() is used to create the frame buffer object. The function takes VkFramebufferCreateInfo as input which specifies the render pass to use, swap chain image view, width & height of the frame buffer.

class VulkanApp : public VulkanHelper{

void **CreateFrameBuffers**();

. . .

std::vector<VkFramebuffer> **m\_hFramebuffers**;

};

void VulkanApp::CreateFramebuffers()

{

**// Resize the list based on swap chain image view count**

m\_hFramebuffers.resize(m\_hSwapChainImageViewList.size());

**// For each item in swap chain image view list**

for (size\_t i = 0; i < m\_hSwapChainImageViewList.size(); i++)

{

**// Setup VkFramebufferCreateInfo to create frame buffer**

VkFramebufferCreateInfo framebufferInfo = {};

framebufferInfo.sType = VK\_STRUCTURE\_TYPE\_FRAMEBUFFER-

\_CREATE\_INFO;

framebufferInfo.renderPass = m\_hRenderPass;

framebufferInfo.attachmentCount = 1;

framebufferInfo.pAttachments = &m\_hSwapChainImageViewList[i];

framebufferInfo.width = m\_swapChainExtent.width;

framebufferInfo.height = m\_swapChainExtent.height;

framebufferInfo.layers = 1;

**// Create frame buffer object**

VkResult vkResult = vkCreateFramebuffer(m\_hDevice, &framebufferInfo, nullptr, &m\_hFramebuffers[i]);

if (vkResult != VK\_SUCCESS)

{

LogError("vkCreateFramebuffer() failed!");

assert(false);

}

}

}

### Command Buffer

Command buffers are used to record rendering, compute or memory transfer operations (hardware commands) to execute in the GPU. Generally graphics driver is responsible for transforming rendering/compute/memory transfer API calls from application into GPU commands and store them in a command buffer. The command buffer is then submitted to the GPU to execute them based on synchronizations calls from application or when the command buffer is full. This is performed in every frame hence it may end up taking CPU cycles in translating API calls to GPU commands on every frame.

Vulkan expose the command buffer creation at the API level which enables application to reuse the command buffer as well as create many command buffers asynchronously and submit them to the GPU. This is one of the power of Vulkan for enabling application to use many cores in the CPU to generate command buffers.

The following steps are used to create command buffer objects in Vulkan:

1. Create a command pool object using vkCreateCommandPool(). A command pool object manages the memory to be used by the command buffer objects.
2. Create command buffer objects using vkAllocateCommandBuffers(). The number of command buffer objects is based on number of frame buffer objects in the swap chain.

class VulkanApp : public VulkanHelper{

virtual void CreateCommandBuffers(); **// Overide the default**

**implementation as per application requirement**

VkCommandPool m\_hCommandPool;

std::vector<VkCommandBuffer> m\_hCommandBufferList;

};

void VulkanApp::CreateCommandBuffers()

{

bool result = true;

// Create the command buffer pool object

VkCommandPoolCreateInfo poolInfo = {};

poolInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND\_POOL\_CREATE\_INFO;

poolInfo.flags = 0;

poolInfo.queueFamilyIndex = m\_physicalDeviceInfo.

graphicsFamilyIndex;

VkResult vkResult = vkCreateCommandPool(m\_hDevice, &poolInfo,

nullptr, &m\_hCommandPool);

if (vkResult == VK\_SUCCESS)

{

m\_hCommandBufferList.resize(

m\_hSwapChainImageViewList.size());

VkCommandBufferAllocateInfo allocInfo = {};

allocInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND-

\_BUFFER\_ALLOCATE\_INFO;

allocInfo.commandPool = m\_hCommandPool;

allocInfo.level = VK\_COMMAND\_BUFFER\_LEVEL\_PRIMARY;

allocInfo.commandBufferCount = (uint32\_t)

m\_hCommandBufferList.size();

vkResult = vkAllocateCommandBuffers(m\_hDevice,

&allocInfo, m\_hCommandBufferList.data());

if (vkResult != VK\_SUCCESS){

LogError("vkAllocateCommandBuffers() failed!");

assert(false);

}

}

else

{

LogError("vkCreateCommandPool() failed!");

assert(false);

}

assert (result);

}

### The Render Loop

The VulkanApp provide a run() function that renders and present the drawing output using render() and present() functions.

class VulkanApp : public VulkanHelper{

void Run(); **// Render loop**

virtual bool Render(); **// Draw the primitive on surface**

virtual bool Present(); **// Swap the drawn surface on window**

};

The VulkanApp::run() function is called from the Qt’s custom Window class at the fix rate of 1ms using. The Window class executes a slot Run() when the renderTimer signal’s triggers every 1ms, the slot calls the VulkanApp::Run() function resulting a continuous loop.

class Window : public QWindow

{

. . .

public slots:

void **Run()**;

private:

QTimer\* **renderTimer**; **// Refresh timer**

VulkanApp\* m\_VulkanApp; **// Used to call run() by the timer**

};

Window::Window(VulkanApp\* vulkanApp) : m\_VulkanApp(vulkanApp)

{

renderTimer = new QTimer();

renderTimer->setInterval(1);

connect(renderTimer, SIGNAL(timeout()), this, SLOT(Run()));

renderTimer->start();

}

void Window::Run()

{

m\_VulkanApp->Run();

}

#### Synchronization

Vulkan uses a separate queue for graphics and a separate queue for presenting the content to display, hence we need to create synchronization objects to synchronize images used in these 2 queues.

The following code creates 2 semaphore objects using vkCreateSemaphore()

1. One to signal that an image has been acquired for rendering
2. One to signal that rendering has finished and presentation can begin.

void VulkanApp::CreateSemaphores()

{

VkSemaphoreCreateInfo semaphoreInfo = {};

semaphoreInfo.sType = VK\_STRUCTURE\_TYPE\_SEMAPHORE\_CREATE\_INFO;

semaphoreInfo.flags = 0;

**// Create 2 semaphore objects**

**// One to signal when the image is ready to render**

**// Another one to signal when the image is ready to present**

if (vkCreateSemaphore(m\_hDevice, &semaphoreInfo, nullptr,

&m\_hRenderReadySemaphore) != VK\_SUCCESS ||

vkCreateSemaphore(m\_hDevice, &semaphoreInfo, nullptr,

&m\_hPresentReadySemaphore) != VK\_SUCCESS)

{

LogError("vkCreateSemaphore() failed");

assert (result);

}

}

#### Render

The following Render() method is called on every frame to submit the active graphics queue’s command buffer to the GPU. vkAcquireNextImageKHR() is used to obtain the active swap chain image index. Then VkSubmitInfo is initialized with the pointer to the command buffer object, semaphore object to wait before executing the graphics queue as well as semaphore object to signal after the graphics queue is executed in the GPU.

bool VulkanApp::Render()

{

m\_activeSwapChainImageIndex = 0;

const uint64\_t timeOut = std::numeric\_limits<uint64\_t>::max();

vkAcquireNextImageKHR(m\_pDevice, m\_hSwapChain, timeOut, m\_hRenderReadySemaphore, VK\_NULL\_HANDLE, &m\_activeSwapChainImageIndex);

VkPipelineStageFlags waitDstStages[] = { VK\_PIPELINE\_STAGE\_COLOR\_ATTACHMENT\_OUTPUT\_BIT };

**// Setup VkSubmitInfo to submit graphics queue workload to GPU**

VkSubmitInfo submitInfo = {};

submitInfo.sType = VK\_STRUCTURE\_TYPE\_SUBMIT\_INFO;

submitInfo.waitSemaphoreCount = 1;

submitInfo.pWaitSemaphores = &m\_hRenderReadySemaphore;

submitInfo.pWaitDstStageMask = waitDstStages;

submitInfo.commandBufferCount = 1;

submitInfo.pCommandBuffers = &m\_hCommandBufferArray[m\_activeSwapChainImageIndex];

submitInfo.signalSemaphoreCount = 1;

submitInfo.pSignalSemaphores = &m\_hPresentReadySemaphore;

VkResult vkResult = vkQueueSubmit(m\_pGraphicsQueue, 1, &submitInfo, VK\_NULL\_HANDLE);

return (vkResult == VK\_SUCCESS);

}

#### Present

Once the graphics queue is submitted to the GPU we need to call vkQueuePresentKHR() to execute the present queue. Note that graphics queue and render queue are asynchronous hence we need to setup proper synchronization to synchronize the present queue with the graphics queue. In the following method VkPresentInfoKHR is initialized with swap chain details, synchronization object to wait before executing the present queue. vkQueuePresentKHR() method is called to execute the present queue.

bool VulkanApp::Present()

{

bool result;

**// Setup VkPresentInfoKHR to execute the present queue**

VkPresentInfoKHR presentInfo = {};

presentInfo.sType = VK\_STRUCTURE\_TYPE\_PRESENT\_INFO\_KHR;

presentInfo.waitSemaphoreCount = 1;

presentInfo.pWaitSemaphores = &m\_hPresentReadySemaphore;

presentInfo.swapchainCount = 1;

presentInfo.pSwapchains = &m\_hSwapChain;

presentInfo.pImageIndices = &m\_activeSwapChainImageIndex;

result = vkQueuePresentKHR(m\_pPresentQueue, &presentInfo);

if (result == VK\_SUCCESS)

{

// Wait for present queue to become idle

result = vkQueueWaitIdle(m\_pPresentQueue);

}

return (result == VK\_SUCCESS);

}

#### Drawing a Triangle

The base class (VulkanApp) implemented methods to create the command buffer objects, render pass, frame buffer etc. In the application class, we need to record the commands to bind the graphics pipeline, render pass and draw command in the command buffer to render the triangle. Note that we have a command buffer for each of the swap chain buffer hence we need to add the draw command in all command buffers associated with the swap chain buffer.

For each of the command buffer in the command buffer list,

1. Begin the command buffer by calling vkBeginCommandBuffer()
2. Setup the VkRenderPassBeginInfo structure and call vkCmdBeginRenderPass() to begin the render pass.
3. Bind the default graphics pipeline created by the VulkanApp base class.
4. Call vkCmdDraw() to submit 3 vertices to render the triangle.
5. End the render pass with vkCmdEndRenderPass()
6. End the command buffer vkEndCommandBuffer()

The following method contains the full implementation for HelloVulkan::Setup() that records the app specific rendering commands to command buffers in the command buffer list.

bool HelloVulkanApp::Setup()

{

**// Background color (Blue)**

VkClearValue clearColor = { 0.0f, 0.0f, 1.0f, 1.0f };

**// Offset to render in the frame buffer**

VkOffset2D renderOffset = { 0, 0 };

**// Width / Height of to render in the frame buffer**

VkExtent2D renderExtent = m\_hSwapChainExtent;

**// For each command buffers in the command buffer list**

for (size\_t i = 0; i < m\_hCommandBufferArray.size() && (result == true); i++)

{

VkCommandBufferBeginInfo beginInfo = {};

beginInfo.sType = VK\_STRUCTURE\_TYPE\_COMMAND\_BUFFER\_BEGIN\_INFO;

**// Indicate that the command buffer can be resubmitted to the queue**

beginInfo.flags = VK\_COMMAND\_BUFFER\_USAGE\_SIMULTANEOUS\_USE\_BIT;

**// Step 1: Begin command buffer**

vkBeginCommandBuffer(m\_hCommandBufferArray[i], &beginInfo);

VkRenderPassBeginInfo renderPassInfo = {};

renderPassInfo.sType = VK\_STRUCTURE\_TYPE\_RENDER\_PASS\_BEGIN\_INFO;

renderPassInfo.renderPass = m\_hRenderPass;

renderPassInfo.framebuffer = m\_hSwapChainFramebufferArray[i];

renderPassInfo.renderArea.offset = renderOffset;

renderPassInfo.renderArea.extent = renderExtent;

renderPassInfo.clearValueCount = 1;

renderPassInfo.pClearValues = &clearColor;

**// Step 2: Begin render pass**

vkCmdBeginRenderPass(m\_hCommandBufferArray[i], &renderPassInfo, VK\_SUBPASS\_CONTENTS\_INLINE);

**// Step 3: Bind graphics pipeline**

vkCmdBindPipeline(m\_hCommandBufferArray[i], VK\_PIPELINE\_BIND\_POINT\_GRAPHICS, m\_hGraphicsPipeline);

**// Step 4: Draw a quad using 4 vertices**

vkCmdDraw(m\_hCommandBufferArray[i], 4, 1, 0, 0);

**// Step 5: End the Render pass**

vkCmdEndRenderPass(m\_hCommandBufferArray[i]);

**// Step 6: End the Command buffer**

VkResult vkResult = vkEndCommandBuffer(m\_hCommandBufferArray[i]);

if (vkResult != VK\_SUCCESS)

{

LogError("vkEndCommandBuffer() failed!");

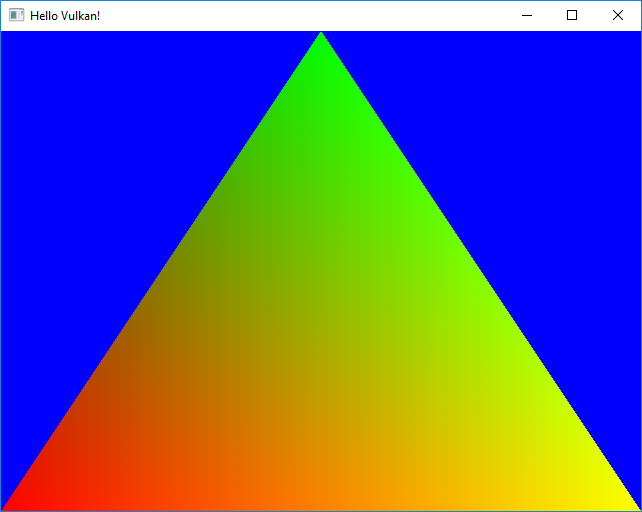
assert(false);

}

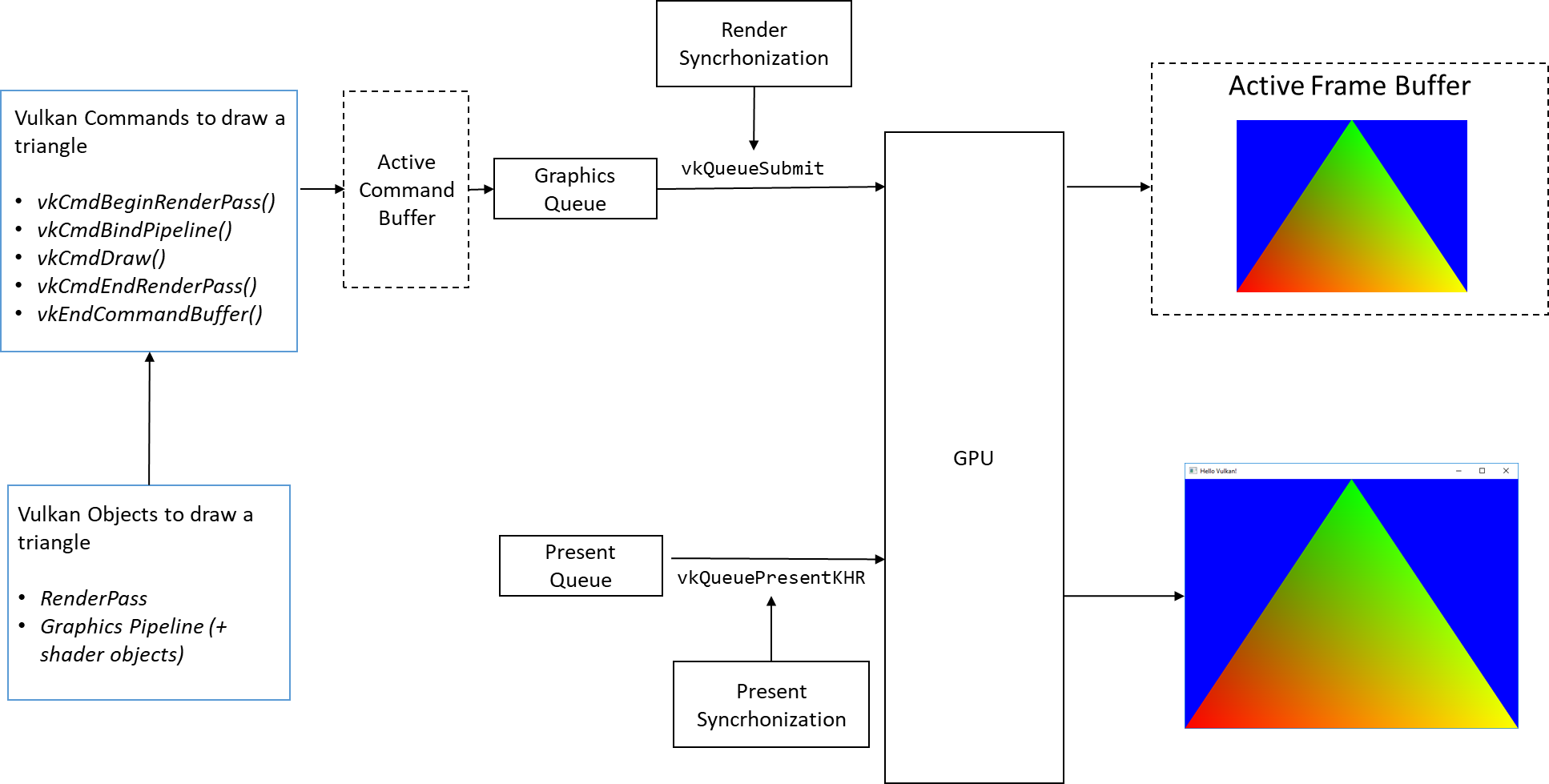
}

}

Finally the implementation to render a triangle is complete. Run the application in Visual Studio to see an output that should look like this:



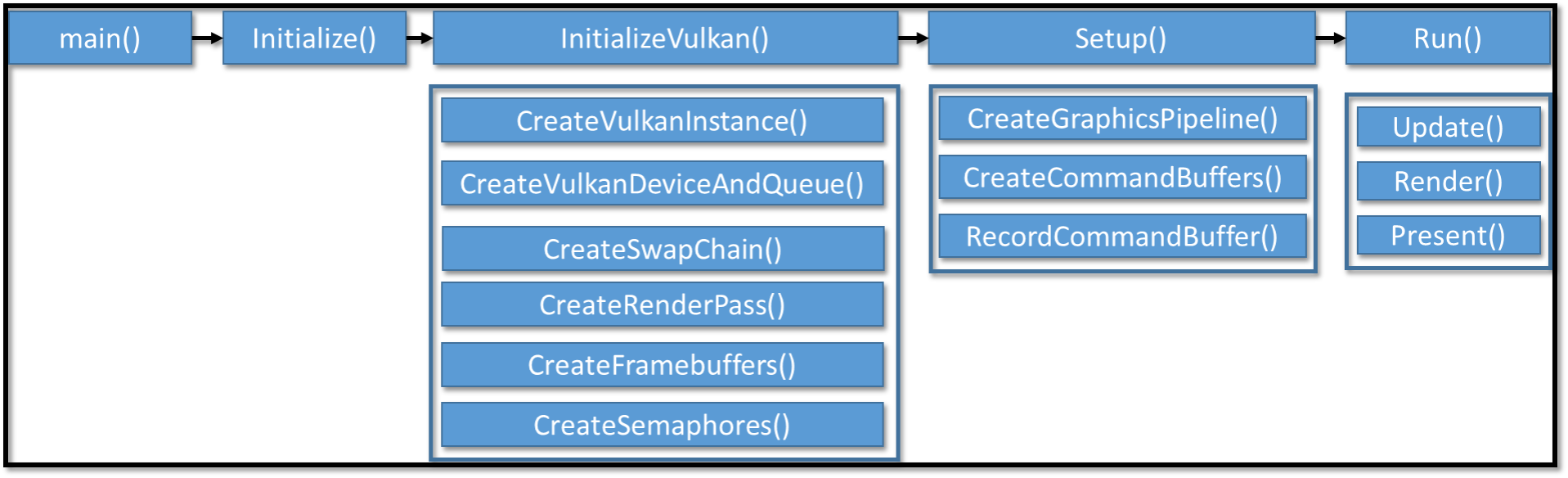
The following diagram illustrates the over-all flow of the application to render the final output in the window.



The following diagram show the program API flow, the program starts with main() and does the initialization(initialize()) followed by render loop(Run()).

The initialization performs two things: First, it initialize the general prerequisites for Vulkan application. This include creating of Vulkan instance, creating device and queue, building swapchain, defining the render pass and creating framebuffer. Secondly, the application setup is done which very specific to what application is intended to implement, here we will setup the graphics pipeline, we create the command buffer and record them into the render pass.

The Renderloop executes in the Run() function where we can update the geometry, render it and present it on the window surface.



## Clean Up

The application must destroy the object in the opposite order in which they are created. Once the user close the application window, the destructor of HelloVulkanApp is called followed up by the base class’s VulkanApp destructor to destroy any application and Vulkan specific objects. The objects must be destroyed in the opposite order in which they are created.

The following method destroys all Vulkan objects created to initialize Vulkan in the base class.

HelloVulkanApp::~HelloVulkanApp()

{

vkDestroyPipeline(m\_hDevice, m\_hGraphicsPipeline, nullptr);

vkDestroyPipelineLayout(m\_hDevice, m\_hPipelineLayout, nullptr);

}

VulkanApp::~VulkanApp()

{

vkDestroySemaphore(m\_hDevice,

m\_hPresentReadySemaphore, nullptr);

vkDestroySemaphore(m\_hDevice, m\_hRenderReadySemaphore,

nullptr);

vkDestroyCommandPool(m\_hDevice, m\_hCommandPool, nullptr);

for (size\_t i = 0; i < m\_hFramebuffers.size(); i++)

{

vkDestroyFramebuffer(m\_hDevice, m\_hSwapChainFrame-

bufferList[i], nullptr);

}

vkDestroyRenderPass(m\_hDevice, m\_hRenderPass, nullptr);

for (size\_t i = 0; i < m\_hSwapChainImageViewList.size(); i++)

{

vkDestroyImageView(m\_hDevice, m\_hSwapChainImage-

ViewList[i], nullptr);

}

vkDestroySwapchainKHR(m\_hDevice, m\_hSwapChain, nullptr);

vkDestroyDevice(m\_hDevice, nullptr);

vkDestroySurfaceKHR(m\_hInstance, m\_hSurface, nullptr);

vkDestroyInstance(m\_hInstance, nullptr);

}

We have successfully implemented all required methods in VulkanApp class to setup Vulkan. The following diagram illustrate the dependencies of Vulkan objects we created so far to initialize Vulkan.

# Buffer Resource

Vulkan divides the memory resource object in to two types - Buffer and Images.

* **Buffer:** In this type of memory resource the data content is stored in the linear array form. The buffer object in Vulkan is represent by VkBuffer.
* **Image:** These are formatted arrays used to store texture contents in the memory. The image object in Vulkan are represent by VkImage.

In this section, we will take our existing program ahead and create the triangle using buffer resource. In the last section, we stored our triangle vertices and colors statically in the vertex shader, now we will use the buffer resource store these vertices in GPU buffer and share it with vertex a shader attribute. Below is the geometry data contaning the vertex position and color in the interleaved form in the Vertex structure in HelloVulkanApp.h

struct Position { float x, y, z, w; }; // Vertex Position

struct Color{ float r, g, b, a; }; // Color format RGBA

struct Vertex

{

Position m\_Position;

Color m\_Color;

};

static const Vertex s\_TriangleVertices[3] =

{

{ Position{0.0f,1.0f,0.0f,1.0f}, Color{1.0f,0.0f,0.0f,1.0} },

{ Position{1.0f,-1.0f,0.0f,1.0f}, Color{0.0f,0.0f,1.0f, 1.0} },

{ Position{-1.0f,-1.0f,0.0f,1.0f}, Color{0.0f,1.0f,0.0f,1.0} },

};

Following are the data structure and functions added in the same file for creating the vertex buffer.

struct {

VkBuffer m\_Buffer;

VkDeviceMemory m\_Memory;

VkDescriptorBufferInfo m\_BufferInfo;

} VertexBuffer;

class HelloVulkanApp : public VulkanApp

{

private:

void CreateVertexBuffer(const void \*vertexData,

uint32\_t dataSize, uint32\_t dataStride);

**// Vertex buffer specific objects**

VkVertexInputBindingDescription m\_VertexInputBinding;

VkVertexInputAttributeDescription m\_VertexInputAttribute[2];

};

## Creating Vertex Buffer

The buffer creation is simple and is a 5 step process, let’s walk to this process step-by-step:-

1. Create the buffer resource object (VkBuffer): First create the buffer object (VkBuffer) using vkCreateBuffer() API. This API use a metadata structure VkCreateBufferInfo that provides necessary information about the buffer object that is going to be created, this include buffer size, usage type, and creation flags etc.

In Vulkan, when buffers object are created they do not have any physical backing at that point. Allocation of the physical memory is a separated process all together and mentioned in step 3. Both these discrete steps are bind together in step 5 of this section where the allocation memory is attached to the created vertex buffer object.

void HelloVulkanApp::CreateVertexBuffer(const void \* vertexData, uint32\_t dataSize, uint32\_t dataStride){

**// 1. Create the Buffer resource**

VkBufferCreateInfo bufInfo = {};

bufInfo.sType = VK\_STRUCTURE\_TYPE\_BUFFER\_CREATE\_INFO;

bufInfo.pNext = NULL;

bufInfo.usage = VK\_BUFFER\_USAGE\_VERTEX\_BUFFER\_BIT;

bufInfo.size = dataSize;

bufInfo.queueFamilyIndexCount = 0;

bufInfo.pQueueFamilyIndices = NULL;

bufInfo.sharingMode = VK\_SHARING\_MODE\_EXCLUSIVE;

bufInfo.flags = 0;

vkCreateBuffer(m\_hDevice, &bufInfo, NULL,

&VertexBuffer.m\_Buffer);

1. Get memory specific requirements: Query theappropriate size of the memory needed by the buffer resource allocation using vkGetBufferMemoryRequirements() API and determining right memory type from the available memories and choose the one that matches application requirement.

**// 2. Get memory specific requirements**

**// 2a. Get the Buffer resource requirements**

VkMemoryRequirements memRqrmnt;

vkGetBufferMemoryRequirements(m\_hDevice,

VertexBuffer.m\_Buffer, &memRqrmnt);

**// 2b. Get the compatible type of memory**

VkMemoryAllocateInfo allocInfo = {};

allocInfo.sType = VK\_STRUCTURE\_TYPE\_-

MEMORY\_ALLOCATE\_INFO;

allocInfo.pNext = NULL;

allocInfo.memoryTypeIndex = 0;

allocInfo.allocationSize = memRqrmnt.size;

VulkanHelper::MemoryTypeFromProperties(

m\_physicalDeviceInfo.memProp, memRqrmnt.memoryTypeBits, VK\_MEMORY\_PROPERTY\_HOST\_VISIBLE\_BIT | VK\_MEMORY\_PROPERTY\_HOST\_COHERENT\_BIT, &allocInfo.memoryTypeIndex);

1. Allocate physical backing: Allocate physical device memory using vkAllocateMemory() API, this will create the memory and return in VkDeviceMemory object.

**// 3. Allocate the physical backing**

vkAllocateMemory(m\_hDevice, &allocInfo, NULL, &(VertexBuffer.m\_Memory));

assert(result == VK\_SUCCESS);

VertexBuffer.m\_BufferInfo.range = memRqrmnt.size;

VertexBuffer.m\_BufferInfo.offset = 0;

1. Copy data into buffer: Copy the geometry data into the allocated memory by mapping the physical device memory using vkMapMemory(), after copying the data unmap it with vkUnmapMemory().

**// 4. Copy data into buffer**

**// 4a. Map the physical device memory region to the host**

uint8\_t \*pData;

vkMapMemory(m\_hDevice, VertexBuffer.m\_Memory, 0,

memRqrmnt.size, 0, (void \*\*)&pData);

**// 4b. Copy the data in the mapped memory**

memcpy(pData, vertexData, dataSize);

**// 4c. Unmap the device memory**

vkUnmapMemory(m\_hDevice, VertexBuffer.m\_Memory);

1. **Bind the allocated memory:** Since allocation and buffer creation are two different process we need to bind them together so that the buffer object can point to the allocated physical backing. Binding of the device memory (VkDeviceMemory) to the buffer object (VkBuffer) if performed using vkBindBufferMemory() API.

vkBindBufferMemory(m\_hDevice, VertexBuffer.m\_Buffer,

VertexBuffer.m\_Memory, 0);

## Defining Vertex Attributes

The geometry information is bind to the shaders in the form of vertex attributes. The vertex input binding information is specified to the graphics pipeline using VkVertexInputBindingDescription structure object m\_VertexInputBinding, this indicates the rate at which the vertex information fetched and what is the size of stride for each vertex.

m\_VertexInputBinding.binding = 0;

m\_VertexInputBinding.inputRate = VK\_VERTEX\_INPUT\_RATE\_VERTEX;

m\_VertexInputBinding.stride = dataStride;

Secondly, the VkVertexInputAttributeDescription structure object m\_VertexInputAttribute in order to interpret the vertex information. Such as the binding indicate from where the vertex information is read by the shader. Also it contains, vertex attribute information for all attribute in the shader, such as in this case position and colors attributes location which specify from where the shader will read different attribute types. For example, we used 0 as position’s the bind point. Within the vertex, we need to mention the offset in bytes indicating where a specific attribute data starts with the given vertex. The format indicates that the format type of each individual vertex attribute.

**// The VkVertexInputAttribute helps in interpreting the data.**

m\_VertexInputAttribute[0].binding = 0;

m\_VertexInputAttribute[0].location = 0;

m\_VertexInputAttribute[0].format = VK\_FORMAT\_-R32G32B32A32\_SFLOAT;

m\_VertexInputAttribute[0].offset = offsetof(struct Vertex, m\_Position);;

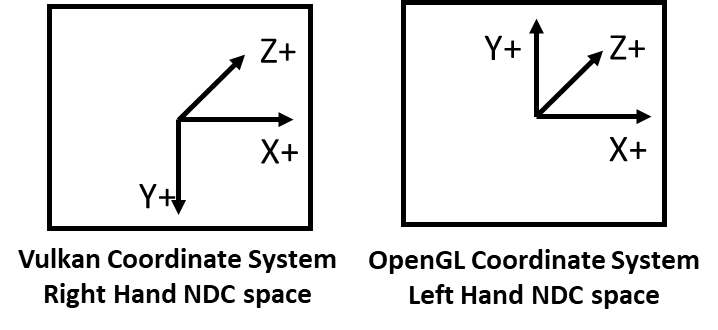
m\_VertexInputAttribute[1].binding = 0;

m\_VertexInputAttribute[1].location = 1;

m\_VertexInputAttribute[1].format = VK\_FORMAT\_R32G32B32A32\_SFLOAT;

m\_VertexInputAttribute[1].offset = offsetof(struct Vertex, m\_Color);

Now, the shader file Triangle.vert are sent with two attributes inPosition and inColor in in the attribute index 0 and 1 respectively. The Y coordinate is inversed since Vulkan follow Right Hand NDC space where the Y+ direction is downwards.



**// Filename: Triangle.vert**

#version 450

#extension GL\_ARB\_separate\_shader\_objects : enable

layout (location = 0) in vec4 inPosition;

layout (location = 1) in vec4 inColor;

layout(location = 0) out vec4 fragColor;

out gl\_PerVertex {

vec4 gl\_Position;

};

void main()

{

gl\_Position = inPosition;

fragColor = inColor;

gl\_Position.y = -gl\_Position.y; **// From GL to Vulkan conventions**

}

### Update Graphics Pipeline

So far the graphics pipeline has not clue that we introduced two new vertex attributes (position and color) in the system. Therefore, we must specify this information with in HelloVulkanApp’s CreateGraphicsPipeline() when we define the vertex input metadata in the VkPipelineVertexInputStateCreateInfo structure as follows:

**// Setup the vertex input**

VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};

vertexInputInfo.sType = VK\_STRUCTURE\_TYPE\_PIPELINE\_-

VERTEX\_INPUT\_STATE\_CREATE\_INFO;

vertexInputInfo.vertexBindingDescriptionCount =

sizeof(m\_VertexInputBinding) / sizeof(VkVertexInputBindingDescription);

vertexInputInfo.pVertexBindingDescriptions = &m\_VertexInputBinding;

vertexInputInfo.vertexAttributeDescriptionCount =

sizeof(m\_VertexInputAttribute) / sizeof(VkVertexInputAttributeDescription);

vertexInputInfo.pVertexAttributeDescriptions =

m\_VertexInputAttribute;

### Binding Vertex Buffer into Render Pass

When the render pass command buffer is recorded in RecordCommandBuffer bind the vertex buffer object with this render pass.

**// Step 4: Specify vertex buffer information**

const VkDeviceSize offsets[1] = { 0 };

vkCmdBindVertexBuffers(m\_hCommandBufferList[i], 0, 1,

&VertexBuffer.m\_Buffer, offsets);

# Detect Memory leaks with Validation layer

In the previous, section we allocated the vertex buffer and successfully executes the program however, we intentionally left a memory leak in the program to demonstrate how validation layers are so useful while development.

So where the memory leaked?

1. We create the vertex buffer but we did not destroyed it.
2. We allocate vertex device memory but did not free it.

Without validation layer the program will execute normally and exits without indicating you if there is leak of memory undergoing. In order to detect general validation layer while development you can enable “VK\_LAYER\_LUNARG\_standard\_validation” when Vulkan instance is created in CreateVulkanInstance() as follows:

std::vector<const char \*> **validationLayers** = {

"VK\_LAYER\_LUNARG\_standard\_validation"

};

**// Fill in the required createInfo structure**

VkInstanceCreateInfo createInfo = {};

createInfo.sType = VK\_STRUCTURE\_TYPE\_INSTANCE\_CREATE\_INFO;

. . . . .

createInfo.enabledLayerCount = **validationLayers**.size();

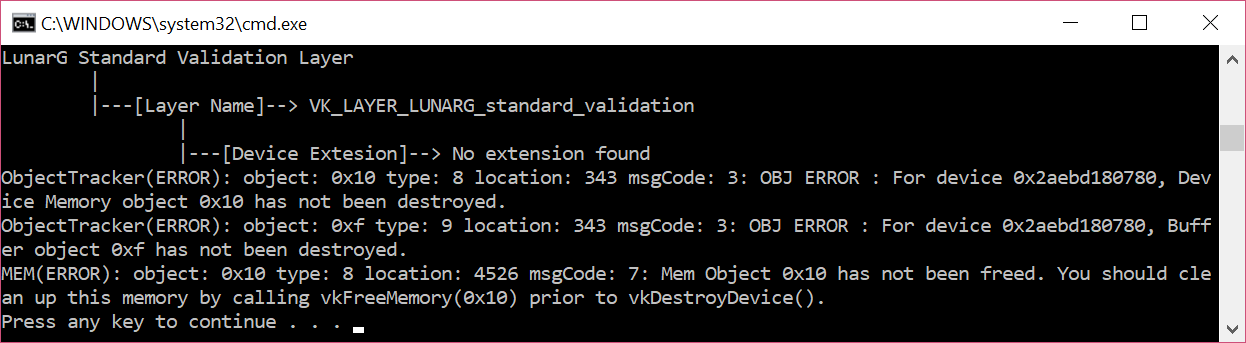
createInfo.ppEnabledLayerNames = **validationLayers**.data();

. . . . .

Enabling VK\_LAYER\_LUNARG\_standard\_validation will automatically enables a set of validation layer in the following order:

VK\_LAYER\_GOOGLE\_threading VK\_LAYER\_LUNARG\_parameter\_validation VK\_LAYER\_LUNARG\_device\_limits VK\_LAYER\_LUNARG\_object\_tracker VK\_LAYER\_LUNARG\_image VK\_LAYER\_LUNARG\_core\_validation VK\_LAYER\_LUNARG\_swapchain VK\_LAYER\_GOOGLE\_unique\_objects

Adding the above code and executing the program will generate the following output, indicating where the memory leak in the program is.



The output indicate the buffer object and device memory object is not been destroyed. This can be fixed by free the allocated memory using vkFreeMemoryand vertex buffer object withvkDestroyBufferin the destructor. The sample code has intentionally commented the below code in order to show the memory leak.

HelloVulkanApp::~HelloVulkanApp()

{

vkDestroyPipeline(m\_hDevice, m\_hGraphicsPipeline, nullptr);

vkDestroyPipelineLayout(m\_hDevice, m\_hPipelineLayout, nullptr);

**// Destroy Vertex Buffer**

**vkDestroyBuffer(m\_hDevice, VertexBuffer.buffer, NULL);**

**vkFreeMemory(m\_hDevice, VertexBuffer.memory, NULL);**

}

After properly destructing the allocated resources the validation will not give the same error, indicating the program is memory leak free.

## Summary

In this chapter, we take off our journey of learning Vulkan where we introduce the very basic concept of Vulkan to our readers revealing its competitive edge over traditional API and other existing APIs. We provided a short overview on the Vulkan programing model that helps existing programmers to build a big picture before they delve into the Vulkan internals. We setup our Vulkan development environment and build the first Hello World! example in a step-by-step manner, we learn through many new Vulkan jargons and terminologies that are used commonly in Vulkan.

Next, we learnt different type of resource in Vulkan and build a practical example to use buffer resource in Vulkan. We also demonstrated how validation layer can be used to detect the memory leaks.

In the next chapter, we create a “Media wall” application to render many images using Vulkan. Here you will learn how to use Vulkan image resource type and display texture. We will also introduce descriptor and descriptor set to see how can we use the uniforms and share them with shaders.