CSC8820 Advanced Graphics Algorithms

Spring 2018

# Ashish Tiwari

**002364321**

Project #3

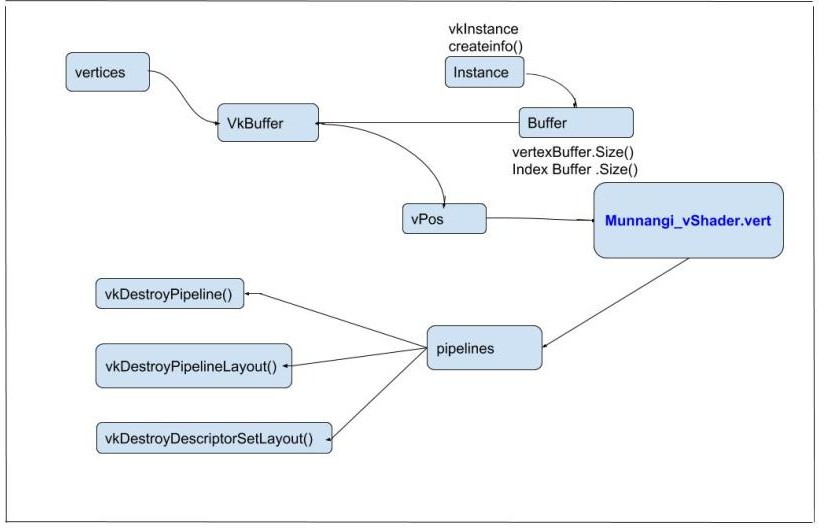
Due date: 11:59 pm, April 9, 2018 (Monday)

In this project, you will learn the basics of drawing and lighting using Vulkan API.

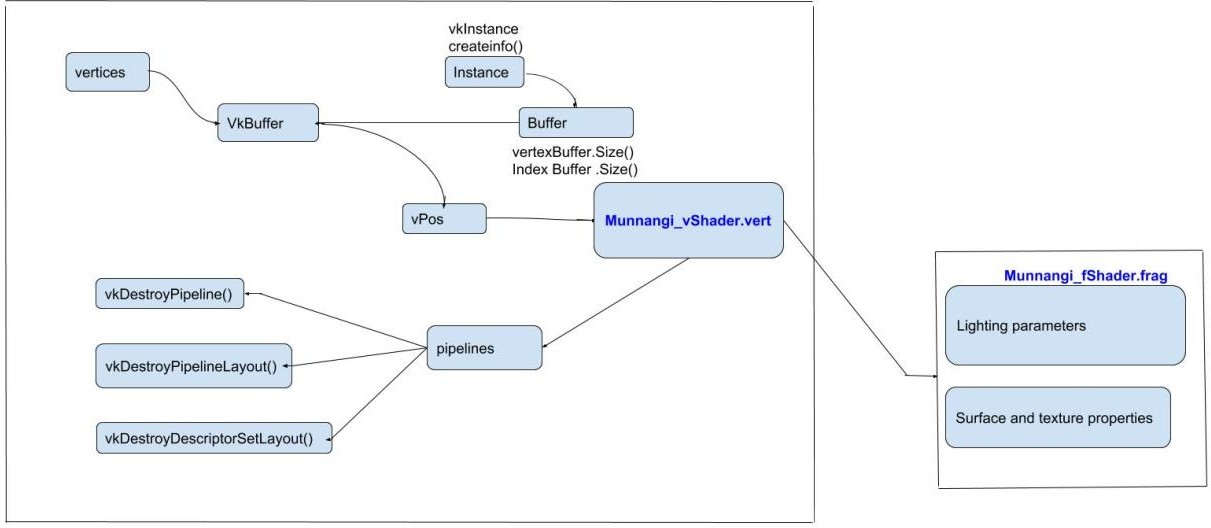
Requirements (100 points total):

1. Write a C or C++ program that draws the 3D object in the attached OBJ file. The output of the program should be like the picture below.
   1. The 3D object file pyramid.obj is attached.
   2. Use Vulkan to develop your program. You can download Vulkan SDK from <https://vulkan.lunarg.com>. Detailed instructions on how to install and compile Vulkan SDK is included with the package or can be found online. Additional instructions can be found at <https://sites.google.com/site/csc8820/educational/vulkan-basics>.
   3. (60 points) The object shall be lighted. You must write a vertex shader and a fragment shader. The lighting must be implemented in the fragment shader. There should be at least one light source.   
         
      

1. (20 points) Draw two data flow diagrams. One diagram shows how the lighting parameters are transferred from the host program to the shader program. The other diagram shows how the vertex data is transferred from the host program to the shader program.
   1. Use free online diagramming tools such as Google Drawing.



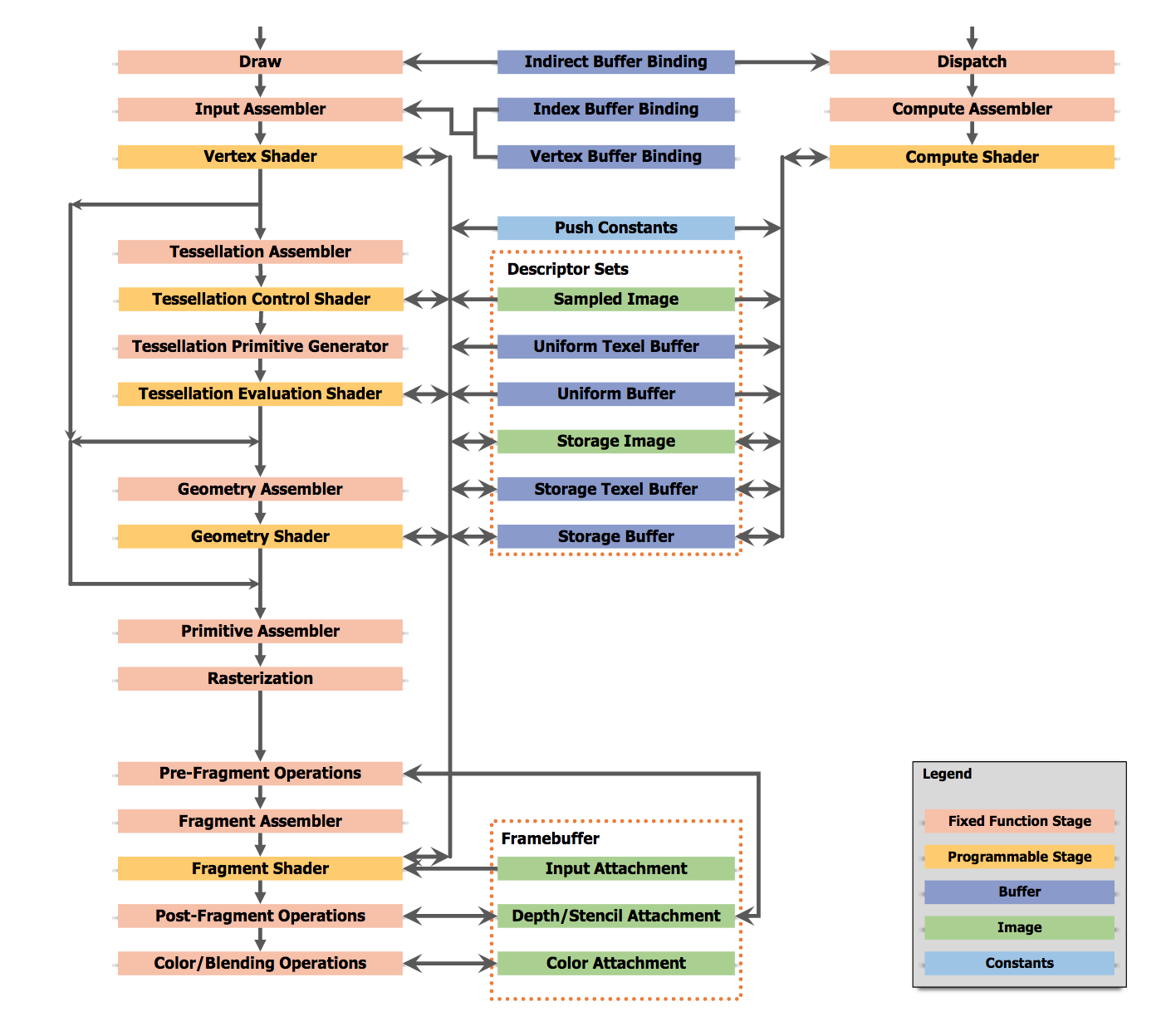
Tiwari\_vShader.vert



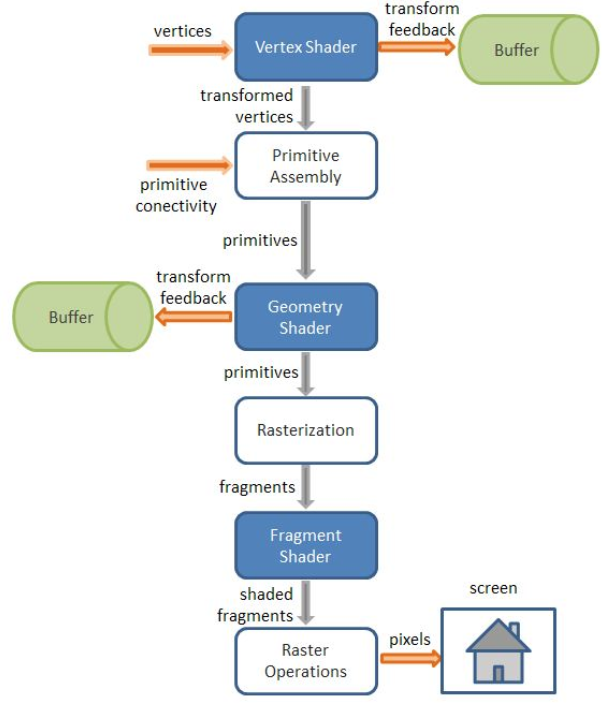
Tiwari\_vShader.vert

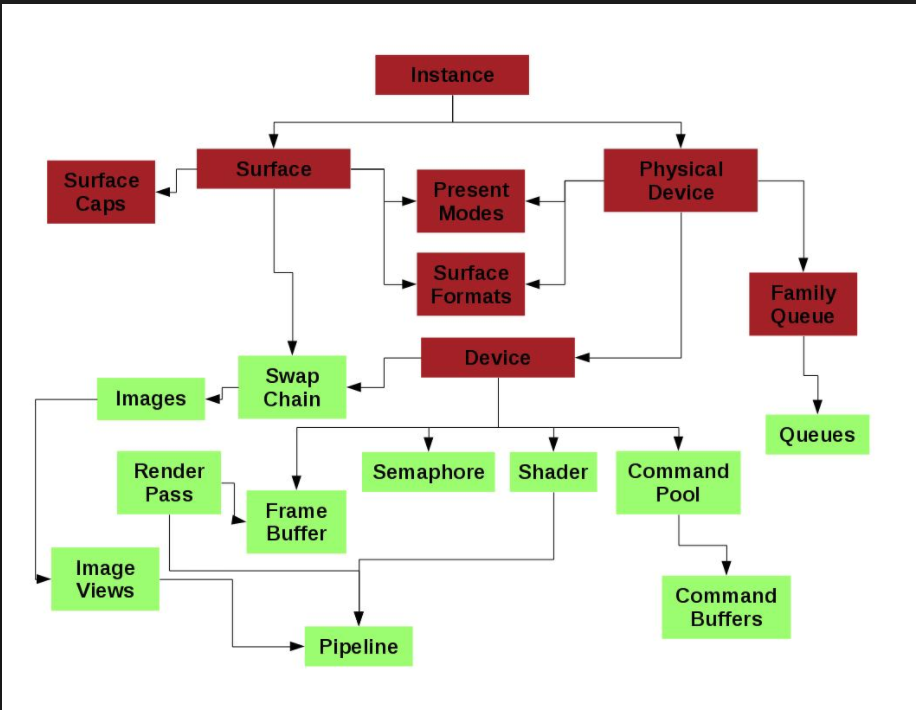
Tiwari\_fShader.frag

1. **(20 points) Answer the following questions:**
   1. **Compare Vulkan and OpenGL in three areas: pipeline set, data transfer, and data processing. In which part is Vulkan more complicated than OpenGL?**

**Compare PipleLine Set in OpenGL and Vulkan as in the below images.**

**Above is Vulkan Pipeline**



**Above is OpenGL Pipeline of Rendering data(Data Transfer)**

**Above image of Vulkan Data Transfer**

**In which part is Vulkan more complicated than OpenGL ?**

**Pipeline area is more complicated in Vulkan than in OpenGL**

* 1. **What design decisions make Vulkan more efficient than OpenGL?**

Okay, let’s get to the nitty gritty of the porting experience and how we approached it. As mentioned earlier about multithreading issues - our engine was single threaded. Therefore, in the first approach  we designed the Vulkan backend on the same model as our existing OpenGL backend. Effectively we just implemented another API without changing any of the existing structure of the engine. In order to achieve that we created a default command buffer to which we internally recorded commands, but all still on one thread. We also had to create several objects such as Pipelines and Pipeline layouts which were reused for subsequent similar drawcalls. On the other hand, we called vkQueueWaitIdle after every vkQueueSubmit which caused stalls and compromised performance. That, however, wasn’t our priority as the main goal was to just get the engine backend up and running on Vulkan.

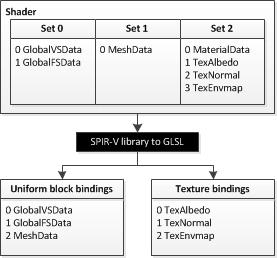
In the engine we had a custom shader language that we can translate for different standards: HLSL, GLSL and GLSL ES. Basically just yet another shader cross compiler. The Vulkan API comes with the new SPIR-V language so that was another significant chunk of work to be done. First we had to cross compile using our custom compiler and then compile this down to SPIR-V and feed it to the Vulkan driver. What we found interesting in compiling this down to SPIR-V was being able to extract binding information directly from the SPIR-V binary blob in order to create compatible pipeline layouts.

In GL, a framebuffer doesn’t have a clear beginning and end. The only way you can tell that with GL is SwapBuffer, clearing/discarding or by switching framebuffer objects. The Vulkan API addresses this by having the developer explicitly declare when a drawing begins and ends. The initial implementation we did was mimicking the GL behavior so that when we either called SwapBuffer or changed framebuffer, we ended the render pass and started a new one. This was obviously a temporary solution and definitely not what we would call best practice, but it helped us get the engine working to the same level as OpenGL with any content, except maybe for performance! then Having the engine working on the Vulkan API as described above doesn’t give you better performance or the benefit of any of the cool features of Vulkan. In order to improve upon the process described above, we took the following steps to make the engine work more efficiently and to properly utilize Vulkan features.

We made the following key decisions:

* Sacrifice some OpenGL performance to get the best out of Vulkan
* Implement block based memory allocator
* Implement new command buffer abstraction
* Implement new explicit render pass interface
* Standardize on SPIR-V for all our shaders

The first step was redesigning the engine backend and, more specifically, the structures and functionality which were responsible for interfacing with the driver. Unfortunately, once we ported to Vulkan, content running on OpenGL performed slightly worse than previously. This is mainly due to descriptor set emulation that we implemented in the OpenGL backend. This means that for our OpenGL backend we added support for allowing multiple sets in the shaders. The SPIR-V cross compiler removes the sets in the OpenGL GLSL output and replaces it with a serialized list of binding locations together with a remapping table. The descriptor set emulation in OpenGL then has to rebind and remap textures, samplers, uniform buffers, etc depending on the remapping table. This means that we get a slight performance hit if we have lots of shaders using wildly different binding tables because of the API overhead of rebinding.



But this is a decision that you will have to make for you graphics engine. We wanted to support descriptor sets at a high level, but this can also be hidden at a lower level in your engine so that you don’t have to think about handling descriptor set remapping in your OpenGL implementation.

* 1. **What is SPIR-V? Why use SPIR-V? What is the relationship between SPIR-V and GLSL?**

**What is SPIR-V ?**

SPIR-V is a Khronos-standard intermediate language that provides an alternative for distributing shaders. OpenGL accepts shaders in SPIR-V form much like it accepts shaders in GLSL form. Typically, for SPIR-V form, an offline tool chain will generate SPIR-V from a high-level shading language such as GLSL, and rather than distributing GLSL source with your application, you would distribute the generated SPIR-V.

SPIR-V is created, distributed, and consumed as binary units called modules. A SPIR-V module can live in memory as a sequence of 32-bit words, or it can be stored as a file, again, as a sequence of 32-bit words. However, as with GLSL, OpenGL does not deal with files, so SPIR-V must be handed to OpenGL as a pointer to an in-memory sequence of 32-bit words.

Each SPIR-V module contains one or more entry points, as places to begin shader execution, and each entry point knows what OpenGL pipeline stage it belongs to. Each of these entry points must form a single, complete OpenGL pipeline stage. That is, unlike in desktop GLSL, SPIR-V shaders don’t hold multiple compilation units to later link together to form a single stage. For SPIR-V, such linkage would be done offline by a front end when it translates the high-level language form to SPIR-V, yielding a result that is a fully linked stage. A single SPIR-V module may contain many entry points, even for the same stage.

SPIR-V modules can be specialized, which means changing the values of some specially identified constants inside the module before final compilation at run time. This is done to reduce the number (or size) of SPIR-V modules needed to represent multiple slight variations of a shader.

**Why Use SPIR-V ?**

There are several potential reasons you might distribute shaders in SPIR-V:

* Better portability. One problem with portability is that each platform’s driver can have a slightly different interpretation of the high-level rules for GLSL. High-level languages are in part high-level because of the freedom of expressiveness they allow the coder. However, the limits of this freedom are sometimes hard to completely pin down, leading to variance in interpretation. SPIR-V is much stricter and much more regular about how constructs are expressed, leaving less room for interpretation. This in turn leads to less variance between platforms’ interpretation of SPIR-V and, hence, improved portability. Of course, you are not coding in SPIR-V, so you still have GLSL to contend with. However, for generating SPIR-V, you can select a single front end for all the platforms you target. That is, you can eliminate portability problems that originate from different GLSL interpretations by sticking with a single GLSL front end. Someone else might select a different front end for their shaders, and that’s fine too. What matters is that one application’s GLSL shaders get the same GLSL interpretation for all platforms on which the SPIR-V(Khronos-standard intermediate language) ends up running.
* Other source languages. SPIR-V enables use of high-level languages other than GLSL. If the distributed SPIR-V is of correct form, it does not matter how it was generated.
* Reduced distribution size. SPIR-V has multiple features to dramatically reduce the size of shader collections as they are distributed. Individual shaders, on their own, are typically larger in SPIR-V than in GLSL, but individual shaders are small in either case. Collections of related shaders can, however, be quite large, and two SPIR-V features in particular are aimed at addressing such collections: specialization and multiple entry points per module. Specialization allows late changing of some constant values, and multiple entry points in the same SPIR-V module allow shipping a single instance of a function body that might be used by many entry points. GLSL distribution might have distributed a copy of the function for each shader in the collection, SPIR-V distribution is able to ship only one copy.
* Protecting your source code. This is sometimes referred to as obfuscation, as there are times you don’t want to distribute your shader source code in an easy-to-leverage form. Shader source code can represent novel ideas or intellectual property, which you don’t want to distribute to other parties in a transparent, easily modifiable format. You can avoid distributing your source code by offline compilation of your source to SPIR-V and distribution of the SPIR-V instead. This makes it much harder to see how a shader achieves an effect. Yes, it is still conceivable that a reverse compiler can re-create GLSL or some other high-level shading language, which would compile down to the SPIR-V you distributed. However, the need for a recipient to undertake such a reverse-engineering activity provides real protection to your intellectual property.

Runtime compiler performance is often sought as another reason to select an intermediate language over a high-level language, but caution is needed here. A high-performing shader executable will typically require scheduling and register allocation algorithms, executed at runtime, that are themselves time-consuming. These later steps cannot be eliminated by using a portable intermediate language. Runtime compiler performance, however, is improved in a number of ways. For one, parsing a high-level language takes some time. Although parsing is normally a small portion of the compilation stack, it becomes more significant for shaders that have lots of unused code or when multiple shaders compile down to the same intermediate result. In these cases, notable parsing time is eliminated through use of SPIR-V. Also, some high-level optimizations can be performed offline, but take care not to perform platform-specific optimizations that would hurt performance on some targets. For example, it might be platform-dependent whether all functions should be inlined at all call sites.

**What is the relationship between SPIR-V and GLSL ?**

SPIR-V is similar to GLSL, but it has some differences. Two differences are particularly relevant.

1. A single SPIR-V file can have function entry-points for multiple shader stages, even of different types.
2. SPIR-V has the concept of "specialization constants": parameters which the user can provide before the SPIR-V is compiled into its final form.

Before a SPIR-V shader object can be used, you must specify which entry-point to use and provide values for any specialization constants used by that entry-point. This is done through a single function:

void [glSpecializeShader](https://www.khronos.org/opengl/wiki_opengl/index.php?title=GLAPI/glSpecializeShader&action=edit&redlink=1)(GLuint shader , const GLchar \*pEntryPoint , GLuint numSpecializationConstants , const GLuint \*pConstantIndex , const GLuint \*pConstantValue );

pEntryPoint  is the string name of the entry-point that this SPIR-V shader object will use. pConstantIndex  and pConstantValue  are arrays containing the index of each specialization constant and the corresponding values that will be used. These arrays are numSpecializationConstants  in length. Specialization constants not referenced by pConstantIndex .

Specializing a SPIR-V shader is analogous to compiling a GLSL shader. So if this function completes successfully, the shader object's compile status is GL\_TRUE. If specialization fails, then the shader infolog has information explaining why and an [OpenGL Error](https://www.khronos.org/opengl/wiki/OpenGL_Error) is generated.

pEntryPoint  must name a valid entry point. Also, the entry point's "execution model" (SPIR-V speak for "[Shader Stage](https://www.khronos.org/opengl/wiki/Shader_Stage)") must match the stage the shader object was created with. Specialization can also fail if pConstantIndex  references a specialization constant index that the SPIR-V binary does not use. If specialization fails, the shader's info log is updated appropriately.

Once specialized, SPIR-V shaders cannot be re-specialized. However, you can reload the SPIR-V binary data into them, which will allow them to be specialized again.

1. **Students whose computers do not support Vulkan still need to fulfill the requirements 1, 2, and 3. However, it is understood that the code may not compile but the code will be inspected for features. For these students, requirement #1 will be worth 40 points. The students need to answer the following questions (20 points):**
   1. **Describe how Vulkan layers can be used for debugging purposes.**

Vulkan™ provides unprecedented control to developers over generating graphics and compute workloads for a wide range of hardware, from tiny embedded processors to high-end workstation GPUs with wildly different architectures. As usual, with great power comes great responsibility, and making sure that your application runs correctly on all these possible target platforms it is crucial to follow all the rules of the API specification even if some level of violation of these rules, either intentional or accidental, seem to not cause any issues on a particular hardware and driver implementation.

Traditional graphics APIs try to solve this issue by defining a set of illegal API usage conditions that are required to be caught by driver implementations and reported to the application through some sort of error reporting mechanism. The problem with this approach is that even though these errors generated in response to incorrect API usage are extremely valuable during the development of an application, checking for all of these error conditions costs significant CPU time spent in the driver that provides no value when running a released application that is known to use the API correctly. Not to mention the fact that practice reveals some driver implementations are less pedantic about certain rules established by the API specifications than others, and thus relying on testing on a particular implementation and observing no problems could still lead to portability issues when the same application is ran against other driver implementations.

**Errors vs Errors**

Unlike traditional graphics APIs, Vulkan groups possible error scenarios into two distinct buckets:

* **Validity errors** are error conditions resulting from incorrect API usage, i.e. the application not respecting the API usage rules that are required in order to get well-defined behavior from the issued commands. These rules are described in the specification for all API commands and structures in text blocks titled “**Valid Usage**“.
* **Run-time errors** are error conditions that can occur even during the execution of applications that use the API correctly, like running out of memory, or failure to present to a window that has been closed in the meantime. Run-time errors are reported in the form of result codes. The specification describes the possible result codes each command may return individually in the form of text blocks titled “**Return Codes**“, accompanied with language describing the situations when each particular result code is expected to be returned by driver implementations.

While many of the Vulkan API commands do return a result code in the form of one of the constants of the VkResult enumeration, these result codes are only used to indicate run-time errors and status information about certain operations or objects, but do not report information about respecting valid usage conditions. This allows release builds of applications to run at maximum performance because the driver implementations don’t have to spend precious CPU cycles on checking for the potential violation of specification rules as that’s anyways unnecessary in case of applications that are known to use the API correctly.

As driver implementations aren’t checking valid usage conditions and expect that all inputs coming from the application to be valid according to the specification, running applications that use the API incorrectly may result in unexpected behavior, including corrupted rendering or even application crashes. Often the consequences of passing invalid parameters to an API command might only manifest when executing latter commands.

**Validation Layers to the Rescue**

We already acknowledged that not having to check for valid API usage for release builds of applications that are known to behave correctly from the point of view of the Vulkan API specification has great benefits, but it’s still very important to be able to identify incorrect API usage during the development of an application because finding the mistake we made that results in the weird corruption we see or the mysterious crash we can’t explain is not trivial to debug without a hint about where we should look for the error.

In order to provide a solution for this, Vulkan comes with a set of validation and debug layers as part of the [Vulkan SDK](https://vulkan.lunarg.com/app/download). At the time of writing the SDK includes almost a dozen layers dedicated for validating certain aspects of API usage and providing debugging tools to developers like an API call dumper. When any subset of these layers are enabled they insert themselves automatically into the call-chain of every Vulkan API call issued by the application to perform their job. A detailed description of the individual layers is outside of the scope of this article but curious readers can find more information [here](https://vulkan.lunarg.com/app/docs/latest/layers).

The benefit of validation layers compared to the approach taken by traditional APIs is that applications only have to spend time on extensive error checking when explicitly requested, during development and typically when using debug builds of the application. This fits naturally in the general *pay for what you use* principle of the Vulkan API. Additionally, as the official validation layers coming with the SDK are maintained centrally and work equivalently across driver implementations, this approach doesn’t suffer from the fragmentation issues often seen in the error checking behavior of traditional APIs thus developers can be confident that the same validation errors are going to be reported in all cases, indifferent of the driver implementation the application is ran against.

What’s even better, the validation layers aren’t just looking for violations of the allowed API usage, but can also report warnings about potential incorrect or dangerous use of the API, and are even capable of reporting performance warnings that allow developers to identify places where the API is used correctly but isn’t used in the most efficient way. Examples of such potential performance warnings are binding resources that aren’t actually used or using a sub-optimal layout for an image.

Application developers willing to validate their API usage during development are going to be primarily interested in VK\_LAYER\_LUNARG\_standard\_validation that bulks all standard validation layers in a big meta-layer. Enabling this layer ensures that all official validation layers will going to be keen on trying to catch any mistake the application makes in the use of Vulkan. In order to report the caught violations of valid API usage to the application the validation layers expose theVK\_EXT\_debug\_report instance extension that allows feeding the detected validation errors and warnings to application-provided callbacks. We are going to present the basic usage of this extension in this article but more information is available in the [Vulkan Registry](https://www.khronos.org/registry/vulkan/).

**Preparing Our Instance For Validation**

We recommend that all applications should enable and use the validation layers in their debug builds in order to make sure their applications are always respecting valid API usage and thus are going to be portable across the wide range of Vulkan driver implementations.

The following code snippet shows a typical C++ example of how applications should enable theVK\_LAYER\_LUNARG\_standard\_validation layer and the VK\_EXT\_debug\_report extension at instance creation time in their debug builds:

std::vector enabledInstanceLayers;

std::vector enabledInstanceExtensions;

#ifdef MY\_DEBUG\_BUILD\_MACRO

/\* Enable validation layers in debug builds to detect validation errors \*/

enabledInstanceLayers.push\_back("VK\_LAYER\_LUNARG\_standard\_validation");

#endif

/\* Enable instance extensions used in all build types \*/

enabledInstanceExtensions.push\_back("VK\_KHR\_surface");

...

#ifdef MY\_DEBUG\_BUILD\_MACRO

/\* Enable debug report extension in debug builds to be able to consume validation errors \*/

enabledInstanceExtensions.push\_back("VK\_EXT\_debug\_report");

#endif

/\* Setup instance creation information \*/

VkInstanceCreateInfo instanceCreateInfo = {};

...

instanceCreateInfo.enabledLayerCount = static\_cast<uint32\_t>(enabledInstanceLayers.size());

instanceCreateInfo.ppEnabledLayerNames = &enabledInstanceLayers[0];

instanceCreateInfo.enabledExtensionCount = static\_cast<uint32\_t>(enabledInstanceExtensions.size());

instanceCreateInfo.ppEnabledExtensionNames = &enabledInstanceExtensions[0];

/\* Create the instance \*/

VkInstance instance = VK\_NULL\_HANDLE;

VkResult result = vkCreateInstance(&instanceCreateInfo, nullptr, &instance);

*Editor’s Note: Based on your input I’ve replaced the use of the*NDEBUG*macro to indicate code that is meant to be built only in debug versions of the application and now the code examples refer to a custom macro called*MY\_DEBUG\_BUILD\_MACRO*that you should replace with the debug build macro used by your project or compiler toolchain.*

Of course, a resilient application should first check for the presence of the used instance layers and extensions before passing them to [vkCreateInstance](https://www.khronos.org/registry/vulkan/specs/1.0/man/html/vkCreateInstance.html) by using the[vkEnumerateInstanceLayerProperties](https://www.khronos.org/registry/vulkan/specs/1.0/man/html/vkEnumerateInstanceLayerProperties.html) and [vkEnumerateInstanceExtensionProperties](https://www.khronos.org/registry/vulkan/specs/1.0/man/html/vkEnumerateInstanceExtensionProperties.html)commands, respectively. After a successful instance creation the validation layers are active for the instance and the debug report extension is available for use.

As the VK\_EXT\_debug\_report instance extension is not a core feature, the addresses of its entry points have to be acquired through the use of the [vkGetInstanceProcAddr](https://www.khronos.org/registry/vulkan/specs/1.0/man/html/vkGetInstanceProcAddr.html) command as shown in the code snippet below:

#ifdef MY\_DEBUG\_BUILD\_MACRO

/\* Load VK\_EXT\_debug\_report entry points in debug builds \*/

PFN\_vkCreateDebugReportCallbackEXT vkCreateDebugReportCallbackEXT =

reinterpret\_cast<PFN\_vkCreateDebugReportCallbackEXT>

(vkGetInstanceProcAddr(instance, "vkCreateDebugReportCallbackEXT"));

PFN\_vkDebugReportMessageEXT vkDebugReportMessageEXT =

reinterpret\_cast<PFN\_vkDebugReportMessageEXT>

(vkGetInstanceProcAddr(instance, "vkDebugReportMessageEXT"));

PFN\_vkDestroyDebugReportCallbackEXT vkDestroyDebugReportCallbackEXT =

reinterpret\_cast<PFN\_vkDestroyDebugReportCallbackEXT>

(vkGetInstanceProcAddr(instance, "vkDestroyDebugReportCallbackEXT"));

#endif

Our First Debug Report Callback

We’ll talk about each individual entry point of the extension separately, but first let’s take a look at how an application-provided debug report callback should look like and what behavior it should follow. The application can register any number of debug report callbacks, they only need to match the signature defined by PFN\_vkDebugReportCallbackEXT . A sample debug report callback that simply directs all incoming debug messages to stderr is presented below:

VKAPI\_ATTR VkBool32 VKAPI\_CALL MyDebugReportCallback(

VkDebugReportFlagsEXT flags,

VkDebugReportObjectTypeEXT objectType,

uint64\_t object,

size\_t location,

int32\_t messageCode,

const char\* pLayerPrefix,

const char\* pMessage,

void\* pUserData)

{

std::cerr << pMessage << std::endl;

return VK\_FALSE;

}

The parameters passed to the callback provide information about where and what type of validation event has triggered the call, like the type of the event (error, warning, performance warning, etc.), the type and handle of the object being created or manipulated by the command triggering the call, the code and text message describing the event, and there’s even a parameter to supply application-specific user data to the callback which is provided when registering the callback. By putting a breakpoint in the callback, developers can also have access to the complete callstack to more accurately determine the location of the offending API call.

The return value of the callback is a Boolean that indicates to the validation layers whether the API call that triggered the debug report callback should be aborted or not. However, developers have to be aware that in case an error is reported by one of the validation layers it’s an indication that something invalid was being attempted by the application thus any operation following the error might result in undefined behavior or even a crash. As such, it’s advised that developers stop at the first error and try to resolve that before making any assumptions about the behavior of subsequent operations. Think about validation errors in the same way like errors reported by compilers: often subsequent errors are just consequences of the first one.

When registering our debug report callback, we can specify what type of events we want to get notification about. Typically we’re interested in errors, warnings, and performance warnings; the following code snipped registers our callback with such a configuration:

#ifdef MY\_DEBUG\_BUILD\_MACRO

/\* Setup callback creation information \*/

VkDebugReportCallbackCreateInfoEXT callbackCreateInfo;

callbackCreateInfo.sType = VK\_STRUCTURE\_TYPE\_DEBUG\_REPORT\_CREATE\_INFO\_EXT;

callbackCreateInfo.pNext = nullptr;

callbackCreateInfo.flags = VK\_DEBUG\_REPORT\_ERROR\_BIT\_EXT |

VK\_DEBUG\_REPORT\_WARNING\_BIT\_EXT |

VK\_DEBUG\_REPORT\_PERFORMANCE\_WARNING\_BIT\_EXT;

callbackCreateInfo.pfnCallback = &MyDebugReportCallback;

callbackCreateInfo.pUserData = nullptr;

/\* Register the callback \*/

VkDebugReportCallbackEXT callback;

VkResult result = vkCreateDebugReportCallbackEXT(instance, &callbackCreateInfo, nullptr, &callback);

#endif

An already registered callback can then be unregistered by destroying the callback object just like any other API object using the corresponding destroy command, vkDestroyDebugReportCallbackEXT. Developers should make sure to unregister their debug report callbacks before destroying the instance, otherwise they going to be notified about their misbehavior through any debug report callback that’s registered to receive errors.

The last remaining entry point of the debug report extension that we didn’t discuss yet,vkDebugReportMessageEXT can be used to generate debug report messages from application code. This can be useful to mark certain points of the execution of the application or to report application specific information to the same stream where the validation messages are fed.

***Update:****Since version 1.0.13 of the Vulkan API specification and the Vulkan SDK device layers have been deprecated so the instructions related to enabling the validation layers at the device level have been removed accordingly.*

**Forcing Validation Externally**

The recommended way to validate an application is the approach presented so far, because it allows developers to enable validation based on the type of the build, as presented, based on some application setting, or through any other mechanism. Additionally, the debug report callback enables fine grained control over which validation events should be captured and how.

However, in some cases it’s possible that modifying or rebuilding the application to enable validation programatically is not viable or convenient. This includes cases like validating release builds of applications that don’t reproduce the issue in debug builds, or validating third-party applications or libraries that we cannot rebuild because of lack of access to the source code.

There’s a solution even for situations like this, as layers can also be enabled through the environment variable VK\_INSTANCE\_LAYERS . This variable accepts a list of layer names to enable separated by semicolons (Windows) or colons (Linux). The following command enables all standard validation layers on Windows:

> set VK\_INSTANCE\_LAYERS=VK\_LAYER\_LUNARG\_standard\_validation

When enabling validation through this approach, besides setting the environment variable to activate the layers, the reporting mechanism must be configured for each layer via a settings file, otherwise the activated layers will produce no output. This settings file must be named vk\_layer\_settings.txtand must be located in the working directory of the application or in the directory specified using theVK\_LAYER\_SETTINGS\_PATH environment variable. A sample layer settings file is provided as part of the Vulkan SDK under the config folder which will simply output all error, warning, and performance warning messages to stdout , if used, but can be easily changed to output a different subset of the validation messages and can be redirected to files instead of console output (which may be necessary to capture the output of applications without a console). The sample settings file contains instructions about how to change the various configuration options.

**Summary**

While getting familiar with the Vulkan API may seem a bit involving at the beginning, as due to its nature it has a steeper learning curve than traditional APIs, the validation layers make it much easier to catch any mistakes, and they also provide a lot of additional useful information beyond just reporting basic errors. While using the validation layers does not completely eliminate the need to test your application on multiple platforms, it minimizes the chances of any portability issues resulting from incorrect API usage.

1. Name your source file (lastname)\_proj3.cc. In the source code, write your name and any special instructions for building and running your program. Either embed your shaders in your program or save them in separate files. Name your shaders (lastname)\_vshader.glsl and (lastname)\_fshader.glsl.
2. Submit your source code as well as a report. The report should contain your diagrams and answers to the questions (requirement 2, 3, or 4).
3. Upload your source code and report in a ZIP file to iCollege under the folder Project 3. Do not submit via email.