LAB 6 Vulkan

# Introduction

While Lab 5 heavily relied on previous code from labs 3 & 4, you technically could start from a blank project. This lab and any future lab cannot. They must directly build upon your previous assignments to be effective.

In this lab we will explore loading 2D textures, using them as material attributes to impact the lighting algorithm.

# LAb 6

## SECTION A | 25% - Project Setup & Loading the first texture into a buffer

### Task A1

Get started by just setting up your project. Copy the completed lab 5 and use it as the base code.

Make sure you edit and re-run the CMakeLists.txt file to accurately name your project solution. I also recommend adjusting the lighting variables and materials from the previous assignment to a basic white material & specular.

A white cat on a red background

Description automatically generated

***Hint:*** *To get this look, I moved the camera to -0.5x, +0.25y, +0.5z and pointed the sun direction at +1x, -1y, -2z.*

### task A2

Load the texture referenced by the first material in the model into a buffer. (Use TextureUtils.h)

### task A3

Check that the texturing functions are all succeeding & be sure to release all resources.

Unfortunately, RenderDoc only lets us inspect resources in use, so we will need to wait a little longer to confirm.

## section b | 50% - Setting up the Vulkan Descriptors & Rendering the first texture

#### task b0 | SECTION NOTES/FORWARD:

Descriptor Sets and Layouts, while complex, are also highly configurable by nature. There is not necessarily only one way to achieve something due to this flexibility. (Though it can certainly feel that way sometimes)

For example: When I first wrote this section, I achieved it by just adding an additional image/sampler binding to the existing layout (which already contained the uniform buffer). However, this was not an optimal solution due to having to bind the textures to multiple VkDescriptorSets. Because static textures are never updated between frames, it made little practical sense to bundle them with cycling uniform or storage buffer sets which are.

Instead, the following steps will ask you to create a separate descriptor set and layout for your textures (sharing the same descriptor pool). This is the technique used in the Vulkan texturing sample in the API\_SAMPLES repo.

### task b1

Create a new Descriptor Set Layout & add a single texture/sampler combo.

### task b2

Adjust the descriptor pool to reserve space to hold the texture.

### task b3

Create a new descriptor set which will hold the texture.

### task b4

Bind the texture to the new descriptor set.

### task b5

Adjust the pipeline layout to include the new descriptor set layout.

### task b6

Bind the new texture descriptor set to the pipeline during rendering.

Now we should be able to see the texture in RenderDoc:

A computer screen shot of a computer

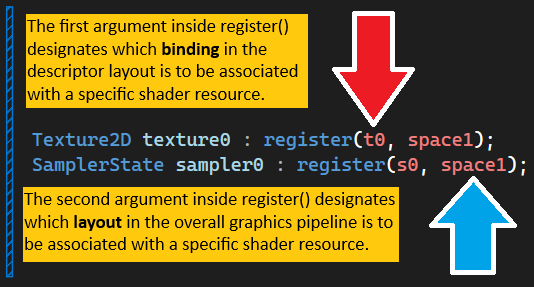
Description automatically generated

The texture looks a bit odd (AI UV unwrapping), but it should be the same as if you opened it in an image viewer.

Be sure to also click on the **Mip** dropdown UI control so you can see that all mip-levels were generated correctly.

### task b7

Adjust the fragment shader to replace the hard-coded diffuse material color with base map texture. When adjusting the shader code, pay special attention to the **register** HLSL keyword. This is a cross API way to link the language to specific resources available on the graphics card.



Once you successfully sample and apply the material’s base color it should look very similar to this:

A cat with green eyes

Description automatically generated

A big improvement! However, you will notice the model is *very* shiny. Makes it look like our kitty here is made of plastic. We will fix this in the next two parts by integrating the “metallicRoughness” texture also included.

## section c | 75% - Unbounded Resource extension & Loading the second texture

#### task C0 | SECTION NOTES/FORWARD:

Though technically not strictly required to load and use our second texture, unbound texture arrays are a staple of modern rendering engines/techniques. They allow for **vastly** more flexible approaches to using textures or other GPU resources than ever before!

Unfortunately, they are not enabled by default in Vulkan. They must be accessed by turning on a specific extension and using new EXT structures to notify descriptors they can be used in far less rigid ways.

In the past, graphics programmers were always constrained by the limited textures that could be bound to the pipeline for a specific draw call. This new extension grants random access to texture arrays for each & every pixel.

Ever heard of an older id game called [Rage?](https://mrelusive.com/publications/presentations/2009_siggraph/05-JP_id_Tech_5_Challenges.pdf) Its but one historical example of trying to bypass such HW limitations.

### task c1

Enable **BINDLESS\_SUPPORT** in the GVulkanSurface. Adjust the Create(…) function in main.cpp to append the new graphics option **bit flag** in both Debug & Release modes. (Do not carry over the validation layer to Release mode)

*Tip: Gateware simplifies adding this extension to Vulkan. However, some older hardware may not support it.*

### task c2

Convert the texture objects to arrays of handles.

A computer screen shot of text

Description automatically generated

Swap the existing code over to iterate through the above array instead and make sure everything still works.

### task c3

Load the “metallicRoughness” texture also referenced by the first material in the model into the texture array.

A green cat statue on a red background

Description automatically generated

Once you do this you may see the second texture overwrite the first texture in Vulkan. To resolve this, we will need to adjust the descriptor layout and bindings to compensate. (this includes using previous mentioned extensions)

### task c4

While in theory we could just add another separate texture sampler, this is precisely where the limitations of the old hardware approach would start to become an issue. Instead, we will convert the existing texture descriptor set layout to use a texture/sampler array.

To do this you must create a **VkDescriptorSetLayoutBindingFlagsCreateInfoEXT** and assign it’s “.**pBindingFlags**” member to a single **VkDescriptorBindingFlagsEXT** variable which has been initialized to the bit flag of: **VK\_DESCRIPTOR\_BINDING\_PARTIALLY\_BOUND\_BIT\_EXT**. What this does is request the ability to utilize only some of the available descriptors at any given time.

To connect this new feature, adjust the **descriptor create info’s** “**.pNext**” member to point to the address of this structure right before creating the existing descriptor layout for the textures.

Finally, you will want to modify the “**.descriptorCount**” of the texture descriptor’s **layout binding** to reflect the maximum number of textures now accessible within our new array. (Avoid hard-coding this)

### task c5

You may be tempted at this point to adjust the size of the Descriptor Pool. However, this is NOT necessary, because we are not actually adding another descriptor set, rather we are converting the existing set to a bindless array that can hold ALL our textures. (which is MUCH better than managing individual descriptor sets!)

Instead, just head over to where you created the original texture descriptor set. Create a **VkDescriptorSetVariableDescriptorCountAllocateInfoEXT** which must be initialized to point to an array which contains the size of each descriptor set array we are initializing. In our case we just have the one descriptor set, but it will now be an array that contains our two textures.

Complete this section by pointing the original descriptor set’s allocate info “**.pNext**” member to the address our new EXT structure describing the properties of our bindless texture array.

### task c6

Now that we have space in an array for multiple textures, we need to **update** all the array slots to have the appropriate Image Views. Adjust your texturing code for **vkUpdateDescriptorSets** so instead of just writing one texture, it writes all the ones you loaded from the GLTF file.

The easiest way I found to do this, is to create an array of **VkDescriptorImageInfo** and use that instead of trying to update each array slot individually.

*Tip:* *Vulkan does not (by default) allow the writing of NULL image views/buffers to descriptor sets. If you want to support “gaps” in your array, you will need to individually write only the valid entries one by one. (dstArrayElement)*

### task c7

Check that the texture is correctly loaded by looking in RenderDoc.

A screenshot of a computer

Description automatically generated

If you don’t see it or have a bunch of Vulkan validation errors in the console: You will need to clear those up and carefully retrace the steps in this section. There should be no visual impact yet, but you need to be clear of errors & the green/cyan texture needs to be selectable/appear in the Texture Viewer when selecting the Draw call.

## section d | 100% - Applying the second texture to the lighting model

### task d1

Adjust the fragment shader to use an array of textures & samplers instead of the singular versions currently in use.

A black background with blue and white text

Description automatically generated

Adjust the code to use the first texture in the array and the visual results should stay the same with no errors.

### task d2

Now that you have array support working, switch the code to use the second texture in the array.

A green cat statue on a red background

Description automatically generated

This is the “roughnessMetallic” texture. As mentioned in the [Blender GLTF 2.0 specification](https://docs.blender.org/manual/en/4.1/addons/import_export/scene_gltf2.html?utm_source=blender-4.1.0#metallic-and-roughness), this texture contains Roughness in the **Green** channel and Metalness in the **Blue** channel. It also can optionally contain baked Ambient Occlusion in the **Red** channel, but our model does not appear to have any Red in this texture.

### task d3

Adjust the fragment shader code so it displays just the **green** channel of this texture **only**: (no lighting)

A computer screen shot of a mouse

Description automatically generated

You can see that while the roughness is mostly uniform, there are spots where there is some minor variation in smoothness. The brighter the green, the “rougher” the surface. In a human made 3D model, I would expect some significant variation around the eyes and the nose(darker). But since this model was AI generated, we should keep our expectations in-check for now.

### task d3

Go ahead and do the same with the blue channel to see what is considered metallic:

A blue cat on a red background

Description automatically generated

Since a cat is not made of metal, it may seem a bit odd that there is any blue at all. However, keep in mind that the “metalness” also helps represent the environmental reflectivity of a material in the PBR model. If I had to guess, the presence of blue on the Black coat is saying that it is somewhat “smooth & shiny” while the white fur is more “poofy” and non-reflective.

Again, this was generated by a brand-new 3D Mesh AI model, so we are not exactly expecting perfection here. In any case, we won’t be using the metalness just yet regardless since we are not using PBR lighting equations.

### task d4

Without an environment map loaded (a.k.a Skybox), our use case for this texture is generally going to be limited to using the roughness component/channel to dampen our Specular Highlights.

Switch the shader back to using the base color like it was before. However, this time also load in the roughness value from the second texture. Use the roughness to dampen the specular highlights in the final formula. Basically, the higher the value, the rougher and less shiny the surface should appear.

Once you get this implemented, the shininess of the surface should be significantly more subtle and vary a bit:

A cat sitting on a red background

Description automatically generated

At first you may think you have just disabled specular highlights. But if you pan across the model slowly you should be able to see a very *subtle sheen* across the cat’s fur. The model you selected may (hopefully) contain much more obvious roughness differences. Again, our cat here is not made of metal or plastic, so any specular highlights should be very minimal and barely noticeable.

*Tip:* *Change the specular color multiplier (Red?) if you are having a hard time spotting these more subtle highlights.*

### TASK D5

As you can see, the specular interpretation on the cat is way too subtle to really notice. Let’s switch to a human made 3D model that more clearly demonstrates changes in specular. Go ahead and swap out the AI generated Cat model with the provided fish model created by an actual human artist. (don’t forget to export it from Blender first)

A fish on a red background

Description automatically generated

Unlike the cat where the specular highlights are almost imperceptible, the fish should clearly show areas that shine overall. Consider rotating the light source or the 3D model slowly to see the differences even more clearly.

A screenshot of a computer

Description automatically generatedA fish on a red background

Description automatically generated

Unlike the cat model, notice that this model also has a red or “ambient occlusion” component in the “occlusion roughness metallic” texture. In the images above, the left image is ignoring the red channel, while the right image is using the ambient occlusion to reduce ambient light where applicable.

The effect is quite subtle but note how around the mouth & eyes of the right fish it appears to have more depth, while the mouth and eyes on the left fish seem flatter in appearance. I increased the overall ambient light in the scene to 0.5f to make this effect easier to see. Do the same shader & lighting adjustment in your assignment.

# Summary

While this does implement our texture’s roughness & occlusion, it is unfortunately not quite as accurate as it could be. Roughness was designed for PBR shading models, not the older Blinn-Phong model we are using currently. Notice the use of a fixed Exponent (Blinn-Phong power) instead of roughness when determining highlight intensity.

Unfortunately, there is no perfect conversion between these two things. Therefore, we ended up interpreting the roughness component as the reversed version of the older “specular map” value found in classic materials. It’s a noticeable change, but it’s a poor substitute for a true PBR shading model.

The last major missing piece to this puzzle is the normal map (the purple/blue/green image). In the start of the last assignment in the course we will have you incorporating the normal map to complete the classic material look. We will then wrap up the course by having you integrate a true PBR shader.

# Resources

If you want to be a programmer, you must learn to read (and eventually write) API documentation. Period. In this section I have included links to said documentation and some handy reference books. Have them open, use them.

## Vulkan API

<https://vulkan.lunarg.com/doc/view/latest/windows/apispec.html>

<https://www.khronos.org/files/vulkan11-reference-guide.pdf>

[ebooks.fullsail.edu (if the link does not work directly, copy it to your browser)](C:\\Users\\lnorr_000\\AppData\\Roaming\\Microsoft\\Word\\ebooks.fullsail.edu (if the link does not work directly, copy it to your browser)https:\\learning.oreilly.com\\library\\view\\vulkantm-programming-guide\\9780134464701\\)

[https://learning.oreilly.com/library/view/vulkantm-programming-guide/9780134464701/](C:\\Users\\lnorr_000\\AppData\\Roaming\\Microsoft\\Word\\ebooks.fullsail.edu (if the link does not work directly, copy it to your browser)https:\\learning.oreilly.com\\library\\view\\vulkantm-programming-guide\\9780134464701\\)

[https:/github.com/SaschaWillems/Vulkan](https://github.com/SaschaWillems/Vulkan) (will not transfer directly, but you can study the code for some insight)

<https://github.com/KhronosGroup/Vulkan-Guide> (nice overview of more specific resources)

## HLSL High Level Shading Language

<https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-reference>

*Note: The above docs often refer to Direct3D APIs. Modern Vulkan can also use the language. You should just study the syntax of the language when using it with Vulkan as other things like compiling are done differently.*

<https://shadered.org> (opensource HLSL & GLSL shader IDE, excellent for learning about modern shaders)

<https://docs.microsoft.com/en-us/visualstudio/designers/shader-designer?view=vs-2019> (Visual Shader Designer)

*Note: The VS Shader Designer is handy for prototyping complex shaders once you are more familiar with HLSL.*

## Gateware

We will be using this API occasionally throughout these assignments for simplicity’s sake. Gateware is a powerful cross-platform API often contributed to by students here at Full Sail just like you. (Designed for 3D Engine builders)

<https://gateware-development.gitlab.io/gcompiler/index.html> (Official Documentation)

*Tip: use the “--->” triple-dash operator on any Gateware proxy to have intellisense show you the actual arguments.*

# FAQ

* How do I know if I am using the Vulkan API correctly?
  + Aside from reading the docs and making sure the code compiles, we have enabled run-time debug output in the Vulkan API. Be sure to pay close attention to the console window when running the program. Any non-fatal mistakes you make will be reported by the Vulkan validation layer and printed there.
* Visual Studio doesn’t seem to be detecting the errors in my shaders, how am I supposed to code like this?
  + Carefully. Believe it or not it was not so long ago that things like intellisense, syntax highlighting and auto complete were not a common thing, especially in shader languages!
  + The way to know if your shader will compile is to… compile it! (right?) Shader languages must be compiled into machine instructions just like C++. If you study the code that loads the shaders you will see that compiling is part of that process.
  + Vulkan uses a binary intermediate language called SPIR-V that higher level shader languages like HLSL and/or GLSL must be compiled into. If there are any issues when converting your code to SPIR-V the **shaderc** compiler will note the error and I added code to print it to the console. Keep your eyes on it.
  + It *is* possible to have visual studio compile your HLSL code - but the output is not compatible with Vulkan, and it cannot compile Vulkan-specific features like push constants. Once your shaders get complex, I recommend using a dedicated shader IDE like [ShaderEd](https://shadered.org/).
* I have no compiler errors or run-time errors, yet nothing seems to be drawing. What do I do now?
  + Check over your code carefully to ensure you did not miss anything obvious such as having the wrong shader or geometry assigned to a pipeline. (or just setting up your vertex data wrong)
  + Problems like this can be difficult to track down, mainly because your C++ code cannot really see what is happening on the GPU. You can download a third-party tool called [RenderDoc](https://renderdoc.org/) to dig much deeper.
  + Once you have installed RenderDoc, in main.cpp uncomment the line "VK\_LAYER\_RENDERDOC\_Capture". This will allow RenderDoc to be attached to your program and capture data about it for a deeper look at what is going on in the API and the GPU itself.
  + If you are still lost, talk to an instructor. We can often point you in the right direction or help you make sense of the error messages you encounter until you get more comfortable dealing with them yourself.
* Is possible to do these assignments without Gateware? I prefer to do things from the ground up.
  + Technically yes, practically no. While someone (Derrick Ramirez) did originally have to write the Vulkan interface to Gateware, setting up a modern Graphics API like Vulkan or Direct3D12 from scratch takes a substantial amount of time. It’s just something we don’t have enough time for in a one-month course.
  + If you still really want to learn how to initialize a 3D API with no dependencies, there are plenty of online resources out there (including a few of my own) on how to do exactly that once you complete this course.