**Instructions**Provide the code, files that were modified and video recording when working with render doc for each applicable part in separate folders named accordingly (Part 1, Part2 etc.). Each separate part should work and provide the same image as shown at the end of each part below. Images below sections (1b, 1d etc.) also should match when that section is done. Add comments to the new code to which section of part it belongs if not specified already.

A white cat on a red background

Description automatically generated

### Task A1

Get started by just setting up your project. Copy your completed previous work (Work1 folder) source code and use it as this project’s base.

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.

***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

To help make the process of working with GPU textures a bit more straightforward, we have provided you with a utility header called **TextureUtils.h**. Add the file to your project and include it in the renderer.

Essentially this code allows you to transfer CPU texture data to the GPU. Use it to load the texture referenced by the first material in the model onto the video card and save the returned Vulkan objects. (which we will need)

### task A3

Step through the code and check that the texturing functions are all working/succeeding. Make sure to **destroy** all the newly allocated texturing resources you create during shutdown.

Unfortunately, The RenderDoc debugging tool only lets us inspect Vulkan resources actively in use. We will need to wait a tiny bit longer to 100% confirm the textures are being loaded properly.

#### 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 **VkDescriptorSetLayout** with a single texture/sampler combo. This is like setting up the uniform buffer, but we need only target the Fragment Shader and need to switch the type of descriptor in use.

### task b2

Allocate additional room in the existing **VkDescriptorPool** to reserve space to hold the texture/sampler descriptor.

### task b3

Create a new **VkDescriptorSet** specifically for holding texture descriptors. Use the function **vkUpdateDescriptorSets()** to write a **VkDescriptorImageInfo** referencing your newly uploaded texture.

### task b4

**Bind** your new **texture descriptor set** to the graphics pipeline alongside the existing one for the uniform buffer.

### task b5

During **VkPipelineLayout** creation, adjust the layout to include the additional **texture descriptor set layout**.

### task b6

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. (Works just fine in Vulkan & D3D11/12)

A screen shot of a computer code

Description automatically generated

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



You will notice the model is *very* shiny. Makes it look like it is made of plastic. We will fix this in the next two parts by integrating the “metallicRoughness” texture also included.

#### 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.

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

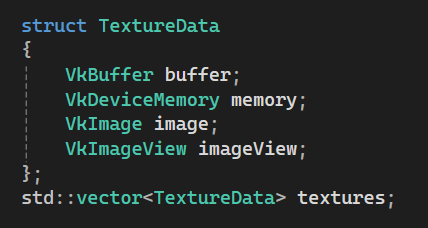
Enable **BINDLESS\_SUPPORT** in the GVulkanSurface. Adjust the Create(…) function in main.cpp to append the new graphics option **bit flag** in both Debug & Release modes. (Do not carry over the validation layer to Release mode)

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

### task c2

Convert the existing texture objects to an std::vector<> of handles.

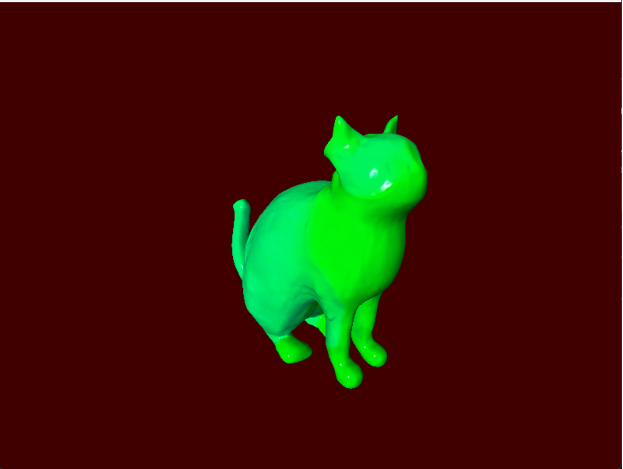
These changes will make adding multiple textures easy & convenient in the future.



Swap the existing texture code to iterate through the above array instead. Make sure everything still works after.

### task c3

Load the **metallicRoughness** texture (also referenced by the first material in the model) into the texture array.



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. (Again, don’t hard code)

Complete this section by pointing the texture descriptor set’s **AllocateInfo** “**.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

Verify that **both** textures are correctly loaded by looking in RenderDoc:

A screenshot of a computer

Description automatically generated

If you don’t see them, 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 in the Event Browser.

### task d1

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

A close up of a black background

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.

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 green light on 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. (If you generated your own AI model, it may be much more pronounced)

### task d3

Adjust the fragment shader to 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 with 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 fish with a black tail

Description automatically generated with medium confidence

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 channel to reduce ambient light where applicable.

The effect is quite subtle but notice how around the mouth & eyes (fish on the right) 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.