Assignment 2 Vulkan

# Introduction

In the final guided API assignment in this course, we will learn about importing external graphics data(3D models) onto the graphics card by carefully following the **rule of three**. Normally one would read a 3D model format using File I/O or use an [external API](https://www.assimp.org/) to access this type of data. In the interest of time, we will use a custom tool called **obj2header** to covert an **.obj** model file to a **header file** containing our 3D model data.

In addition to importing our vertex and index data, we will also import **material** data for use by our pixel shader. **Materials** are information representing how a surface is supposed to behave/react when interacting with light. We will learn how to import this data in the form of a **uniform buffer** and then have it respond to a directional light source with a specular component.

# Getting Started

## Preparing to use the Vulkan API

1. Download & install the latest graphics drivers from your laptop/video card manufacturer.
2. Download & install the Vulkan SDK for your platform: <https://vulkan.lunarg.com/sdk/home>
3. Reboot your computer. (Or type **taskkill /f /im explorer.exe && explorer.exe** into a command prompt)

## Use CMake to build your assigned API template

1. Open the directory containing this document in windows explorer and select the path bar at the top.
2. Type **cmd** into the bar and a command prompt should open. Type: **cmake -S ./ -B ./build** enter.
3. This should generate a solution inside a new folder. Open it and set it as your startup project.

# Assignment 2

## Part 1 | 25%

### Part 1a

Choose some colors you like(optional). Study the code and familiarize yourself where things are.

Use the **SetWindowName** function from **GWindow** to place your name and API variant at the top.

Shape

Description automatically generated

### Part 1B

In this assignment we will be loading in the **FSLogo.obj** 3D wavefront model into our application so we can draw it using the graphics card. Take the above file and **drag it into Visual Studio** or some other 3D model previewing software so we can get a good look at it.

Text

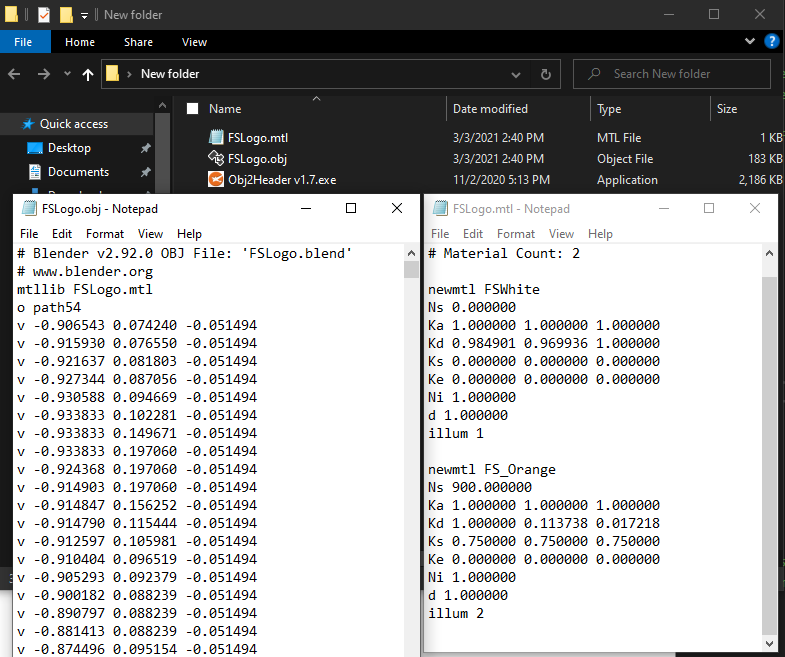
Description automatically generated

Make note that even though VS does not show the proper **materials** visually, it does read them in and does have their data in the properties. You can see that this model should have an **orange material** and a **white material**. By **looking at the values** you can tell that the **FS\_Orange material is quite shiny** with a lot of **specular**(Bounced) energy while the **FSWhite material appears matte** with mainly just **diffuse** (Lambertian) reflection from light sources.

A screenshot of a computer

Description automatically generated with low confidence

When **imported** into a more robust 3D modeling tool like [Blender](https://www.blender.org/) we can see the two **materials** appear correctly. This is what we will be aiming to replicate in our own application! But first we will need direct access to the data contained in the **.obj** and its companion **.mtl** file if we are to render it correctly.



Included with your model file is a tool of my own design called **Obj2Header.** Basically, it parses any nearby **.obj** file and its associated **.mtl** file and exports it to a convenient **C header file**. Simply run the executable in any folder containing wavefront models and watch it do its thing.

A screenshot of a computer

Description automatically generated with medium confidence

Take a careful look at the **generated header file**. You will see that it contains the familiar vertex and index data needed to render the model. However, if you **scroll to the bottom of the file**; you will also see that it contains **material** information read from the **.mtl** file as well! (We will need this in our pixel shader)

Finally make your way back to the actual source code and **include your new header file**. (Optional) If you would like the file to be **permanently included as part of the actual solution filter**, you will need to edit the **CMakeLists.txt** file and **rebuild** the project. (Look carefully, you will see the other source files. And yes, it is [possible](https://www.jetbrains.com/help/clion/cmakelists-txt-file.html) to put them all in a single list and use that instead)

### Part 1C

With the model data now available to us, we turn our attention to replacing our existing 2D NDC triangle with the new data. To do this we will need to abide by the **rule of three.** First, find the code where the triangle is currently being copied to a GPU **vertex buffer** and **replace the data** with all the vertex data from the **model header** file.

After doing this you will probably notice that your triangle has now disappeared!

### Part 1D

Let us keep in mind that out new 3D model contains **many triangles**, not just the one. Go to where the triangle is being drawn and adjust it to **draw** the **correct number of vertices** that are listed in the model data.

A picture containing chart

Description automatically generated

As you can see, **the current code is designed to draw 2D NDC triangles.** Though we are drawing the full amount of vertex information, the API does not understand that this data is meant to be used as a **full 3D model that includes an XYZ position, UVs and even normal data.**

### Part 1E

If you look inside the model header, you will notice all the vertex data comes in the form of an **OBJ\_VERT** structure. This is the **first part of the rule of three**, and we will need to match the rest of rules if we are to be successful.

Next, we will need to adjust the **vertex attribute descriptions** to correctly match the binary format of the **OBJ\_VERT** structure so the data is read in correctly. This will be the second part of the rule of three.

Logo

Description automatically generated

Its starting to look like something… but still very much off from what we are expecting!

***Note:*** *It is extremely likely that you will encounter validation errors in the console while doing this step. Be sure to resolve them all before continuing!*

### Part 1F

For the **last part of the rule of three** we need to adjust our vertex shader. Keep in mind the shader was originally written to draw a 2D NDC triangle. We will at least modify it enough so that the **correct data is coming in** and the **full 3D position is going out**.

Adjust the incoming vertex so that **all three components(position, UV & Normal)** are now received by the shader. Use your prior experience with the **HLSL shader language** to make these changes. Tweak the output data so that it uses the full **XYZ** component of the position, setting the **W** to its standard homogenous starting value.

A picture containing logo

Description automatically generated

Not that much of an improvement, still feels like we are missing something…

***Note:*** *It is extremely likely that you will encounter HLSL syntax errors in the console while doing this step. Use the output of the shaderc compiler to figure out what you are getting wrong with the syntax!*

***Tip:*** *Vulkan does not care what you call your* ***HLSL input semantics****, but you still need some!*

### Part 1G

Even though we seem to have the **rule of three** correct now it turns out we are still missing a **major** piece of the puzzle! Take a moment to go into the model header file and scroll past the **vertex array**.

Text

Description automatically generatedUh oh… Looks like we missed an entire section of data! **Index data** is a critical part of almost all 3D model files. This data is used during the **vertex assembly** process by the GPU to efficiently **reuse** existing vertices shared by multiple primitives (lines & triangles).

While it is possible to make and draw a shape without index data, GPUs are optimized to render with them and pretty much all 3D model files require it to be used one way or another. Thankfully, all modern graphics APIs can accept **index buffers** and draw using them.

We can make an **index buffer** easily by copying the code used to create the **vertex buffer** and adjusting it to take in the **index array** we just looked at. We will need new a **VkBuffer** and **VkDeviceMemory** to hold our indices. Do not forget to **release the memory** we allocated for these new objects at the end of the program as well.

***Note****: You just wrote a block of code that creates an index buffer. Sounds like a single responsibility to me! To keep your initialization code clean I recommend extracting it out into a well-named helper function.*

*I won’t bother you with more reminders since you had plenty of them in assignment 1. In any case, I recommend you continue to practice and grow your clean coding skills throughout this assignment. You will not be docked points for messy code, but clean code is easier to expand and debug. You will likely have an easier time with this assignment and future programming if you code cleanly.*

### Part 1H

With our **index buffer** now allocated and populated, we can use it to draw the model as intended. You will want to **bind** the index buffer to the API and switch to using a **draw operation** that supports **indexed** geometry submission.

If you do this correctly, the 3D model will **disappear**! This seems counterintuitive but remember that the without a **View Matrix** the camera is technically located at the **origin**. Because of this, it is very possible we are **inside** the 3D model and cannot see the back of it due to **back-face culling**. (Which all APIs have **ON** by default)

Graphical user interface, application

Description automatically generated

**(Optional Step)** GPU/API Debuggers like [**RenderDoc**](https://renderdoc.org/) are crucial for programmers writing graphics code. In the above screenshot I have used the tool to **inspect** the indexed draw call we just wrote. Even though the running program shows **nothing** but a grey screen, I can tell the geometry is **loaded correctly** because **RenderDoc** has a **visual inspector** that lets me look at any **geometry** we have already copied onto the card.

***Tip:*** *To enable RenderDoc capture and debug of the program, you should* ***uncomment*** *the* ***extension*** *in* ***main.cpp****.*

Now that we know the geometry is there, we can move on to making our vertex shader 3D just like we did in the previous assignment. However, before we do that it would be nice to just see our model on-screen. To do so, we can **temporarily** adjust the **vertex shader** so it **shifts all the Z coordinates by +0.75f and shifts the Y coordinates down by -0.75f.**

Text, logo

Description automatically generated

***Note:*** *I chose the(****0.75f****) numbers above out of experimentation and because I knew the 3D model was small and created around the origin. This will not work for any model and is just temporary so we can feel good about seeing something. It is no substitute for writing a real 3D vertex shader, which is what we will be doing later.*

## Part 2 | 50%

### Part 2a

In this section we will **create the matrices and other data** required to render our model in 3D. We will then **copy this data to the GPU** so it can be **accessed** directly by our **vertex** **and** **pixel** **shaders**.

In the previous assignment you learned how to use **Gateware**(or some other math library) to build a **World**, **View** and **Projection** matrix for use in 3D. Use that prior experience to create these matrices with the following properties:

**World:** *An identity matrix that slowly* ***rotates*** *along the* ***Y axis*** *over* ***time****.*

**View:** *A camera positioned at* ***0.75x +0.25y -1.5z*** *that is rotated to look at* ***+0.15x +0.75y +0z****.*

**Projection:** *A vertical* ***field of view*** *of* ***65*** *degrees, and a* ***near*** *and* ***far*** *plane of* ***0.1*** *and* ***100*** *respectively.*

Additionally, we are going to need some variables to represent a **directional light source** shining on our 3D model:

**Light Direction:** A light shining forward with a strong tilt down and to the left. **-1x -1y +2z** (normalize)

**Light Color:** The light is almost white with a slight blueish tinge. **0.9r 0.9g 1.0b 1.0a**

***Tip:*** *Vulkan has a unique NDC which requires a* [*unique projection matrix*](https://vincent-p.github.io/notes/20201216234910-the_projection_matrix_in_vulkan/)*. Gateware has functions that can directly accommodate for this if you wish. (Or just make it on your own)*

### Part 2B

Vulkan has **three primary ways** to communicate **variable data** to running shaders: **Push Constants**, **Uniform Buffers** and **Storage Buffers**. Push Constants are an extremely convenient and easy way to move CPU data to the shaders in a draw call. Unfortunately, they have a serious limitation. Graphics cards are only required to support **128bytes** of data via this method. Essentially this is enough room for two 4x4 float matrices and nothing else.

In addition to matrix variables, we are going to want to upload the information about **materials & lights** to both the vertex shader and pixels shader so we can correctly visualize this model as intended. For this reason, we are going to use **Storage Buffers**. Though not quite as efficient, they also **do not have the harsh size limitations** of Push Constants or Uniform Buffers.

**Uniform buffers** are a compromise between Push Constants and Storage Buffers. They are very efficient like Push Constants but also hold much more data. (Up to 16KB guaranteed, often up to 64KB) Still 16KB is nothing compared to the **gigabytes** a storage buffer can hold, and they lack the simplicity and convenience of Push Constants.

Before we dive into their creation lets **organize the data** we intend to send to our shaders. Something like the following should do the trick:

Text

Description automatically generated

***Tip:*** *While the* ***FSLogo*** *model only has* ***two*** *sub-meshes, MAX\_SUBMESH\_PER\_DRAW is set much higher. This can account for more complex models. There are ways to organize this data to avoid hard coding if you like.(ask us)*

You will also need to **mirror this structure in both HLSL shaders**. Be sure to match the order and size of the variables using the language’s built-in types. Since there is no **OBJ\_ATTRIBUTES** type in HLSL you will need to **make your own**, again mirroring the **size and order** of the data in the C++ struct.



Complete this step by making an **instance** of your new C++ struct and **initializing** it to all the math variables we made at the start of this process. Init the initial **material** to the first material entry in the model’s **material array**.

***Note:*** *Though you could represent them with a* ***64bit double*** *type, I chose to ignore the material string pointers in HLSL since they are only theoretically used during initialization. Therefore, structs only contain* ***OBJ\_ATTRIBUTES.***

### Part 2C

Now that we have the data required to draw our 3D model created, we will need to get it onto the GPU. We can do this by creating a **storage buffer** which is very similar to the vertex and index buffers you already created.

There is one catch however… because this buffer will be **updated each frame**, we must take care to **avoid synchronization issues** between the GPU and CPU. Unlike older 3D APIs like D3D11 and OpenGL, in Vulkan this **synchronization is not handled for you** automatically.

That is the bad news, the good news is that the Gateware template is already **synchronizing on a per-frame basis**. What this means is that if we **create a storage buffer for each simultaneous frame** that can be processed, we will **not run into any sync issues** as each in-flight frame will have its **own storage buffer** it can use directly.

Using the **same types you used for vertex and index buffers**, go ahead and allocate a **std::vector** of each for your **storage buffers**. In the next part we will **resize()** them based on the **maximum number of frames** reported by **GVulkanSurface**.

### Part 2D

Once you determine the **maximum number of active frames\***, use a **for loop to initialize all** the **storage buffers** the exact same way. Again, you can pretty much copy how this is done for the vertex and index buffers, however we will change the data going in to be a copy of the struct you initialized from the end of step **2B**. (Do not forget to also adjust the buffer size and USAGE\_BIT to match)

Even though we made multiple buffers, in the next section we will focus on just linking the first one to the pipeline.

To finish this step, we will also be sure to **free** these new **storage buffers** on program completion.

***\*Tip:*** *Vulkan refers to frames displayed to the end user as swap chain images.*

### Part 2E

Buckle-up, this is where Vulkan starts to get a bit more gnarly than the simpler APIs. Next thing we need to do is describe to the existing **VkPipelineLayoutCreateInfo** that it will be using a descriptor set to supply external data to the shaders(in this case our storage buffer).

The interface that does this is called a **VkDescriptorSetLayout**. To make one you need two things, a **VkDescriptorSetLayoutCreateInfo** which itself needs a **VkDescriptorSetLayoutBinding**.

The VkDescriptorSetLayoutBinding should only have 1 descriptor as that is all we need for now. The **type** of descriptor should be used for **storage** buffers. The **stage** it is assigned to should be the **Vertex & Fragment**(Pixel) shaders, as that is who needs this data. Don’t forget to fill out the other values even if we are not using them.

Next, we make the VkDescriptorSetLayoutCreateInfo which tells Vulkan how many bindings we have and where they are. The rest of the arguments can be set to nullptr or whatever their required defaults are. (Read the docs)

Ok… now we can finally call **vkCreateDescriptorSetLayout**. Add a permanent handle to a VkDescriptorSetLayout in your class, we will need it so we can **free its memory** at the end of the program. Speaking of, go ahead and take care of that now once you have created it.

The final step in this section is to tell the existing **VkPipelineLayoutCreateInfo** that you have a usable descriptor set layout now.

### Part 2F

Well, all of that was just to tell the pipeline “Hey! Descriptors are coming!”. Now we need to supply said external descriptors. These external descriptors are called **VkDescriptorSet**(s), but before we can make one, we need something called a **VkDescriptorPool**.

“Pools” are how Vulkan efficiently reserves memory on the video card, there are many different kinds. A VkDescriptorPool is used to reserve descriptor sets, **add one to your class** and add code to **free it** during clean-up. You can then create one using **vkCreateDescriptorPool**.

To do this you will need to supply a **VkDescriptorPoolCreateInfo**(starting to see a pattern here?). We need a very shallow pool, so it should only take 1 descriptor and no special flags other than the required defaults. Remember, we are trying to link our new **storage buffer\*** to the shaders. Set the **VkDescriptorPoolSize** it wants appropriately.

\****Note:*** *Again, just focus on using one buffer for now. The others will be used later to solve any sync issues.*

### Part 2G

Are we there yet? I know, I know… this is getting a bit absurd, but the good news is once you do this once; you pretty much have a blueprint on how to upload most any non-geometry resources to Vulkan.

Anyway, our new **VkDescriptorPool** can actually allocate some of those **VkDescriptorSet**(s) we have been wanting for a while now. To do this we can use the function **vkAllocateDescriptorSets**. Before we do so, lets add an actual VkDescriptorSet to the class. You **do not have to free** the memory for the descriptor set as it is part of the descriptor pool, though it is possible to do so if you need to.

Like most things in Vulkan, you must describe the thing that you wish to create/allocate in the API. In this case you do so using a **VkDescriptorSetAllocateInfo** structure. Thankfully, the arguments to this structure are self-explanatory at this point, use the docs to fill them out.

### Part 2H

So, the good news is we have everything we need allocated now. The bad news is none of it knows about each other. To correct this issue, we will start by linking our new **VkDescriptorSet** to our **storage buffer**.

To do this you will need a **VkWriteDescriptorSet** and a **VkDescriptorBufferInfo** structure to describe what you are trying to do. Filling their members out is obvious for the most part. Keep in mind, we are connecting one **storage buffer** and we do want access to **all** of it.

Once you have filled everything out use **vkUpdateDescriptorSets** to tell the **VkDevice** to link them together.

### Part 2I

Yes… this is the last part; and yes, we will finally see our model in 3D after this! There are only two things left: **Connecting** the descriptor(s) to the command buffer and finally **using** the model’s matrix data in the vertex shader.

First, inside the **Render()** function we will **connect** the descriptor set to the command buffer using **vkCmdBindDescriptorSets**.It will ask for many of the items we created over the last few sections.

Finally, we should be able to access and use the data in our **storage buffer**.In the **HLSL vertex shader** make the following changes:

* Add a **StructuredBuffer<>** to the code so we can access the **storage buffer**(code example in **2B**).
* Define  at the top of the vertex shader. (HLSL is **column major**)
* Use the **mul** [intrinsic](https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-intrinsic-functions) to multiply the **outgoing position** with  to move the model’s local vertex into **world space**. Then do the same thing again using the **View** and **Projection** matrices also stored in the **StructuredBuffer**.

Text

Description automatically generated

Wow! its **3D** now, just a bit upside down it seems. Thankfully, this is an easy fix. Currently when the **HLSL** shaders are **compiled**, there is a setting that asks the shader compiler to **invert the y axis** for any shader code it generates. The reason this was enabled in the template by default is so that **NDC** for Vulkan would be the same as it is for other APIs like D3D11 and D3D12.

In theory this allows you to use the **same projection matrix as you would for Direct3D**. The benefit for game engines is that the game code will not have to switch based on what rendering API is in use. Because we are using a real Vulkan projection matrix in step **2A** we need to turn this feature **off**. (Alternatively, you could use a DirectX style projection matrix instead) Once you do, you should get the following:

Logo

Description automatically generated

That was quite a journey, but our shader code now has access to the model’s transform and material data! As you can tell Vulkan is a very verbose API that basically gives you control over every little detail. This can make learning it a bit overwhelming at first, but as you get better with the API you may grow to appreciate the extra level of control & performance it gives you over the GPU.

In the last two sections, we will focus on using our new material data to **visually enhance** the 3D model, so it **appears as the artist had originally intended**.

## Part 3 | 75%

### Part 3A

First thing we should do is get rid of that hardcoded red color and **replace it with a color from our actual materials**. To do so, copy the **HLSL** code related to **materials** and the **StructuredBuffer<>** from the vertex shader and make sure it also available to the **pixel shader**.

**Replace the outgoing color** with the **diffuse color** from the **first material slot**. It should just be a solid white color.

Text

Description automatically generated

This appears to be the **correct color of the Text** of the Full Sail Logo. Unfortunately, the **Logo itself** should be closer to an **orange** color.

### Part 3B

To solve this, go back to part **2B** and instead of just copying the very first material, **loop through all the materials**. Assign each of them to materials stored in the **SHADER\_MODEL\_DATA** structure so we have complete access.

So… how do we correctly draw this model so that the Text is white, and the Logo is orange? Or to be more precise, how do we draw each **mesh** based on its **material attributes**? The good news is that how the model is split-up is already outlined in the **obj** file:

Text

Description automatically generated

We will now **adjust our drawing code to draw each mesh individually** instead of drawing the entire model all at once. In theory, this gives us a small window to switch the material used before drawing the next mesh. **Make a loop** to iterate across all the available meshes, drawing **only the indices listed in each mesh**. If you do this correctly **everything should look the same**. The key difference being that drawing has now been split into multiple submissions.

### Part 3C

Ok… so now we have separate draw calls for each mesh. Now we need a way to not hard-code the specific material that is supposed to be being used for a given mesh. Given that this info is just a small ID number/integer and needs to change rapidly between draws, it seems like the perfect application for Vulkan **push constants**.

Thankfully, unlike storage or uniform buffers **push constants** are minimal in setup. We do need to make some minor changes to the setup code, but they will be quick. Start by tweaking the **VkPipelineLayoutCreateInfo** to hold at least one **VkPushConstantRange** and initialize it to target both the **vertex** and **fragment** stages with enough room to store at least **one unsigned integer**. (Our mesh/material ID)

***Note:*** *Another approach may be to create a secondary vertex buffer that simply contains a material ID per-vertex. This takes much more memory but allows you to draw multiple meshes at once with differing materials. (very fast!)*

### Part 3D

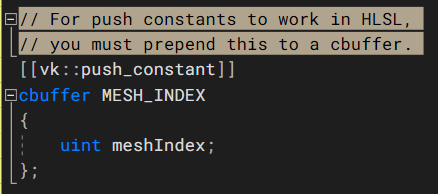
Now the pipeline is aware that we can send it **push constants**. Make your way to the mesh drawing loop you wrote earlier and use the function **vkCmdPushConstants** inside the loop to send each mesh’s **materialIndex** to the shaders right before calling **draw**.

Make sure you target **both applicable shader stages** and send the **correct number of bytes**. In theory the **materialIndex** is now available to a shader during the next draw call. Setup is quite short overall, especially compared the amount of steps **storage** and **uniform** buffers need!

***Note:*** *Push Constants are awesome, just do not forget they are only promised to be* ***128 bytes*** *or larger. ☹*

### Part 3E

Ok, this is the step we have been building towards. We will use the **push constant** data we just uploaded in our fragment/pixel shader. To access it in **HLSL** you will need to make something called a **cbuffer** (constant buffer) which is very similar to a **struct** but is specific to both **uniform** data and can also be used with **push constants** when pre-fixed with a **special attribute**:



Once you do this, the bytes of whatever you uploaded via **vkCmdPushConstants** should **overwrite** the contents of this **cbuffer.** You should be able to now **modify your HLSL code** to use the **index** to access the **correct material** for a given pixel:

Logo

Description automatically generated

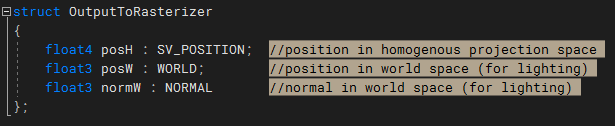
Looking good! Now we not only have access to all the data for the model using a **storage buffer**, but also a simple and clean way to select which data to use during a draw via **push constants**.

## Part 4 | 100%

### Part 4a

After you get to grips with how to upload and access static data with a graphics API; you then get to the fun part, **playing with shader code!** The 3D model looked much nicer in **Blender** earlier because it was **applying a light source** to the model and **using the given materials** to tune how each surface **interacts** with that light.

We will start by applying a basic **directional light source** the surface of our model. To make this possible we will need to **output a world space normal** from our vertex shader for use in the **pixel shader**.



In the **vertex shader** make a **struct** to that can be used to **output more than just the position**. (Optional) You can also declare this struct in the pixel shader if you would rather work with it instead of taking in the parameters separately. (Either way can work if the proper semantics/types are used)

***Note:*** *I did not include a* ***UV coordinate*** *in the output struct since we won’t require it. However, you might as well add one since the* ***input*** *has one available, and you might end up using this code in your* ***Level Renderer****.*

### Part 4B

Now **adjust the output of the vertex shader** so it will **return** the new structure you defined in the last step. You will need to fill out each member of the structure based on the input values. Do not forget to **transform the outgoing normal into world space** since our lights are also defined there.

You will also now **adjust the arguments of the pixel shader** so that they correspond the **exact type and semantics** now being provided by the vertex shader’s output. After you do this, everything should **still compile and draw** like it did before. (We will use the new data in the following steps)

### Part 4C

Now we should have everything we need to apply a **directional light formula** to each of our pixels. Assuming you did not memorize this formula, it was covered on **CGS day 7**. Use the **diffuse color** of the **material** as the **surface color** and our new **normal** to compute the **amount of light** scattering from the surface. Remember to also multiply by the **color of the light** itself. (This is called **Lambertian** shading)

Logo

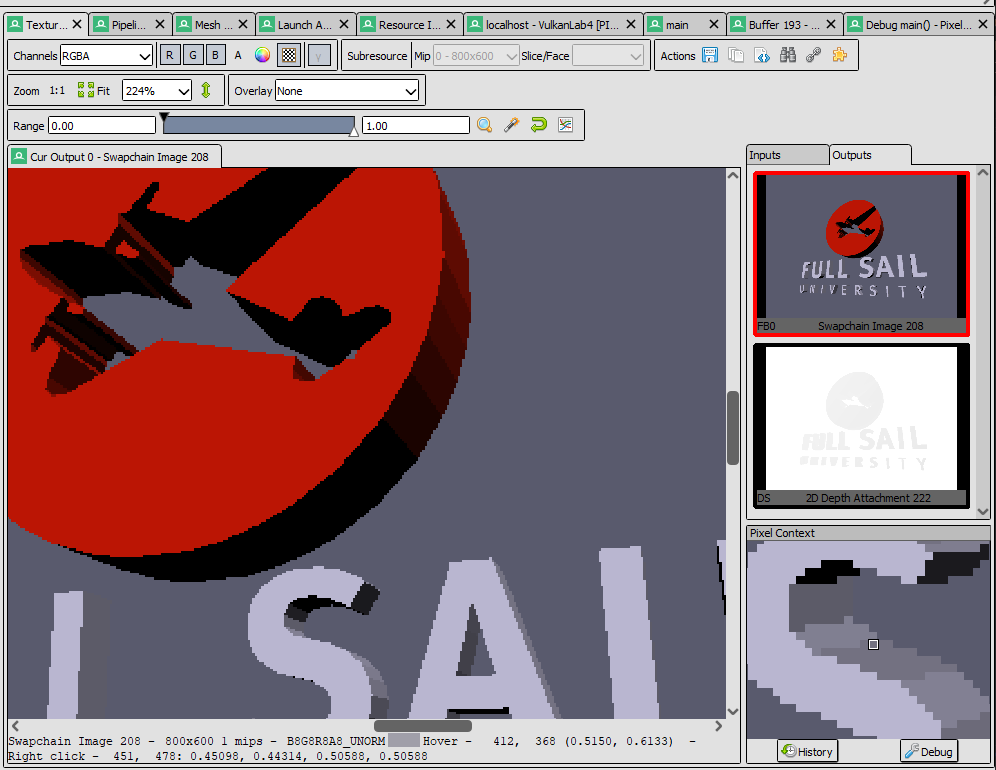
Description automatically generated

***Tip:*** *Normal data coming from the* ***rasterizer*** *has been* ***interpolated*** *across a primitive. This means it* ***may no longer be normalized*** *when it reaches a particular pixel. Thankfully, there is an* ***HLSL***[*intrinsic*](https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-intrinsic-functions) *you can use to renormalize it.*

### OPTIONAL

If you are struggling to complete this part or any of the later parts its probably because you are not **debugging your shader code**. Writing working shaders is all about getting the math right. If you cannot inspect what is going on in the code, then finding success may prove elusive.

The good news is that **debugging your GPU shaders is possible!** If you run a **capture** in [RenderDoc](https://renderdoc.org/) it is possible to **inspect and debug individual pixels:**



Select your **draw call on the left side** and open the **texture viewer tab**. Select the **Swapchain Image** and then **right-click the pixel** you wish to inspect. Once you do so a **Pixel Context** window will be populated. From here click the **Debug button** and you will be able to follow exactly what happened when that pixel was drawn. (Both raw source and disassembly views are available)

### Part 4D

In Part **2A** you created a **Y rotation matrix** that slowly rotated **over time**. It is now time to put that matrix to use! Go ahead and **set** the **second world matrix** in the **SHADER\_MODEL\_DATA** structure to be **equal** to this constantly **rotating matrix**. (This matrix will be used for the top mesh of the Logo)

Now that the data has been **updated**, we need to **copy it** to the **storage buffer** associated with **this frame**. To do this you can use the helper function **GvkHelper::write\_to\_buffer** which will allow you to **send the updated structure data** you just modified to the **current storage buffer** for use in the shaders.

### Part 4E

Now that we have uploaded our rotating matrix to the **storage buffers** its time to use it in the shader code! Go to the **vertex shader** and instead of always using the first world matrix ([0]) use your mesh/material ID from **Part 3E** to pick the correct matrix for a **specific mesh**.

Logo

Description automatically generated

After you do this, you should see the top part of **Logo spinning,** but the **lighting will not update** correctly yet. To resolve this, go to shader code that **transforms** your **normal** into **world space** and **adjust it appropriately**.

### Part 4F

Depending on what graphics card you have you **might** see some flickering/hitching while the logo is rotating. This is due to us currently **only using one** of the **storage buffers** we have for multiple frames. Earlier we made sure to create a separate storage buffer for each rendered frame to avoid such synching problems. Unfortunately, when we created our **VkDescriptorSet** earlier we only made one. Let’s correct that oversight now.

Start by replacing your single **VkDescriptorSet** with a **std::vector** of them. The idea is that we will have one VkDescriptorSet for each **storage buffer** you made in **Part 2C**.

Make your way over to **Part 2F** where you made your **descriptor pool**. Adjust the **size** of the descriptor pool and the maximum number of **sets** to have enough room to have **one descriptor for each rendered frame**.

Now we can go back to **Part 2G** where we **allocated** our **VkDescriptorSet** and adjust the code to allocate a descriptor set for each frame instead of just one. (***Hint***: call **vkAllocateDescriptorSets** multiple times)

Starting to see a pattern? Finish setup by going to **Part 2H** and **vkUpdateDescritorSets** so that each of our **descriptor sets point to its corresponding storage buffer**. (Ex: descriptor[0] -> storagebuffer[0] etc.…)

Now we can select and **bind** the **appropriate descriptor** and storage buffer while rendering instead of being forced to share the first one. Find the code in the **Render function** that does this and adjust it as needed.

***Note:*** *Even if you are not experiencing sync issues right now, if you don’t adapt your code to compensate you will experience them at some point when things get more complex. Better to avoid such issues early on.*

### Part 4G

The final part of this assignment will just have us cleaning up and enhancing the lighting effects. We will start by finding the section in the code where we our **shader variable structure** is and **adding two new vectors**.

We will need an **Ambient** component to our directional light source, (I called mine **sunAmbient**) and we also need to know where our **camera’s position is in world space**. (Ex: **camPos**) The former will be used in inject **indirect** or bounced light into the scene, while the latter will be used to compute the amount of **reflected light** bouncing off our model’s surface.

Logo

Description automatically generated

The initialization of the camera’s world position should be self-explanatory; however, our sun’s ambient term should be set to **25% red 25% green and 35% blue** indirect light. **Use this new variable in the pixel shader** to compute the **total** amount of light striking a pixel before multiplying it by the **surface color**. If you do this correctly it should look like the above picture. (*If you need a refresher on the ambient term, again check CGS day 7*)

The last step in our journey is to use the **camera’s position** to calculate the **specular reflection** or bounced light coming off the surface from the light source. Use the formula provided on **CGS day 7** to create the highlights shown below. Take note that many of the arguments used in this formula will be pulled directly from the mesh’s **material properties**.

Logo

Description automatically generated

(Optional) instead of using the classic **half-vector method** provided in the slides, you can instead compute the exact vector reflected from the surface and compare that to your view vector. This will get you a much cleaner and more accurate specular reflection as shown below:

Logo

Description automatically generated

***Tip:*** *HLSL has the* ***reflect*** *intrinsic built directly into the language. It has many useful applications in graphics.*

# Summary

Nice work! Using Vulkan for the first time to immediately draw 3D model is no small feat. From here its time to start thinking about which of the 3D hardware APIs you want to use for your Level Renderer. They all have advantages and disadvantages. APIs like Vulkan and D3D12 are more complex but also more efficient and more in-demand on a resume. APIs like D3D11 and OpenGL are older/simpler but still commonly used, and often are plenty good enough if you don’t need bleeding edge performance and features. (Ex: 2D/Mobile/Cell Shaded games)

It is important to note that most modern real-time 3D programs use a shading model called **PBR (Physically Based Rendering)**. If you wish your graphics applications to have the same level of fidelity seen in many modern games; I highly recommend you read some [articles](https://marmoset.co/posts/basic-theory-of-physically-based-rendering/) on the topic and check out some [sample PBR shaders](https://github.com/Nadrin/PBR). (The math is quite complicated, but it is not 100% necessary to understand all of it to make use of it)

# 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 platform abstraction API 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

* Your example looks less jagged than mine and does not stretch or skew weirdly when I resize the screen?
  + When I completed the sample, I added the **MSAA\_8X** flag to GVulkanSurface.Create(…). This enables smooth **anti-aliasing** of the rendered polygons. I fixed the distortion of the screen by simply **recalculating my projection matrix** each frame much in the same way you did in **assignment 1**.
* 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 am struggling to complete Part 1. Any additional places I can look to help figure out what may be wrong?
  + Part 1 heavily involves changing your vertex structure/format being passed to the GPU. This directly impacts something mentioned on day 1 called “The Rule of Three”. I have created a document specifically tailored to help you find where these mismatch issues may exist. You can find it in the day 2 handout.
* I am really lost on part 2E and the steps after, Vulkan Descriptor Sets are not making sense to me. Help?
  + Descriptor Sets are without a doubt one of the most challenging parts of the Vulkan API to understand. I included some sample code with this assignment showing how they are used to attach a uniform buffer to the vertex and pixel shaders. Studying this code should help you get through this section more easily.
  + The first 5 minutes or so of [this video](https://youtu.be/d5p44idnZLQ) is an excellent visual break down of Descriptor Sets in Vulkan. If you are struggling to wrap your head around what is going on I highly recommend giving it a quick watch.
* 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 write the Vulkan interface to Gateware, setting up a modern Graphics API like Vulkan or Direct3D12 from scratch would quickly turn this into a full-blown Project. Unfortunately, we only have time for one of those this month. ☺
  + 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.