LAB 4 Vulkan

# Introduction

In this lab, 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 convert 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 add this data to our **uniform buffer** and then have it respond to a directional light source with a specular component.

Drawing dynamic objects in a 3D game requires the ability to efficiently submit whatever needs to be drawn during a given frame. For Example: Enemies may need to be removed or added, certain objects may be behind the camera and can safely be ignored, etc. We can solve this problem by creating a list/buffer that has instructions on what to draw at a given moment.

We will use something called an HLSL StructuredBuffer to hold this information. This data could be variable in size and very large in a game world. Therefore, we will use a Vulkan **Storage Buffer** to hold this information in a dynamic way that won’t require hard coding the size. We will adjust our **Descriptor Sets** to hold this extra buffer.

# Getting Started

The method of getting started with this and most future labs should be identical to the first lab assignment. The main difference is that you now have Vulkan installed, so reinstalling it should not be required.

However, all the other steps still matter. In particular: Cloning this repository so your progress can be saved. If you don’t remember all the steps, please review the getting started section from Lab 1.

This lab assignment will lean on a lot of the Uniform Buffer & Descriptor Set code you wrote in lab 3. It may be useful to have that code handy later in this assignment for quick reference.

# Assignment 2

## Part 1 | 25%

### Part 1a

Choose some colors you like(optional). Study the code and familiarize yourself with 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

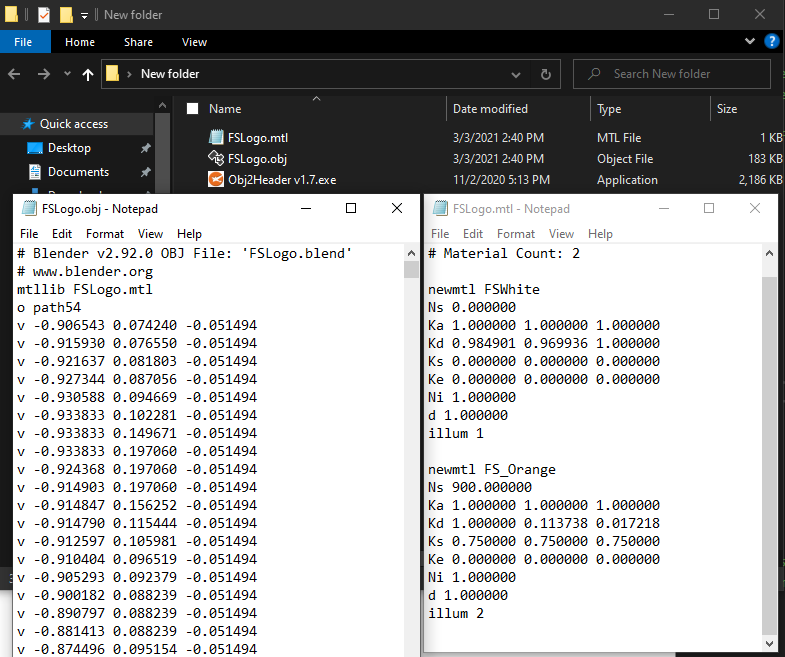
Description automatically generated

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**.

### 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 our 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

It’s 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 lab 3. 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 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:*** *When launching the debug .exe through RenderDoc, make sure to set the working directory path 1 folder up.*

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.*

## LAB CHECKPOINT | MANUAL COMMIT | DO NOT SKIP

To receive credit for your assignments, you must manually commit to your Repo 4 times (Once at the end of each Part/Section). If you skip this commit (or disable auto commits) you will not receive credit for the previous section.

1. Change the lab’s Title Bar so it says where you are in the lab.
   1. For Example: “John Smith – Lab 1 – Part 1 Complete”
2. Take a Screenshot of the lab window running showing your work. (Windows Key + Shift + S)
3. On Windows the screenshot is in your clipboard, open MS Paint and (Ctrl + V) to load it in.
4. Save the image with the same name as your title bar in the root folder of this repository.
5. Open GitHub Desktop for this repo. You should see the new image as a pending change.
6. Create a commit message with the same name as the title bar & commit your changes.

If this section is not perfect but you need to continue forward, you must still do this to get partial credit!

## 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:

**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

Create a C++ structure designed to transfer the information you created in the last step, to the GPU.

A screen shot of a computer screen

Description automatically generated

Notice the absence of a World Matrix this time around. Along with materials, a world matrix is a unique matrix and not meant to be shared with everyone. While we could have made a fixed array of them like we did in lab 3, there are better and more flexible/efficient ways we will use later instead.

Create an instance of this structure in your class and transfer the global matrix/light information inside. Alternatively, you could just use this structure to replace any previous individual variables instead.

### Part 2C

Create an equivalent to this structure in HLSL in both your Vertex and Fragment shaders. Take care to ensure that the order and sizes of the variables are the exact same so that the raw memory copies over correctly.

The type of this structure should be a **cbuffer** since we are going to be using it as a Vulkan **Uniform Buffer**.

### Part 2D

Ok, so the bad news is that this situation calls for a uniform buffer, descriptor sets, and everything else that entails.

The good news is that you have already done this. And what we need to do here again is hardly any different.

Copy your code from **lab 3** that creates or is related to creating or initializing all the following things:

**VkDescriptorSetLayout**, **VkDescriptorPool**, **VkDescriptorSet**

The only real difference this time around is that instead of setting up the **uniform buffer** to hold **SHADER\_VARS**, we are going to have it hold our **SHADER\_SCENE\_DATA** instead.

If your lab 3 code is properly setup to do an std::vector<> of **VkDescriptorSet**(s) for GPU synchronization, then be sure to bring that code over as well. This will provide a template for how to do this with a **storage buffer** later.

***Tip:*** *If you did not fully or correctly complete lab 3, now is the time to do so.*

### Part 2E

Just like last time, to make the geometry draw properly in 3D there are two key steps.

The first is to **connect** the proper **descriptor set** to the active **command buffer** using **vkCmdBindDescriptorSets**.

### Part 2F

The second step is to use the View and Projection matrices inside the Vertex shader to transform the positions.

Text

Description automatically generated

Ok! It’s **3D** now, just a bit upside down it seems. Thankfully, this should be an easy fix.

### Part 2G

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. (This is also how NDC was covered in CGS)

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

Our shader code now has access to global View, Projection & Light data (which we are not using yet). However, we currently have no way to customize the different/unique parts of the model.

In the next section, we will focus on making specific material & transform data available to each separate piece of the 3D model (and any duplicates of those pieces). This will allow for three new capabilities:

1.) Allow us to adjust the color/material of each sub-mesh.

2.) Give full translation/rotation/scale control of each sub-mesh.

3.) Provide the capability to add or remove more copies of a mesh with unique transforms & materials.

## LAB CHECKPOINT | MANUAL COMMIT | DO NOT SKIP

To receive credit for your assignments, you must manually commit to your Repo 4 times (Once at the end of each Part/Section). If you skip this commit (or disable auto commits) you will not receive credit for the previous section.

1. Change the lab’s Title Bar so it says where you are in the lab.
   1. For Example: “John Smith – Lab 1 – Part 1 Complete”
2. Take a Screenshot of the lab window running showing your work. (Windows Key + Shift + S)
3. On Windows the screenshot is in your clipboard, open MS Paint and (Ctrl + V) to load it in.
4. Save the image with the same name as your title bar in the root folder of this repository.
5. Open GitHub Desktop for this repo. You should see the new image as a pending change.
6. Create a commit message with the same name as the title bar & commit your changes.

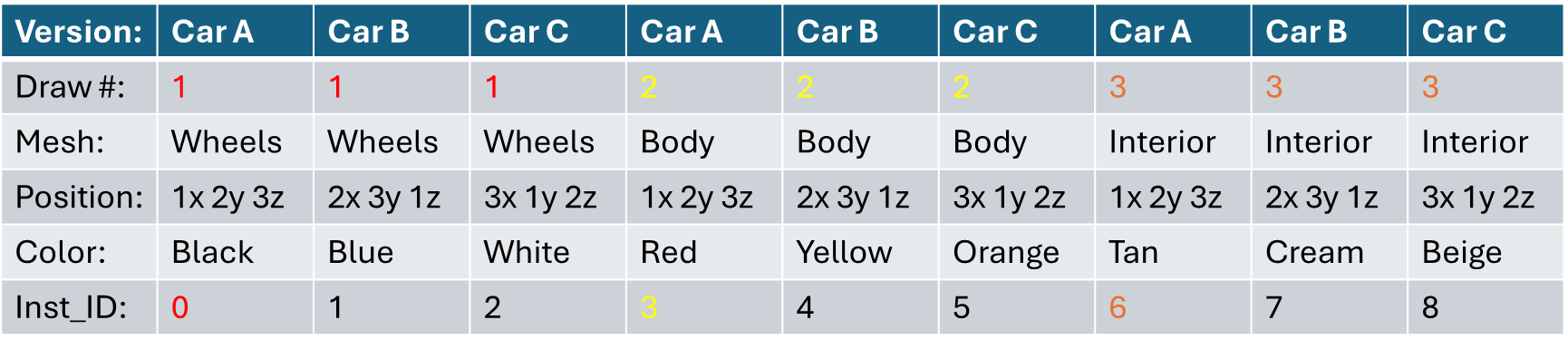
If this section is not perfect but you need to continue forward, you must still do this to get partial credit!

## Part 3 | 75%

#### OVERVIEW

In the last section we added a **uniform buffer** to hold the global data for our scene. In this section we will add a way to customize the world matrix and surface material of each instance of a mesh that we draw.

The overall idea is that at the time of drawing a specific frame, we know which 3D models (and each sub-mesh) we need to render. We then organize each visible item into a sequential buffer, where each element represents where something is and how that copy should look.



This is a significantly more efficient and flexible setup vs. simply hardcoding multiple vkDraw calls sequentially. It will involve some prep-work however, as well as expanding the use of shader buffers and descriptor sets.

Though we technically know exactly what we want to draw in this assignment, video games constantly need to add and remove what is in the game and even what the player can see at a given moment. Therefore, learning how to manage large sets of draw operations efficiently will be a crucial skill. (Particularly in the classes after this one)

### Part 3A

First things first, we need a C++ data structure to represent each drawable instance of a mesh. Something like this ought to do the trick:

A black background with white text

Description automatically generated

Since our FSLogo.h model contains two meshes, go ahead and create an std::vector<> of the above type. Then loop for the model’s mesh count, pushing back an identity matrix and the correct material for each mesh.

***Tip:*** *I called my std::vector<> “****perFrame”*** *even though we just fill it once. This is just to reinforce its true purpose.*

### Part 3B

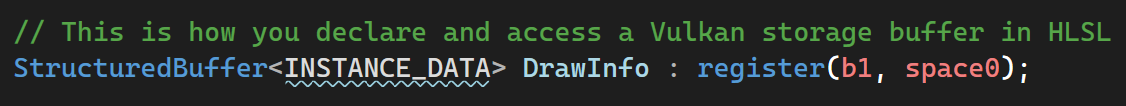
Now that we have the C++ version, we will need an equivalent in HLSL. As you might expect, representing the world matrix is easy enough, however there is no direct equivalent to the **OBJ\_ATTRIBUTES** structure.

This means you will need to study the C++ one provided and roll your own in HLSL. As always when doing this conversion, be extra careful with the size, type and order of the variables.

Place the new structure in both the Fragment and Vertex shaders since they both will need access to the data.

### Part 3C

Due to the dynamic nature of our approach, we want to avoid hard coding the size of the vector/sequential buffer in the shader code. In HLSL the way to represent dynamic storage is through a **StructuredBuffer<>** type.



Looking at the syntax, you can see it looks very similar to an std::vector<> C++ type. Go ahead and add a Structured Buffer to both of your shaders. That way we can access them later, once they are connected to a vkBuffer.

### Part 3D

Now we have a way to access the instance data in the shader, we also have that data represented in C++ via a populated std::vector<> container. What we DON’T have is a way to upload said container to VRAM. For the global variables, we were able to use a **cbuffer** in HLSL connected to a **uniform buffer** in C++. That isn’t going to cut it for the Structured Buffer unfortunately.

As you can see from the HLSL comment in the last section, the primary way to represent a Structure Buffer in C++ is through something called a Vulkan **Storage Buffer**. So, the bad news is that we need a new type of buffer. The good news is that a **storage buffer** is nearly identical to a **uniform buffer** on the C++ Vulkan side.

To add storage buffers to your code, do the following:

1. Create an std::vector<> of vkBuffer(s) just like you did for your uniform buffers.
2. Ensure they have enough room to hold your instance data and then write it to the buffers.
3. Free their memory at the end of the program.
4. EXTEND your **VkDescriptorSetLayout** to have another binding for the storage buffer.
5. EXPAND your **VkDescriptorPool** to have enough room for your new set of buffers.
6. WRITE the handles of your new storage buffers to the **VkDescriptorSet** of each frame.

Once you complete these steps, then in theory your instance data should be accessible by your shaders. However, we will need to split-up our drawing code to be per-mesh type if we are going to make use of this new data.

***Hint:*** *This section will truly test your understanding of Descriptor Sets. Read any API debug/error output carefully!*

### Part 3E

Once the **storage buffer**(s) has been integrated and connected to the pipeline, try accessing it in the pixel shader. The Structured Buffer’s HLSL syntax works like an array/vector. So try grabbing the diffuse (.Kd) color of the first INSTANCE\_DATA [0] in the array. Return the color you find to the screen, and you should see this:Text

Description automatically generated

If material sub one [1] returns orange, then you know your data/setup is good to go!

### Part 3F

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 **instance data**?

The good news is that the model is already split-up by mesh in 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. **Make a loop** to iterate across all the available meshes, drawing **only the indices listed in each mesh**. The key difference being that drawing has now been split into one vkCmdDrawIndexed call per-each unique mesh.

If you do this correctly **everything should look the same**.

### Part 3g

We are now ready to start referencing the correct transform/material when drawing the sub-mesh. To do this we will lean on a hardware technique we learned about last time called **instancing**. (Review Lab 3 Vertex Shader)

Typically, we use **SV\_InstanceID** to help select the correct matrix in Vertex Shader. Though the array is no longer hardcoded, the technique is the same. Apply the proper World Matrix associated with a specific instance index.

Because we used identity matrices during the INSTANCE\_DATA setup, we should not see a change when they are applied. (but it will matter soon enough)

### Part 3H

The **System Semantic** SV\_InstanceID is not directly available in the Fragment shader. However, we need it to access the correct material color for each pixel.

This can be achieved by adjusting the **output** of the Vertex shader to forward the incoming index. That way the Fragment shader can now know which index to use for the pixels of this primitive.

Unfortunately, by default the output of the Vertex shader is **automatically interpolated** across the pixels. This is no good for an index value that must remain consistent across the topology of the mesh.

Here is how you fix that:

A black screen with colorful text

Description automatically generated

The HLSL keyword **nointerpolation** ensures that only the attribute of the first vertex in the primitive is transferred.

Logo

Description automatically generated

Once you have access to the **instance index** in the Fragment shader, you should be able to select the correct color.

***Hint:*** *vkCmdDraw and its variants have an argument called* ***firstInstance****, this is crucial for instancing shader code.*

### Part 3I

Ok, now we are really in business. Let’s check that everything is fully working. Add some code to your Render loop to slowly rotate a GMATRIXF over time. Assign this matrix to the instance representing your Orange Logo mesh.

If you don’t see a change, that may just mean you are not updating the contents of the storage buffer each frame before rendering. Use the same process you used in **lab 3** to update the camera. (GvkHelper::write\_to\_buffer)

A screen shot of a computer

Description automatically generated

You now have full control over rendering the various parts of any 3D models you decide to load in a very efficient and scalable manner. There is a bonus section at the end of this document where you can test it out.

## LAB CHECKPOINT | MANUAL COMMIT | DO NOT SKIP

To receive credit for your assignments, you must manually commit to your Repo 4 times (Once at the end of each Part/Section). If you skip this commit (or disable auto commits) you will not receive credit for the previous section.

1. Change the lab’s Title Bar so it says where you are in the lab.
   1. For Example: “John Smith – Lab 1 – Part 1 Complete”
2. Take a Screenshot of the lab window running showing your work. (Windows Key + Shift + S)
3. On Windows the screenshot is in your clipboard, open MS Paint and (Ctrl + V) to load it in.
4. Save the image with the same name as your title bar in the root folder of this repository.
5. Open GitHub Desktop for this repo. You should see the new image as a pending change.
6. Create a commit message with the same name as the title bar & commit your changes.

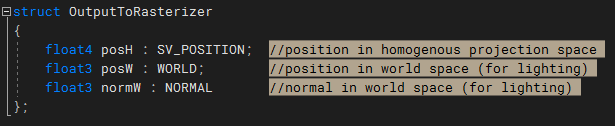
If this section is not perfect but you need to continue forward, you must still do this to get partial credit!

## Part 4 | 100%

### Part 4a

After you get to grips with how to upload and access data with a graphics API; you then get to finally **play 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 later anyway.*

### 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 to 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

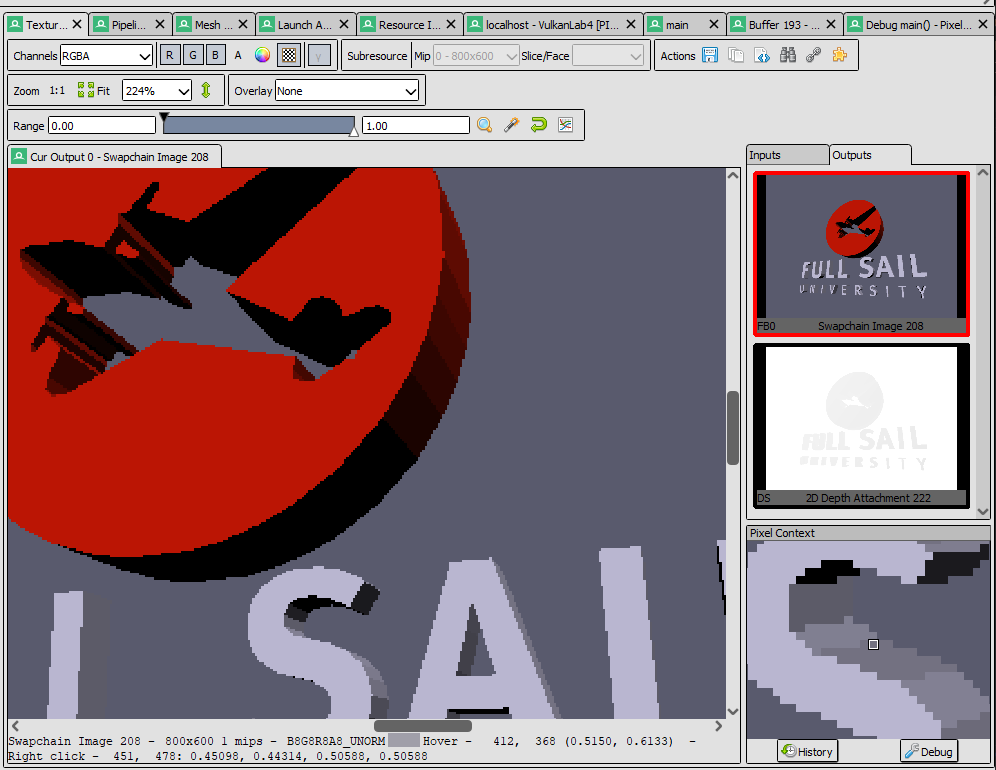
***Hint:*** *The lighting algorithms from CGS expect your Normals to be in World Space. Did you multiply them into WS?*

***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, it’s 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

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 our **global** **shader structure** is and **adding two new vectors**.

We will need an **Ambient** component for 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 to 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.*

## LAB CHECKPOINT | MANUAL COMMIT | DO NOT SKIP

To receive credit for your assignments, you must manually commit to your Repo 4 times (Once at the end of each Part/Section). If you skip this commit (or disable auto commits) you will not receive credit for the previous section.

1. Change the lab’s Title Bar so it says where you are in the lab.
   1. For Example: “John Smith – Lab 1 – Part 1 Complete”
2. Take a Screenshot of the lab window running showing your work. (Windows Key + Shift + S)
3. On Windows the screenshot is in your clipboard, open MS Paint and (Ctrl + V) to load it in.
4. Save the image with the same name as your title bar in the root folder of this repository.
5. Open GitHub Desktop for this repo. You should see the new image as a pending change.
6. Create a commit message with the same name as the title bar & commit your changes.

If this section is not perfect but you need to continue forward, you must still do this to get partial credit!

# Summary

Nice work! Using Vulkan for the first time to draw a 3D model is no small feat. APIs like Vulkan and D3D12 are complex but also very efficient and in-demand on a well-rounded game programmer’s 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)

Once you get to the final assignments in this course, you will receive some hands-on time with PBR materials! Even if you don’t get quite that far this month, I highly recommend completing everything once the class is over.

# 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 **lab 3**.
* 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 2D/3D 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 code in the API\_SAMPLES repo 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.
  + 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.

## Part 5 | 0% | KNOWLEDGE/SKILL CHALLENGE

Attempt to re-create the image below using your understanding of the sequential instanced buffers discussed in the overview of Part 3. You should only have to add two more INSTANCE\_DATA and NO additional draw calls.

That said, you will need to adjust the Instancing arguments of the two existing draw calls.

A screen shot of a logo

Description automatically generated

Once you get this working, try adding 1000+ randomly positioned and colored instances. No matter how many you add, you should still have only 2 total draw calls.

Use a framerate measurement tool like **NVidia FrameView** to see how many you can simultaneously render on-screen before your machine dips below 30 fps in Release mode. (You may be surprised)