Assignment 2 Direct3D12

# 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 DirectX API

1. DirectX and subsequently Direct3D10-12 are included with the Windows SDK: <https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/>

## 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

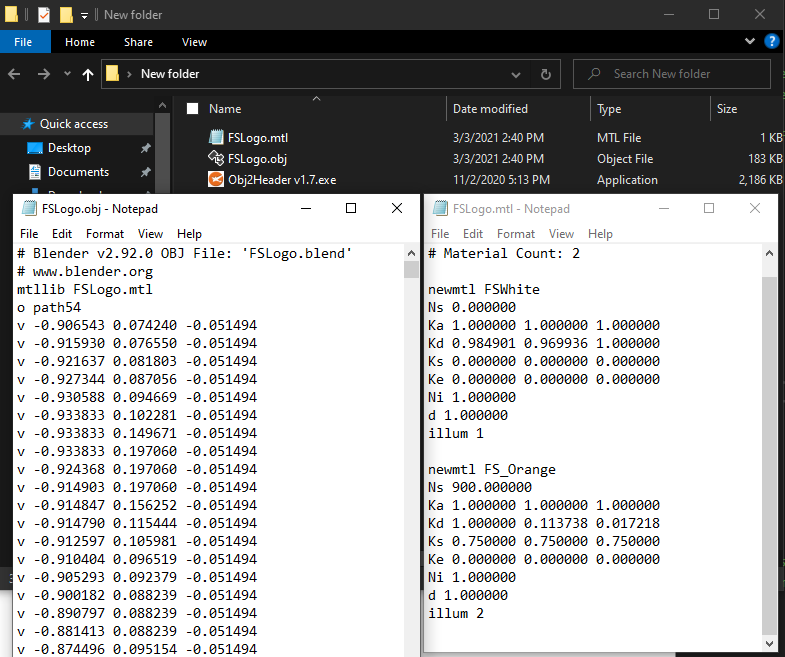
Description automatically generated

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.

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 **D3D12\_VERTEX\_BUFFER\_VIEW** and the **D3D12\_INPUT\_LAYOUT\_DESC** 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. You will need to create one **D3D12\_INPUT\_ELEMENT\_DESC** for each vertex attribute in **OBJ\_VERT**. (The **semantic** names are up to you, just try and remember what you set them to)

Diagram

Description automatically generated

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

***Note:*** *Keep your eyes glued to the Output Tab in Visual Studio while working with D3D12. If the API detects it is being used in an improper or unexpected way, it will print error messages there. Resolve errors 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 understanding of 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.

Diagram

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 D3DCompile to figure out what you are getting wrong with the syntax!*

***Tip:*** *Match your* ***HLSL input semantics*** *to the same string names you used when filling out the* ***D3D12\_INPUT\_ELEMENT\_DESC*** *structures in the last part. This lets the API know where to pull the vertex data into.*

### 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 **D3D12\_INDEX\_BUFFER\_VIEW** and **ID3D12Resource** to hold our indices. If you mimic the vertex buffer and use the **Microsoft::WRL::ComPtr<>** smart pointer for your ID3D12Resource you will not need to **Release()** it manually. (Which is a handy feature)

***Tip:*** *Beware of copy-paste errors here, also when selecting the* ***Format*** *of the index view be sure to select the one that most accurately identifies what each element of the index buffer contains.*

### Part 1H

With our **index buffer** now allocated and populated, we can use it to draw the model as intended. You will want to **Set** 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 counter intuitive 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)

Table

Description automatically generated with low confidence

**(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 must direct the program to your executable file.*

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.**

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:*** *Since D3D11 & D3D12 use the same* ***NDC*** *you learned in CGS; you could use the same math here if desired.*

### Part 2B

Direct3D12 has **three primary ways** to communicate **variable data** to running shaders: **Root Constants**, **Constant Buffers** and **Structured Buffers**. Root Constants are an efficient 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 **256bytes** of data via this method. Additionally, this memory must be shared with inline descriptors and links to descriptor heaps.

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 a **Constant Buffer**. Though not quite as easy to use(or as blazing fast) as Root Constants, they can hold a more useful amount of memory. The first section of this buffer will hold the data for our overall scene, the second area we will use later to store & retrieve information about each sub-mesh in the model.

**Structured Buffers** allow you to store significantly more than Root Constants or Constant Buffers. If you potentially have megabytes or even gigabytes of non-vertex/non-texture data, then a Structured Buffer may be the correct choice. The trade-off is potentially slower memory access; API complexity is similar to Constant Buffers.

Before we dive into their creation lets **organize the data** we intend to send to our shaders. The following two data types should cover most of our needs when rendering the Logo:

Text

Description automatically generated

***Note:*** *I have added additional unused data to the above structures to simplify treating them as Constant Buffers.*

***Tip:*** *You will notice above shader data is also 16byte**aligned. Structures on the CPU may not line up with the equivalent version on the GPU if you are not careful about aligning data to 16byte register boundaries.*

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.

A screenshot of a computer

Description automatically generated with medium confidence

***Note:*** *Your current template code may not be compiling your shader code in a version of HLSL new enough to support the above syntax. Go find where the shaders are compiled and ensure they are running at least SM 5.1.*

Complete this step by making an **instance** of your new C++ structures and **initializing** them to all the math variables we made at the start of this process. The model we are attempting to draw contains more than one sub-mesh, be sure to take this into account. Transfer all the **materials** from the model’s **header file** to the materials of each MESH\_DATA we are going to need. We can use the same world matrix for both meshes for now.

***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 **Constant 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 Direct3D12 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 enough constant buffer memory 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 set of constant buffers** it can use directly. Depending on the current frame we are drawing, we will offset between the different sets.

Similar to **vertex and index buffers**, go ahead and create a **Constant Buffer**. You will need the **ID3D12Resource** type again. There is no such thing as a “D3D12\_CONSTANT\_BUFFER\_VIEW”, unlike vertex & index buffers we must connect Constant Buffers to the shaders a bit differently. In the next part we will reserve enough GPU memory based on the **maximum number of frames** reported by **GDirectX12Surface**.

### Part 2D

The amount of memory required in our constant buffer will be: (The size of our **SCENE\_DATA** + The **number of meshes** to draw multiplied by the size of one **MESH\_DATA**) multiplied by the **maximum number of active frames\***. This will ensure that each simultaneously rendered frame has its own copy of SCENE\_DATA and MESH\_DATA structures. Again, the idea being to edit them without causing synchronization artifacts due to data sharing.

Once you have successfully calculated the total size, use the function **CreateCommittedResource** to allocate the Constant Buffer. For this you can essentially just duplicate the same code used to allocate the vertex & index buffers. Once you have done so use **Map**(…) and **Unmap**(…) like before, but this time you will need to copy the bytes for the SCENE\_DATA and MESH\_DATA to fill the buffer and repeat the process for each active frame.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Frame 0 | Frame 0 | Frame 0 | Frame 1 | Frame 1 | Frame 1 |
| SCENE\_DATA | MESH\_DATA | MESH\_DATA | SCENE\_DATA | MESH\_DATA | MESH\_DATA |

You can use **pointer arithmetic** along with the **std::memcpy** function to transfer the data you initialized earlier in **2B** to the buffer. The above image is an example of how you could layout your data if you had two maximum active frames at any one time. Do not hardcode the size, however the pattern should be the same for each active frame.

***\*Guide:*** *For the time being getting the maximum frame count is a bit clunky, you will need to do the following: Call GDirectX12Surface.****GetSwapchain4****(…) then use the function* ***GetDesc****(…) from* ***IDXGSwapChain4****. The resulting* ***DXGI\_SWAP\_CHAIN\_DESC*** *struct will contain a “****BufferCount****” member. This is the maximum active frames.*

***Note***: *Constant Buffers are required to be multiples of 256 bytes. If your constant buffer is producing API errors, then you will probably need to add some extra padding bytes onto the end to satisfy this requirement.*

### Part 2E

Ok… with our Constant Buffer allocated and populated its time to link it to our Vertex & Pixel shaders. This task is a bit more complicated to do than it was in older APIs like Direct3D11 and OpengGL, but it does offer a lot of performance and flexibility improvements.

The first thing we are going to need is a **ID3D12DescriptorHeap**, add one to your Renderer class. A Descriptor Heap is how we can store pointers to things like Constant Buffers and Textures that need to be connected to Shaders.

Finish this section by allocating a descriptor heap that has enough room to hold our one constant buffer. You can use the API function **CreateDescriptorHeap(…)** to do so. When filling it out, make sure the heap will be **visible to shaders** and support adding at least one **constant buffer view**.

### Part 2F

With our Descriptor Heap allocated, the next item on the list to write a pointer to our new Constant Buffer into it.

You can do this using the function **CreateConstantBufferView(…)**, when populating this function’s arguments, you will need to know the constant buffer’s “**GPUVirtualAddress**” to know its location. Additionally, you will need to know the “**CPUDescriptorHandleForHeapStart**” of our new Descriptor Heap so it can overwrite the correct memory location within the Heap. (When writing multiple Views to a Heap you would need to add to this address)

***Tip:*** *Keep your eye on the Output Window, if this operation fails it may be due to 256-byte alignment issues.*

### Part 2G

I realize this is getting a bit long-winded, but we are getting much closer to seeing something back on the screen. Also, the methods we are using to connect this constant buffer to the shaders are applicable to all sorts of things.

Now that we have a populated Descriptor Heap, we need to inform something called the **Root Signature** that our shaders will be using it to access data. The Root Signature is the central “brain” that controls what the shaders can or cannot see regarding resources.

A Root Signature generally holds some combination of the following items:

Root Constants – Small inline variables directly accessible by shaders. (Ex: An Array Index)

Root Descriptors – A direct link to a View you wish accessible by shaders. (Ex: Constant Buffer Views)

Root Tables – A indirect link to a collection of multiple Views a shader needs to access. (Ex: Textures & Samplers)

As you can probably guess, some Root Descriptors should suffice for what we are trying to do. To inform the Root Signature this is what we want, create two **CD3DX12\_ROOT\_PARAMETERs** and initialize them as a **Constant Buffer Views**. Use the appropriate values that reflect what **registers** the constant buffers should be bound to in the vertex & pixel shaders. (Look inside the Shader code)

Complete this section by passing the two new CD3DX12\_ROOT\_PARAMETERs as arguments when **initializing** the CD3DX12\_ROOT\_SIGNATURE\_DESC. This will **add them to the Root Signature**, so the Shaders expect them.

***Note:*** *You may be somewhat confused as to why we are creating two Root Descriptors when we only have one Constant Buffer View. As described in* ***Part 2D****, we only really need one buffer to hold everything. From the shader’s point of view there will be two different buffers, but we will just direct the shader slots to different locations in the same buffer to avoid having to make multiple buffers. (This is some of that flexibility I was referring to earlier)*

### Part 2H

Now that we have everything setup for the Root Signature, it’s time to connect our constant buffer to the rendering pipeline so our shaders can read the contents. If you set everything up correctly, then in theory your shaders will have access to our single constant buffer like so:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Frame 0 | Frame 0 | Frame 0 | Frame 1 | Frame 1 | Frame 1 |
| SCENE\_DATA | MESH\_DATA | MESH\_DATA | SCENE\_DATA | MESH\_DATA | MESH\_DATA |
| HLSL Register 0 | HLSL Register 1 | | HLSL Register 0 | HLSL Register 1 | |

***Tip:*** *Frames & Registers sharing a buffer is possible because we aligned our data structures to 256-byte boundaries.*

The idea here being that when drawing a specific cycling frame (ex: 0,1,2,0,1,2,0) we will **offset the pointer** when assigning the **constant buffer view** to point at where the data should start for that specific frame. Once again, the purpose of this is to avoid sync issues if multiple frames access the same mutable data.

First you will need to call **SetDescriptorHeaps(…)** to inform the video card that you are about to reference the memory you allocated for your constant buffer view earlier. Then you will need to call **SetGraphicsRootConstantBufferView(…)** two times. First to assign the **SCENE\_DATA** in the buffer to the first **ConstantBuffer<>** at register/slot **0**. Once more to assign the **MESH\_DATA** in the buffer to register/slot **1**.

The key to this part is to determine the **GPU Virtual Address** of your constant buffer, and then use pointer arithmetic to jump to the correct location where the register should be pulling its data from. For the time being just link everything to **Frame 0**, we needn’t to worry about synchronization until we start updating the contents.

***Note:*** *A more traditional approach would be to have* ***multiple separate data buffers mapped to each register slot*** *(one per frame). If you are curious: The Vulkan variant of this assignment takes such an approach, though it can do this type of buffer sharing method as well using* ***dynamic uniform buffers****. (Try both to get a better understanding)*

### Part 2I

Yes… this is the last part; and yes, we will finally see our model in 3D after this! The only thing left is to finally **use** both the **SCENE\_DATA** and **MODEL\_DATA** in the vertex shader.

In the **HLSL vertex shader** make the following changes:

* Add two **ConstantBuffer<>** to the code so we can access the **constant 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 stored in the **ConstantBuffer<>** located at register **b0**.

Logo

Description automatically generated

That was quite a journey, but our shader code now has access to the model’s transform and material data! Much like Vulkan, Direct3D12 is a complex API that gives you significant control over most details. 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 yellow color and **replace it with a color from our actual materials**. To do so, copy the **HLSL** code related to **materials** and the two **ConstantBuffer<>** 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

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**? To get a better understanding, look at how the model is split-up as outlined in the **obj** file itself:

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.

Thankfully we have already included all the material info we need in the constant buffer. For each frame all the material data has been stored in a contiguous fashion, so all we need to do is adjust the **offset** where the Constant Buffer at **register b1** starts for each mesh.

|  |  |  |
| --- | --- | --- |
| Frame 0 | Frame 0 | Frame 0 |
| SCENE\_DATA | MESH\_DATA | MESH\_DATA |
| HLSL Register 0 | HLSL Register 1 | HLSL Register 1 |
| Applies To Any Draw | Only When Drawing Mesh 0 | Only When Drawing Mesh 1 |

***Note:*** *In a larger project it may be more efficient to further split mesh materials and matrices into their own arrays.*

The good news is you already have the blueprint of how to do this. Find the code where you call **SetGraphicsRootConstantBufferView(…)** to point **register b1** (the second constant buffer) to the start of the MESH\_DATA. **Move** this code **inside** the drawing loop. Using your understanding of the diagram above, adjust the pointer arithmetic to select the correct **MESH\_DATA** for each drawn **sub-mesh**. (Before calling draw)

Text, logo

Description automatically generated

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

***Tip:*** *A more flexible and spacious option for large data sets would be to switch from using Constant Buffers to Structured Buffers**(specifically just for your mesh information). Unlike constant buffers, structured buffers are designed for* ***array style access using indexing****. Instead of pointer math, you can use Root Constants to select the correct mesh data more easily for techniques like Instancing. (Worth considering for the Level Renderer)*

## 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**.

Text

Description automatically generated

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)

Text, logo

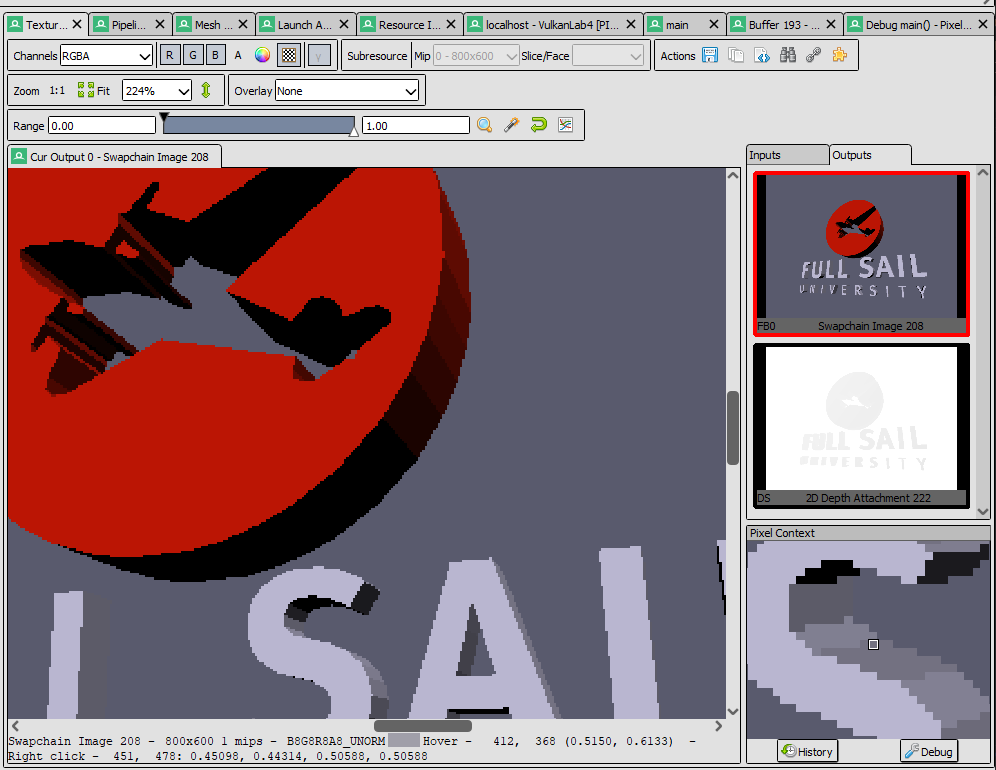
Description automatically generated

***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! On the CPU side **set** the **second world matrix** in the **MESH\_DATA** 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 **constant buffer** associated with **this frame**. To do this you can use the same method you used to originally copy the data to buffer after its creation(Map & Unmap). However, this time we will want to copy the data each time we **Render** a new frame to reflect the new changes.

After you do this, you should see the top part of **Logo spinning**. The lighting should change based on the rotation of the logo. If it does not, make sure you are correctly transforming the **normal** attributes into **world space**.

Text

Description automatically generated with medium confidence

***Tip:*** *Updating GPU shader variables is an operation that typically happens once each frame after all CPU game objects & matrices have been altered based on any running game logic (AI, Physics, Input etc..).*

### Part 4E

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 the first section** of the **constant buffer** we reserved for multiple frames. Earlier we made sure to **create enough room for each rendered frame** to avoid such synching problems. Unfortunately, we are currently only using the **Frame 0**’s data set. Let’s correct that oversight now.

Use the existing **GDirectX12Surface** to capture the active **Swap Chain Image** we wish to render to (A.K.A the current/active frame). Use this to calculate a **“frame offset”** value that represents how many **bytes** we need to **skip** to get to specific data reserved for the **active frame** in use.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Frame 0 | Frame 0 | Frame 0 | Frame 1 | Frame 1 | Frame 1 |
| SCENE\_DATA | MESH\_DATA | MESH\_DATA | SCENE\_DATA | MESH\_DATA | MESH\_DATA |
| Frame Offset = 0 Bytes | | | Frame Offset = N Bytes | | |

Once you compute this value, add it to every call of **SetGraphicsRootConstantBufferView** so we ensure no data is shared across frames. If successful everything should look the same, and any flickering issues should be resolved. We should now be safe from any current or potential cross-frame synchronization problems.

***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 4F

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.

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 below picture. (*If you need a refresher on the ambient term, again check CGS day 7*)

Text

Description automatically generated

***Note:*** *If the ambient is correct, no part of the image should be fully devoid of light. (Ex: dark oranges & greys)*

### Part 4G

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

***Note:***[*Evidently*](https://gamedev.net/forums/topic/679072-d3d12-issue-pow-hlsl-function/5294516/) *the pow intrinsic has a quirk in SM 5.1 where if you pass 0 as the power it can get a NAN result instead of it safely returning one. I suggest adding 0.000001f to all material power/exponents to be safe.*

(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:

Text, 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 D3D12 for the first time to 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.

## Direct3D12 API

<https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics> (Main Documentation)

<https://github.com/microsoft/DirectX-Graphics-Samples> (Official GitHub API Samples)

## HLSL High Level Shading Language

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

<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)

[..\..\..\Gateware\documentation\html\index.html](file:///C:\Users\lnorr_000\AppData\Gateware\documentation\html\index.html)

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

# FAQ

* Your example does not stretch or skew weirdly when I resize the screen?
  + When I completed the sample, 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 Direct3D12 API correctly?
  + Aside from reading the docs and making sure the code compiles, we have enabled run-time debug output in the Direc3D12 API (In Debug mode only). Be sure to pay close attention to the Visual Studio **Output** window when running the program. Any non-fatal mistakes you make will be reported by the Direct3D12 runtime and printed there.
* The HLSL shader code appears to just be a string, 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.
  + DirectX has a shader compiler called FXC, it can convert your shaders into shader byte code used by the GPU drivers. In-case there are errors while compiling your shaders I added code to print them to the console. Keep your eyes on it.
  + Visual Studio can compile your HLSL code into header files, look inside the CMakeLists.txt file to learn how. You can do this as an alternative to compiling your shaders at run-time. Once your shaders get very 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 first handout.
* Working with the Constant Buffer/Root Signature is super confusing… Help?
  + Attaching resources in D3D12 can be pretty intimidating at first, studying code examples can be very helpful. I suggest going to the official Microsoft GitHub DirectX12 sample repository and checkout the [D3D12HelloWorld](https://github.com/microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop/D3D12HelloWorld) samples. It should be clear which one is most relevant to the task at hand.
  + If you want to know more, there is some decent content out there covering these topics. [This video](https://youtu.be/Wbnw87tYqVg) is probably a bit more in depth than you really need right now, but the first part of it is clear and to the point. I recommended watching the start of it when trying to get a better sense about what is going on.
* 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(Kai Huang) did originally have to write the Direct3D12 interface to Gateware, setting up a modern Graphics API like Vulkan or Direct3D12 from scratch would quickly turn this into a full-blown Project and 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.