Assignment 2 Direct3D11

# Introduction

Unlike the other APIs, we offer a **full top-to-bottom video series** on this API and **significantly more** example code for you to study. (Check the links under RESOURCES)

In this assignment, 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 **constant 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. Download & install the CMake build tool [cmake.org](file:///C:\Users\lnorr_000\AppData\Roaming\Microsoft\Word\cmake.org) (be sure to check “install for all users”)
2. Reboot your computer. (or type **taskkill /f /im explorer.exe && explorer.exe** into a command prompt)
3. Open the directory containing this document in windows explorer and select the path bar at the top.
4. Type **cmd** into the bar and a command prompt should open. Type: **cmake -S ./ -B ./build** enter.
5. This should generate a solution inside a new folder. Open it and set it as your startup project.

# Assignment 2

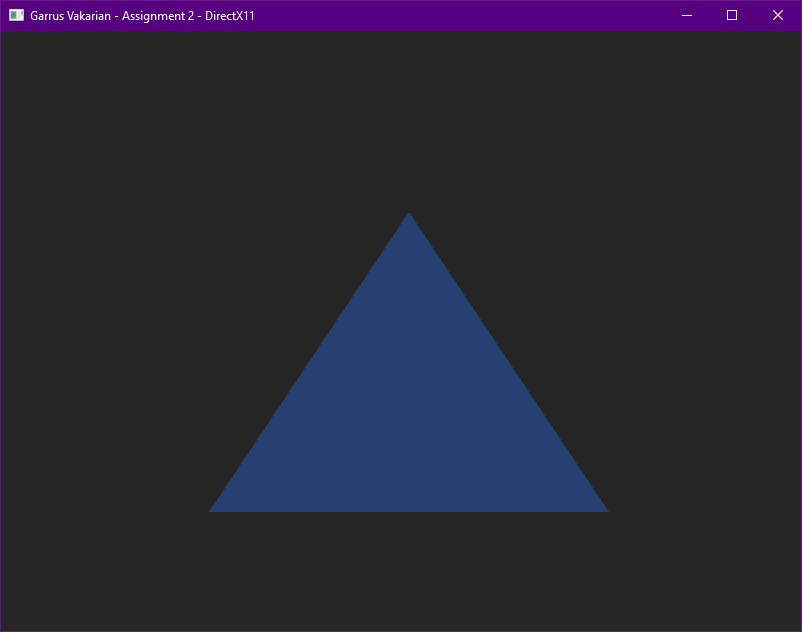
Part 1 | 25%

### Part 1A

Study the code and familiarize yourself with where things are and what they’re doing.

(optional) Change the triangle and background colors to some colors you like.

Use the **SetWindowName** function from **GWindow** to set your name and API variant as the window’s name.



### 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 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 reads them in and has 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 to render it correctly.

Graphical user interface, text

Description automatically generated

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)

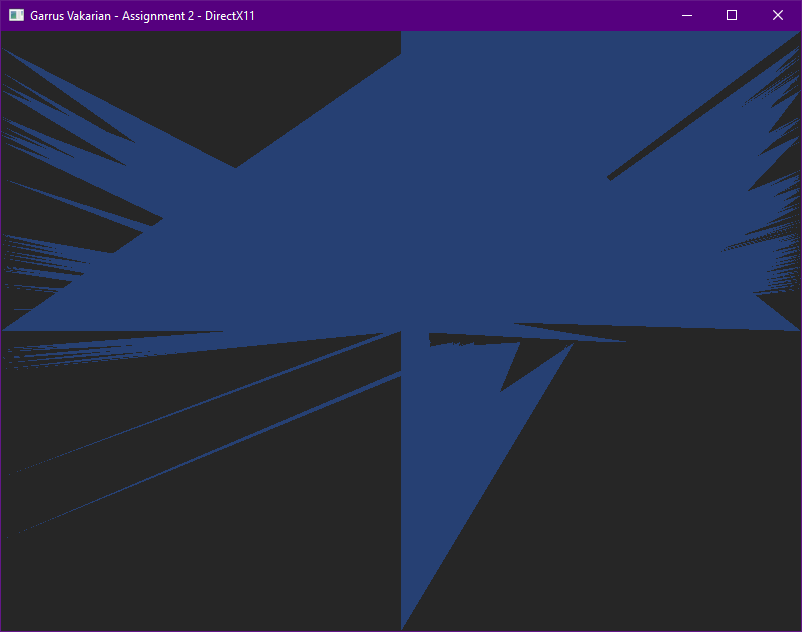
### Part 1C

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.



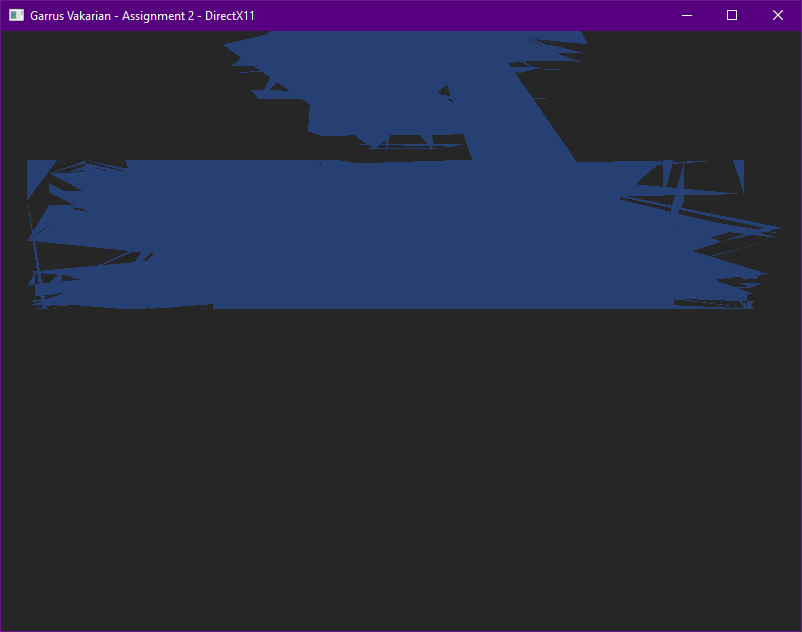
As you can see, **the current code is designed to draw 2D NDC triangles.** Though we are drawing the full amount of vertex information, the API does not understand that this data is meant to be used as a **full 3D model that includes an XYZ position, UVs and even normal data.**

### 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, adjust the **D3D11\_INPUT\_ELEMENT\_DESC** to correctly match the format of the **OBJ\_VERT** structure – this is the second part of the rule of three. You will need one **D3D11\_INPUT\_ELEMENT\_DESC** for each vertex attribute. The **semantic** names are up to you – just try and remember what you set them to, you will need to reference them in your shaders later.

Once you’ve corrected the vertex view, make sure to adjust the **stride** is updated to match the size of your vertices.

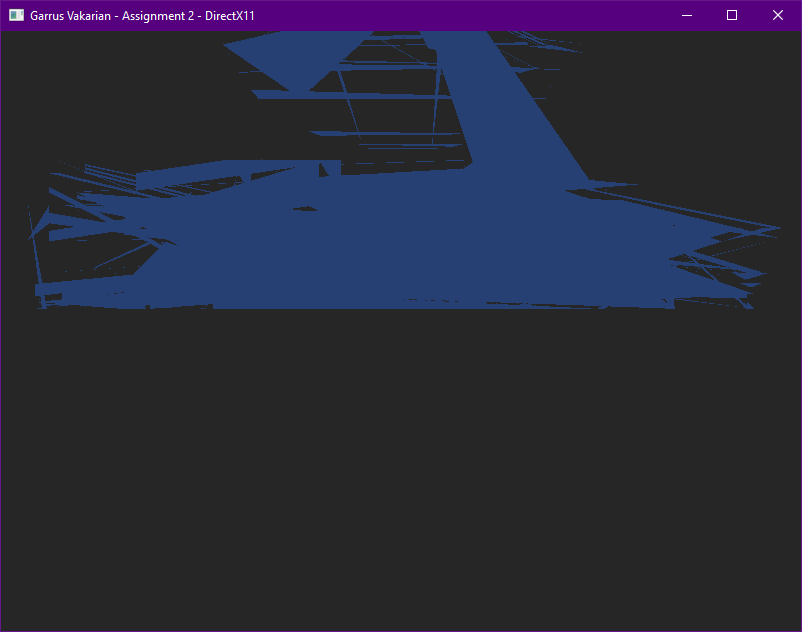


It’s starting to look like something… but still definitely not correct!

### 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. You can use multiple parameters to do this, or you can make an HLSL struct that mirrors your vertex struct. Tweak the output data so that it uses the full **XYZ** component of the position, setting the **W** to its standard homogenous starting value.



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

***Tip:*** *Match your* ***HLSL input semantics*** *to the same string names you used when filling out the* ***D3D11\_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, most 3D model files require it, and GPUs are optimized for indexed draw operations. Thankfully, all modern graphics APIs can accept and use **index buffers**.

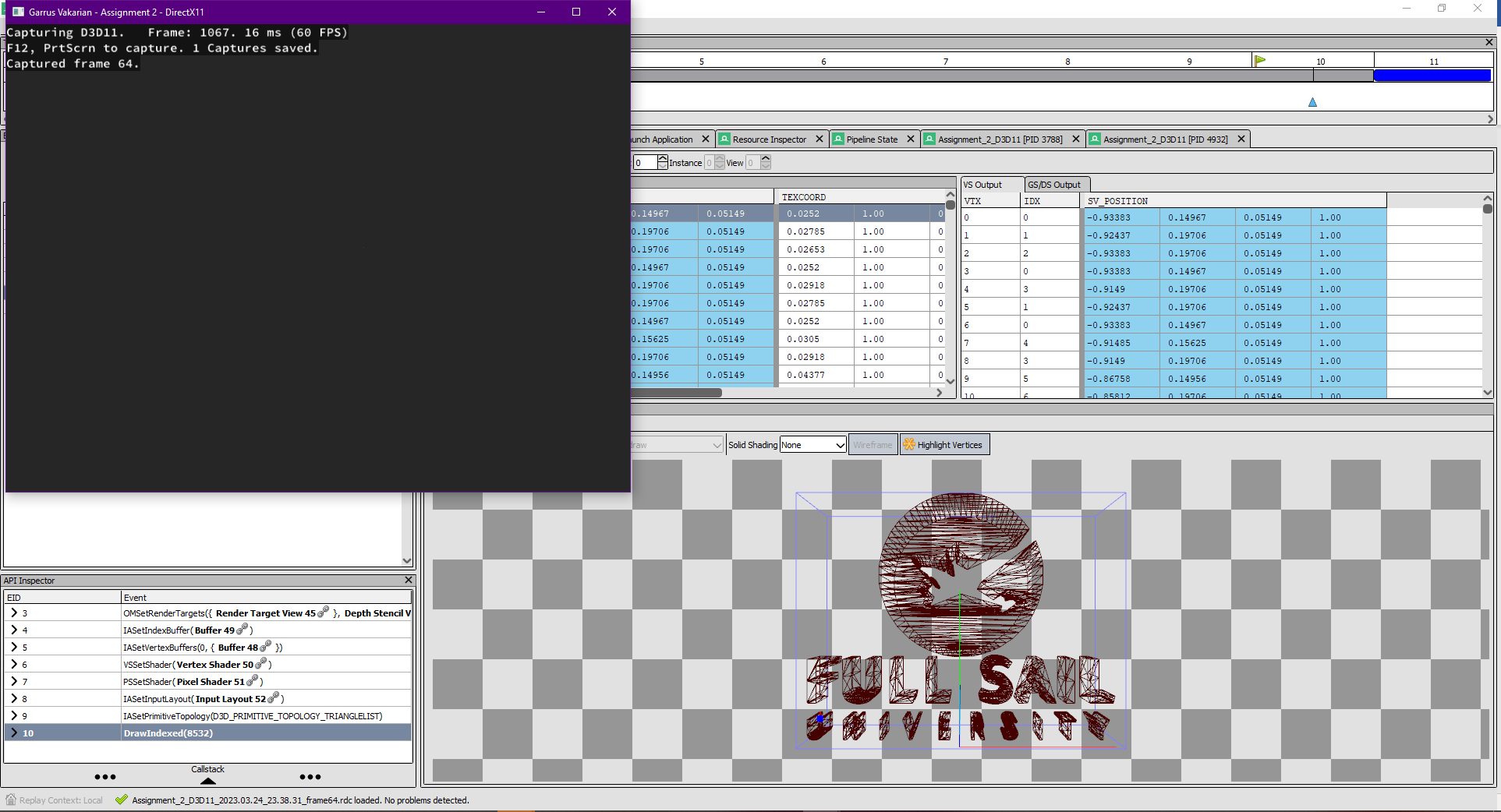
We can create an index buffer easily by copying the code we’re using to create our **vertex buffer**, and adjusting it to take in the index array we just looked at. Make sure the buffer descriptor struct you pass to the **CreateBuffer** call correctly describes an index buffer. We also will need an ID3D11Buffer pointer to hold our handle to this buffer. If you mimic the vertex buffer and use a **Microsoft::WRL::ComPtr<>** smart pointer, you will not need to **Release()** it manually, which is handy.

***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 have an easier time with this assignment and future programming if you code cleanly.*

### Part 1H

Now that our **index buffer** has been allocated and populated, we can use it to draw the model as intended. You will need to **set** the index buffer before drawing, and also switch to a **draw function** that supports **indexed** geometry submission.

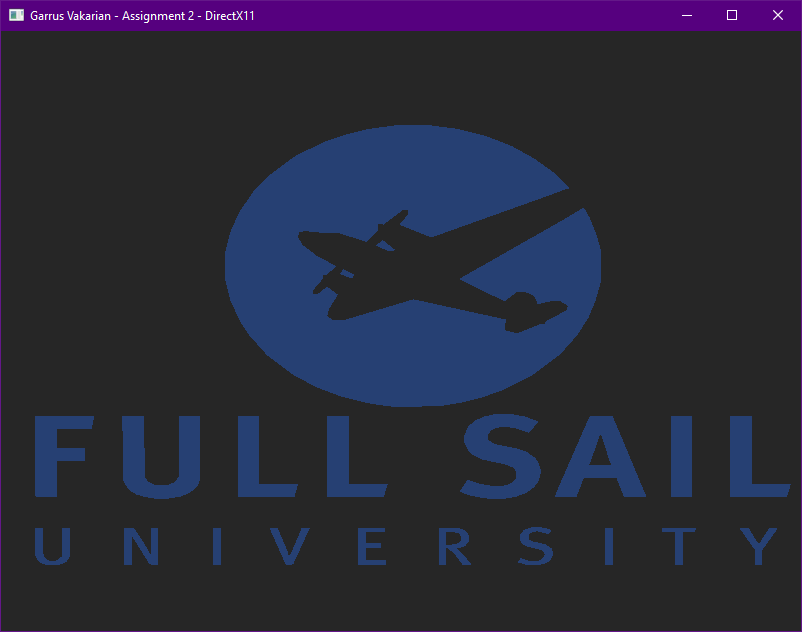
If you do this step correctly, the model will disappear! This may seem counterintuitive, but remember that without a **View Matrix**, the camera is functionally located at the **origin**. Since we haven’t implemented a **world matrix** either, our model is also at the origin – this places our camera **inside** of the model, and we cannot see the back of it due to **back-face culling** (which all APIs have enabled by default).



**(Optional Step)** GPU/API Debuggers like [**RenderDoc**](https://renderdoc.org/) are crucial for programmers writing graphics code. In the above screenshot I 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 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.**



***Note:*** *I chose the numbers above out of experimentation and because I knew the 3D model was small and created around the origin. This will not necessarily work for every model and is just temporary so we can see something. It is not a substitute for writing a 3D vertex shader, which is what we will be doing next.*

Part 2 | 50%

### Part 2A

In this part 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 in our shaders.

In the previous assignment, you used **Gateware** (or some other math library) to make **World, View, and Projection** matrices. 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:

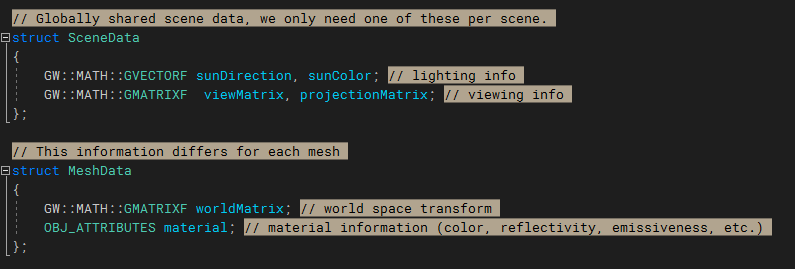
**Light Direction:** *Forward with a strong tilt down and to the left.* ***-1x -1y +2z*** *(normalize)*

**Light Color:** *Almost white with a slight blueish tinge.* ***0.9r 0.9g 1.0b 1.0a***

### Part 2B

Since the amount of data we need to transmit is pretty small, we will be using **constant buffers** once again. This time we will be using *two* constant buffers – one for our scene-wide data, and one for our per-mesh data.

To make transmission of this data easier, create two structs for the data we want to send – one for each buffer. The following screenshot is an example of what these structs might look like:



After you’ve made these structs in C++, mirror them in your vertex shader. You will also need to make an HLSL struct that mirrors the **OBJ\_ATTRIBUTES** struct. Once you’ve done so, you can switch your HLSL SceneData and MeshData structs to be cbuffers as you did in the previous assignment. Be sure to put each of these buffers into a unique register.

Complete this step by making an **instance** of your new C++ structs and **initializing** them to all the variables we created at the start of this process. For the material, initialize the struct to the first mesh’s material (we will be handling the fact that our meshes use different materials in a later step).

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

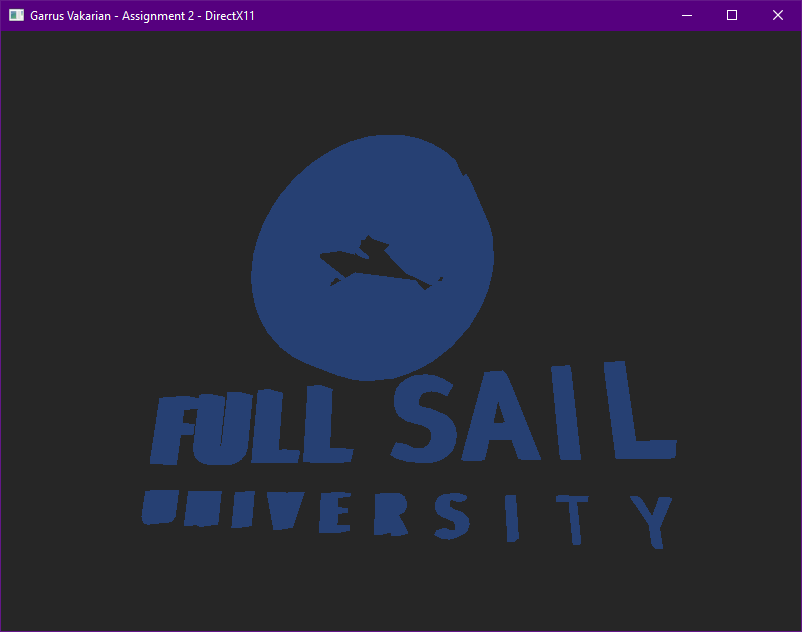
### Part 2C

Use your prior experience in creating constant buffers to make two constant buffers in your renderer – one for your scene data and one for your mesh data.

Next, **Set** the constant buffers of your shaders before **Drawing.**

### Part 2D

Finish this section by implementing your world, view, and projection matrices in your **Vertex Shader**.



***Note***: *Constant Buffers are required to be 16-byte aligned. 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. Also don’t forget that by default HLSL treats matrix data as column major not row major!*

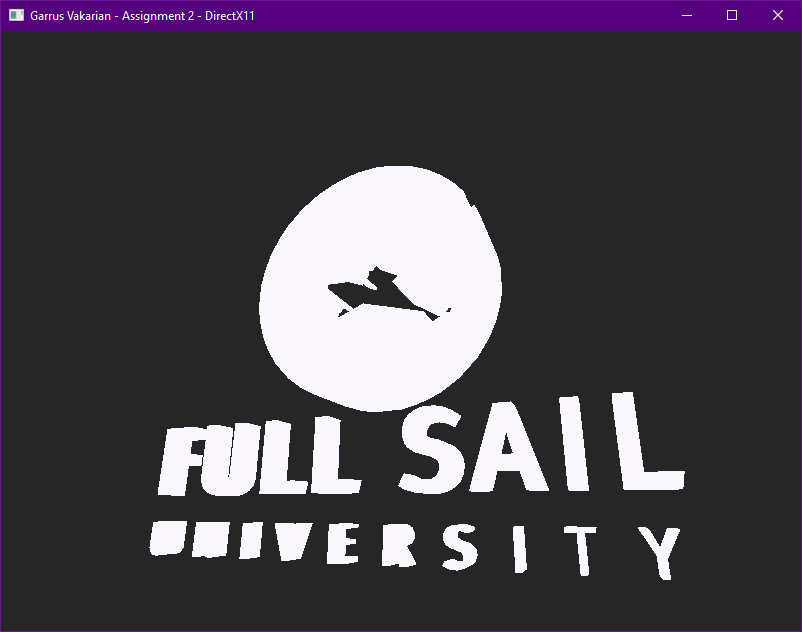
In the last two sections, we will focus on using our new material data to **visually enhance** the 3D model, so it **appears as the artist had originally intended**.

Part 3 | 75%

### Part 3A

The first thing we should do is get rid of the hardcoded color and **replace it with a color from the actual materials**. To do so, copy the **HLSL** code that declares the **constant buffers** from your vertex shader into your **pixel shader**. Keep in mind that shaders **do not share namespaces** – any structs you want to use in your pixel shader will need to be declared **inside** the pixel shader.

**Replace the output color** with the **diffuse color** from the **material** – it should be solid white.



This is the correct color for the **text**, but the **logo** is supposed to be orange…

***Note:*** *When setting the constant buffers for your shaders, remember that each shader stage has its own* ***SetConstantBuffers*** *function.*

### Part 3B

So… how do we correctly draw the model so that the text is white and the logo is orange? Or to be more precise, how do we draw each **mesh** using its respective **material**? To get a better understanding, look at how the model is split up in the FSLogo.h file itself:

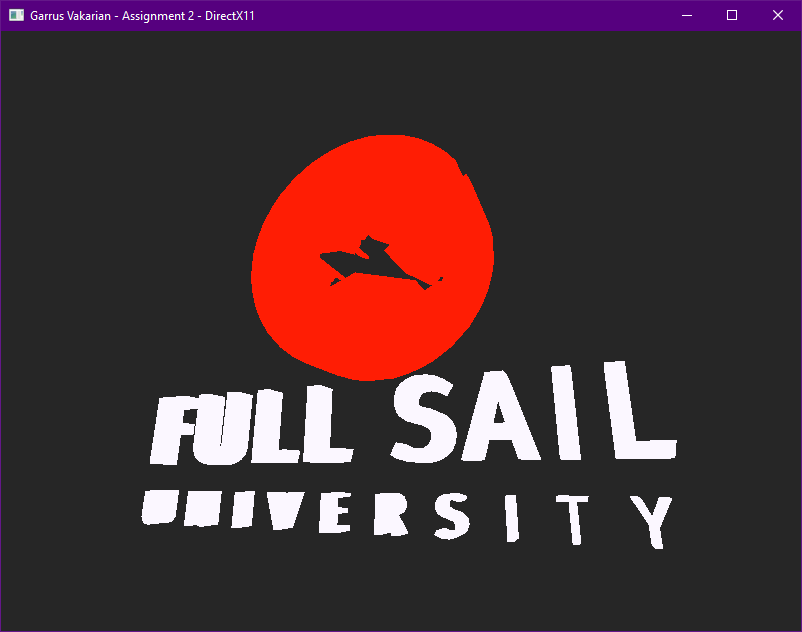
Text

Description automatically generated

We will now **adjust our drawing code to draw each mesh individually** instead of drawing the entire model all at once. This gives us a window to update the material attributes 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

Once you have each mesh drawing separately, use the **Map/Unmap** or **UpdateSubresource** commands to switch the contents of the buffer to the correct material just before your draw call.



Looking good! Now we have a simple and effective way to edit the data used in our shader’s **constant buffer**.

***Note:*** *There are a few different methods of getting constant data over to the GPU in DirectX11.* ***Constant Buffers*** *are simple to set up, but have a limit of 4096 constants. A more flexible and spacious option would be to use a* [***Structured Buffer***](https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/sm5-object-structuredbuffer) *(specifically for our per-mesh information). Structured buffers are indexable buffers that contain many instances of a* ***structure*** *– this makes them very friendly to instanced draw operations. Instead of needing to update a buffer before each draw call, you can instead use* ***SV\_InstanceID*** *to index into a* ***Structured Buffer*** *and select the correct data for your mesh (worth considering for the Level Renderer).* [***Other resource types***](https://learn.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-cs-resources) *also exist, but they are out of the scope of this assignment.*

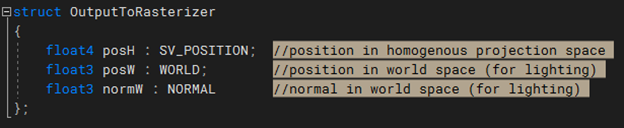
Part 4 | 100%

### Part 4A

After getting to grips with how to upload and access data within DirectX11, we get to the fun part: **playing with shader code**! The 3D model looked much nicer in **Blender** because it was applying a **light source** to the model and using the model’s **materials** to tune how each surface interacted with that light.

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

In the **vertex shader**, make a **struct** that can be used to **output more than just the position**.

***Note:*** *I did not include a* ***UV******coordinate*** *in the output struct since we won’t require it in this assignment. You could add one if desired since the input has one available, and you may end up using this code in your* ***Level Renderer****.*

### Part 4B

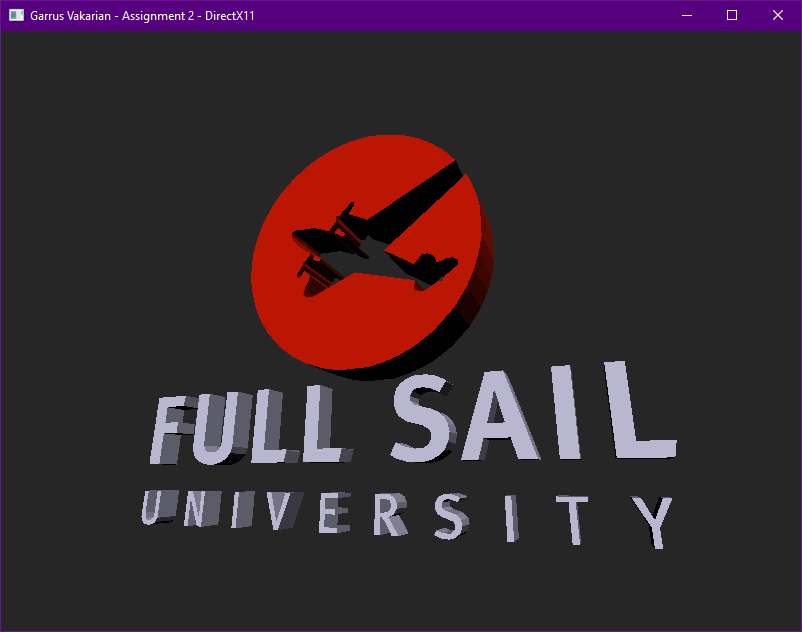
Now **adjust your vertex shader** so it will **return** an instance of the new struct you defined. You will need to fill out each member of the structure based on the input values. Don’t forget to **transform the outgoing normal into world space** – your light positions and directions will be defined in world space.

Next, **adjust the arguments of the pixel shader** so they match the **exact type, semantics, and order** of 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).

***Note****: You can also declare the vertex shader’s output struct in the pixel shader if you’d rather work with it instead of taking the parameters in separately. Either way will work as long as the correct types and semantics are used.*

### Part 4C

Now we have everything we need to apply a **directional light formula**. Since you probably don’t have this formula memorized, it was covered on **CGS day 7** (you can find the formula in the lecture slides). Use the **diffuse color** of the **material** as the **surface color** and the **world-space normal** from our vertex shader output 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).

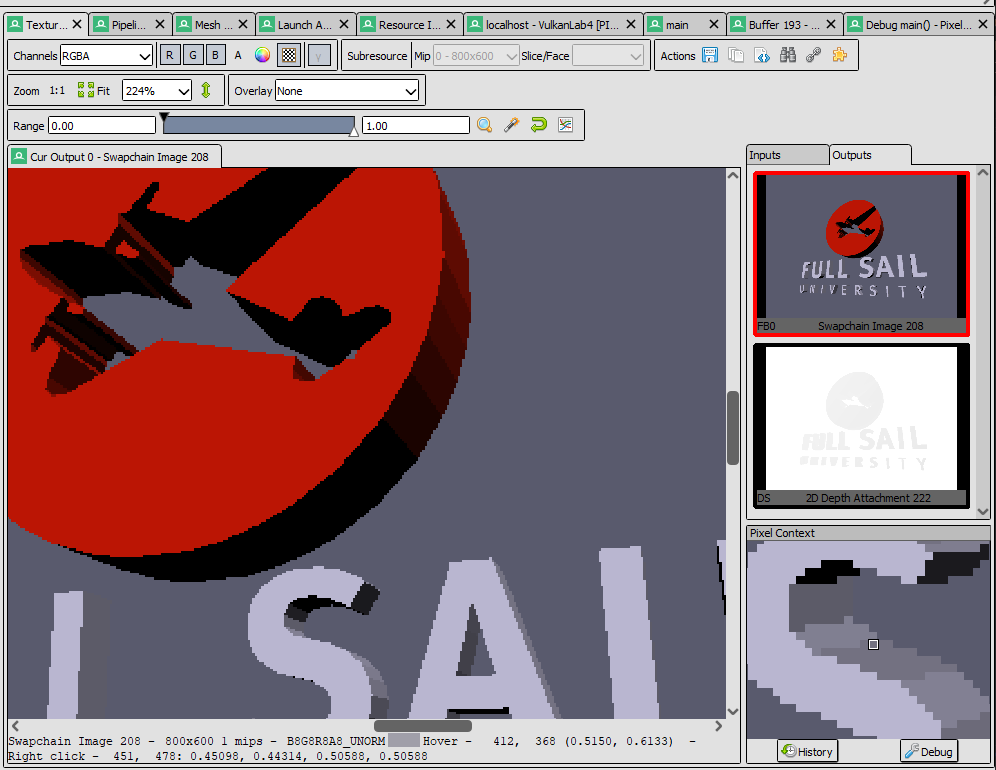


***Note:*** *The normal coming in from the* ***rasterizer*** *has been interpolated across a triangle. This means it* ***may not be normalized*** *for a given pixel. Thankfully, there is an* ***HLSL***[*intrinsic*](https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-intrinsic-functions) *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:**



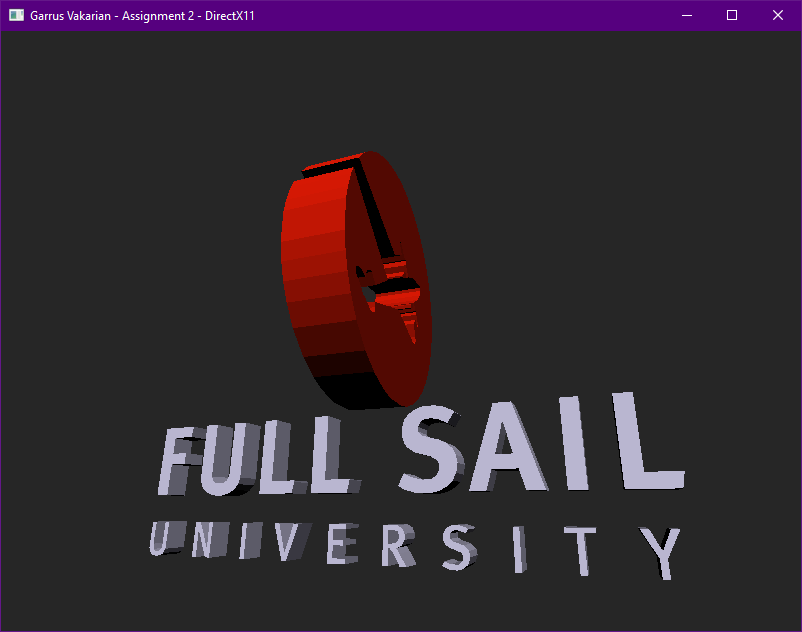
Select your **draw call on the left side** and open the **texture viewer tab**. Select the **Swapchain Image** and then **right-click the pixel** you wish to inspect. Once you do so a **Pixel Context** window will be populated. From here click the **Debug button** and you will be able to follow exactly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

### Part 4D

In part 2A you created a Y rotation matrix that slowly rotates over time. It is now time to put that matrix to use!

On the CPU side, set the world matrix of the second mesh to be equal to this rotating matrix. This matrix will be used for the top mesh (the logo). This should be fairly simple – all you need to do is update the world matrix in your MeshData struct correctly before the map/memcpy/unmap you’re already doing to update the material.

After you do this, you should see the top part of the logo spinning. The directional lighting should change based on the rotation of the logo. If it doesn’t, make sure you’re correctly transforming the normal attributes into world space.



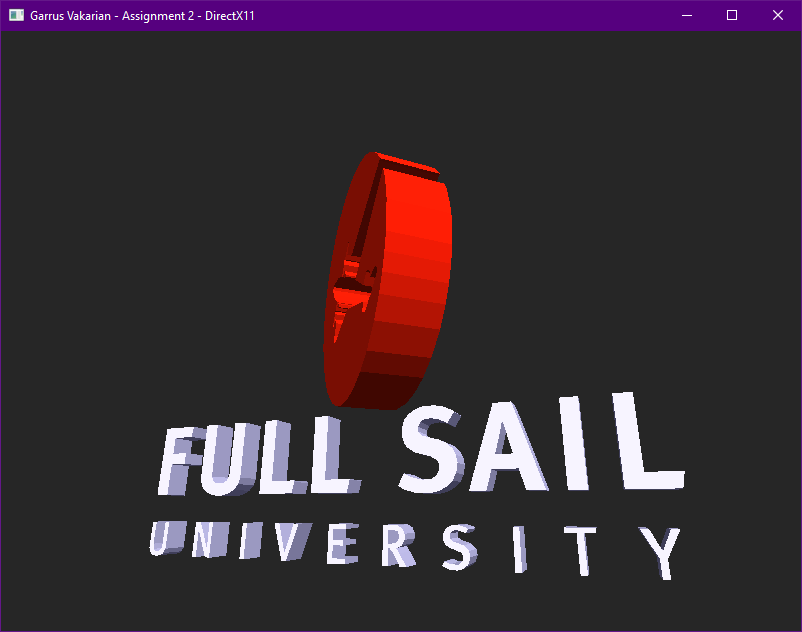
### Part 4E

For the final steps of this assignment, we will be cleaning up and enhancing the lighting effects. Start by adding two new vectors to your SceneData struct.

We will need an ambient component for our scene’s diffuse lighting (I called mine **sunAmbient**), and we will also need to know our camera’s position in world space (ex: **cameraPos**). The former will be used to inject indirect scattered light in the scene, and the latter will be used to compute the amount of reflected light bouncing off our model’s surface and into our camera.

Set your sun’s ambient term to 25% red, 25% green, and 35% blue. Initializing the camera’s world position should be self-explanatory.

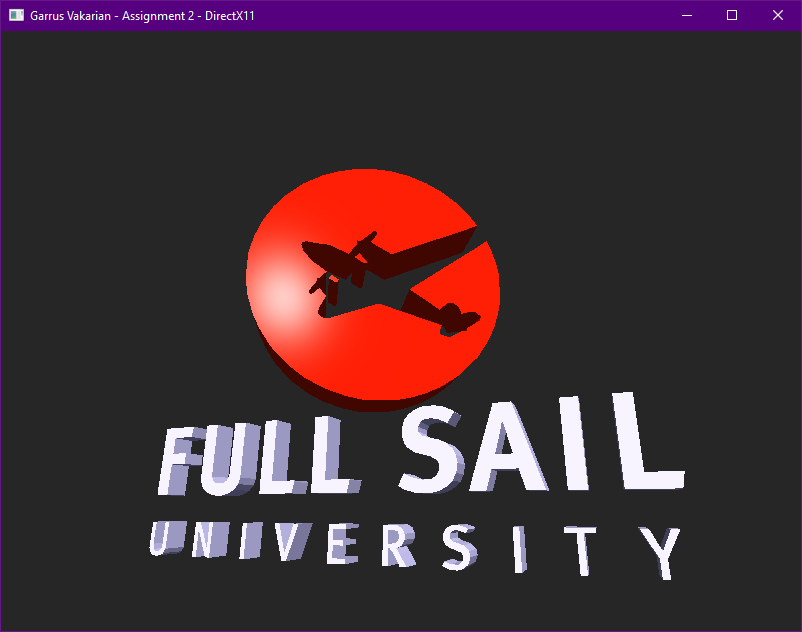
Use the new ambient term in the pixel shader to compute the total amount of light striking a pixel before multiplying it by the surface color. If you do this correctly, it should look like the picture below. (*If you need a refresher on the ambient term, again check CGS day 7*)



***Note:*** *If the ambient is correct, no part of the image should be fully devoid of light.*

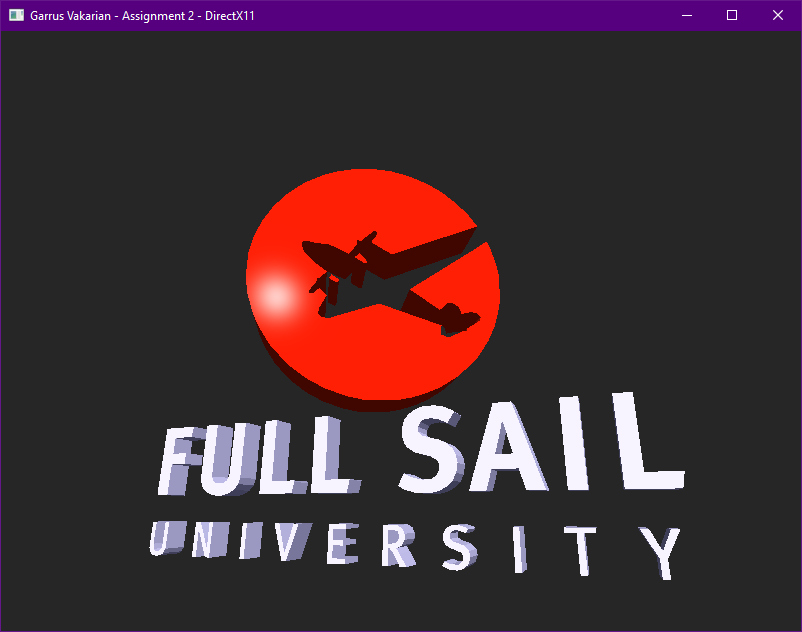
### Part 4F

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**.



***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:



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

# Summary

Great job! It is time to start thinking about if you want to stick with D3D11 for your Level Renderer. All GPU APIs have advantages and disadvantages. APIs like Vulkan and D3D12 are more complex but also more efficient, while APIs like D3D11 and OpenGL are older in design but easier to use and still plenty capable of most rendering tasks.

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.

## Direct3D11 API

<https://docs.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11> (Main Docs)

<https://github.com/walbourn/directx-sdk-samples> (Official GitHub API Samples)

<https://youtube.com/playlist?list=PLnSiYb0Vwn6T0jcOD_3EQyO5s2bkNWYz5> (Full Sail D3D11 Lectures)

<https://youtube.com/playlist?list=PLnSiYb0Vwn6Q6T6lnhOBJRhWZ_M0L1J8q> (Full Sail D3D11 Tutorials)

<https://drive.google.com/drive/folders/1LqBpN0VN50peY1TtLxH_C6qgACcXQBTt?usp=sharing> (Full Sail Samples)

<https://www.d3dcoder.net/> (Frank D. Luna has been writing excellent books on DirectX for a long time)

## 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 cross-platform API often contributed to by students here at Full Sail just like you. (Designed for 3D Engine builders) *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 Direct3D11 API correctly?
  + Aside from reading the docs and making sure the code compiles, we have enabled run-time debug output in the Direc3D11 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 Direct3D11 runtime and printed there.
* 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.
* I am having trouble getting my shader variables (matrices/materials/lights) into the shader, advice?
  + From the obj header the only data you need to upload is the OBJ\_ATTRIBUTES data not any string data (one material per mesh). When adding the matrix & lighting data make sure you are following the 16-byte padding rules carefully (see lecture 4 for the details). Make sure you are using Map/Unmap between draws.
* 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, open it and browse for your debug executable file. 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 it possible to do these assignments without Gateware? I prefer to do things from the ground up.
  + Technically yes, practically no. While someone(Andre Reid) did originally have to write the Direct3D11 interface to Gateware, setting up a modern Graphics API 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.