Assignment 2 Direct3D11

# Introduction

**\*IMPORTANT\*** Unlike every other API, the DirectX11 assignments in this course are NOT explicitly guided from start to finish. If that is what you are looking for then you should select a different API now.

There are two reasons for this: **1.)** Some students prefer to have a loose guide to an assignment while trying to puzzle out the details on their own. **2.)** 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)

This means an additional level of challenge is required to stay remotely similar in difficulty to the other APIs available. However, if you want a Full Sail curated learning experience more akin to the CGS course, then D3D11 is likely your best option. (You will need to invest **significant** **time** watching these videos however)

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%

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.

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)

Now that you are familiar with how the geometry is being supplied, use your understanding of the **rule of three** to load it into a vertex & index buffer and draw it to the screen. You will need to **temporarily** adjust the outgoing **vertex’s** position, 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%

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.*

Combine the above data into a custom structure that also contains one material to be used during rendering (Ex: OBJ\_ATTRIBUTES). Use your understanding of **constant buffers** to load this data into a GPU buffer and expose it to both your vertex and pixel shaders. (Be sure to *Set* the buffer before attempting to use it)

Once this data is available to the vertex shader use the standard 3d world/view/projection transform mathematics to make the logo appear 3D. (Don’t forget to remove the temporary shader adjustments from the last step)

Logo

Description automatically generated

***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%

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 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.

Once you have each mesh drawing separately, use the **Map/Unmap** or **UpdateSubresource** commands to change the attribute values of each mesh’s material to the correct settings right before drawing it.

Logo

Description automatically generated

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

***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 just overwriting data each draw, you can use SV\_InstanceID to select the correct mesh data more easily for techniques like Instancing. (Worth considering for the Level Renderer)*

Part 4 | 100%

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.

Here is an example of what kind of information you will need to have interpolated to your pixel shader:

Text

Description automatically generated

***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****.*

Now we should have everything we need to apply a **directional light & specular formula** to each of our pixels. Assuming you did not memorize these formulas, they were 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)

We will also 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 need a refresher on the ambient term, again check CGS day 7*)

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

***Tip:*** *HLSL has the* ***reflect*** *intrinsic built directly into the language. Use it for more accurate specular reflections.*

Complete this assignment by having the top part of the **Logo spin** slowly on the **Y axis** over time. The lighting should change based on the rotation of the logo. If it does not, make sure you are correctly transforming the **normal and position** lighting attributes into **world space**. (You may want to do this part before adding **specular**, so the bright highlights are much easier to see)

# 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)

[..\..\..\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 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.
* 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.
* 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 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.