Assignment 1 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 build a 3D grid procedurally and then use the core mathematics of computer graphics (world/view/projection) to view the shape in three dimensions. Once the grid creates a 3D shape, we can add fully 3D camera controls.

# 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 1

## Part 1 | 25%

Draw a 3D Grid in **NDC** spanning from -0.5f to +0.5f on the X and Y planes. (Z should be 0.0f, W should be 1.0f)

You will need to adjust the Rule of Three and your Vertex Shader to make this possible.

The grid will need a density of at least **25 horizontal and 25 vertical squares** so for loops are recommended to build the required points. The 2D grid should span exactly half of **NDC**.

A picture containing table

Description automatically generated

## Part 2 | 50%

Use any math library of your choice to create a matrix that rotates exactly **90 degrees around the** **X axis and translates down the Y axis 0.5f units**. Assign the combined matrix to a new class variable, this matrix will be the first of four unique World matrices.

Then use the math library from to create a **View Matrix** so we can see the scene from above(**+Y**), back(**-Z**) and to the right(**+X**). You can do this the same way you did in **CGS day 4** by placing a world space matrix where you want the camera to be and then taking its inverse.

*Tip: there is a function in most 3D math libraries designed to make this process even easier, see if you can spot it*

Essentially you want to build a **camera matrix** that has been **moved backwards, up and to the right**. Then you want to **rotate the matrix slightly to the left and down** so its forward(**+Z**) vector is pointing towards the origin.

After the matrices have been generated, you will need to upload them to the GPU using the D3D11 **ConstantBuffer** system. This should allow you to attach them to the appropriate **cbuffer** in the shader.

***Important:*** *By default, the* ***HLSL*** *language treats matrix data as* ***column major****. Most math libraries are* ***row major****.*

Once you have successfully transmitted your **view matrix** and **world matrix** to the GPU, you should be able to multiply your vertex data into **view space** successfully. Of course, this is done much in the same way as you did it in your first **vertex shader** in CGS.

Once your grid is both in **world** and **view space** it should look something like this:

A picture containing shape

Description automatically generated

To create this image, I placed my camera at **0.25x -0.125y and -0.25z** and angled it so it **Look**s **At** exactly the **center of the grid** after it has been moved into place.

***Important:*** *The conceptual* ***near and far planes*** *do not exist yet, so anything outside the* ***Z range of 0-1*** *will not be drawn. Because of this we will need to choose camera values between* ***-0.5f to +0.5f*** *if we hope to see anything.*

## Part 3 | 75%

In this section we are going to learn how to add perspective to our scene and make it a bit more complex visually by learning how to draw our Grid multiple times in different locations.

Let’s start by using the math library to create a **left-handed perspective projection matrix** specifically for the D3D11 API. Create a matrix variable to store our new matrix and initialize it using the following settings:

**Vertical Field of View:**  65 degrees

**Near Plane:**  0.1 units

**Far Plane:** 100 units

**Aspect Ratio:**  GDirectX11Surface::GetAspectRatio()

Create **five** additional **world matrices** using the same methods from [Part 2](#_Part_2_|). They should be setup so that you have a **ceiling** and **four vertical sides** all connected along the edges. Use combinations of **translations and rotations** to carefully place each wall segment in the same way you manipulated the placement of the original grid.

**Update the world matrix** in the vertex shader before **re-drawing the same grid** in the five new locations/orientations. If successful, you should see some walls appear.

Background pattern

Description automatically generated

***Tip:*** *You will need to update the shader’s matrix between each draw call if you expect it to draw somewhere else.*

## Part 4 | 100%

In the final section of this assignment, we will learn to add both **Keyboard and Mouse** support as well as **Game Controller** support via the Gateware API. Having any PC compatible **XBox controller** is recommended for this step, but only a Keyboard and Mouse are strictly required. (*We recommend also adding code for the controller however*)

Read the Gateware **README** and provided **documentation** to determine how to enable keyboard/mouse & controller support. Once you can successfully read user inputs, use them to build a first-person fly-through camera like most “lobby/ghost cameras” in first-person shooter games.

Motion should be based on time not fps, use your understanding of local vs global matrix operations to move or rotate along the appropriate axis. The final camera should behave as close as possible to the provided sample executable. Your score on this section will depend on how closely the behavior mimics the demo program.

A picture containing text, indoor, electronics, tiled

Description automatically generated

*You should now have total control over your camera matrix. With both PC and Console style FPS input. ☺*

# Summary

Excellent! You now know how to create and navigate a 3D environment using the GPU. All the big-name games you play are built on top of this same fundamental foundation! Direct3D11 is still one of the most used APIs found on Windows & Xbox compatible games. It’s a great skill to possess.

The second introductory assignment in this course will have you loading your first 3D Model and applying a basic lighting algorithms using the flexibility of the pixel shader. This knowledge will be crucial when rendering 3D levels.

# 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

* I’m trying to use std::chrono<> to create proper time-based camera movement, but it is choppy. Advice?
  + Try using the high\_resolution\_clock feature to get more accurate time intervals.
  + Sample Code: [https://www.cplusplus.com/reference/chrono/high\_resolution\_clock/now/](https://www.cplusplus.com/reference/chrono/high_resolution_clock/now/%20)
* 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 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 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.