Assignment 1 Direct3D12

# Introduction

In the first guided assignment in this course, 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%

### 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

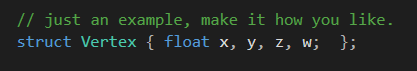
Next, we are going to switch the **TOPOLOGY** to be able to draw a **LIST** of lines. We will attempt to **draw 3 lines around the triangle**. To do this successfully you will need to increase the number of vertices you currently have.

Shape

Description automatically generated

### Part 1C

We are also going to use this opportunity to upgrade our vertex type to be **four floats instead of two.** Seeing as we do not actually have a vertex structure now seems as good a time as any to make one.



For this to work properly you will need to apply the **rule of three** to the code. First convert the existing triangle to the structure, adding zero and one for the Z and W values respectively.

Next find the **D3D12\_INPUT\_ELEMENT\_DESC** array and adjust it so that it will accept the additional position data you are now passing to the video card. There is still only one vertex attribute in use, so we need not add additional items like UVs and Normals for now. (But this is where we would)

The last part of the rule of three is to adjust the **vertex shader** itself. **Modify the vertex shader’s input to accept your additional data**, even if you are not really doing much with it right now. (*Challenge: Instead of just switching to a float4 use a custom struct matching the one in C++)*

### Part 1D

Now that we can successfully draw 3D lines where we want, we are going to draw a grid using our lines which will serve as the eventual walls to our 3D “room”. To do this you will need to significantly increase the number of vertices you copy to the **ID3D12Resource** used to store the vertex buffer**.** 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 shape

Description automatically generated

***Note:*** *You’ve probably just written a block of code that procedurally creates a grid. That sounds like a single responsibility to me! To keep our renderer’s initialization clean, it’s a good idea to extract this functionality out to a well-named helper function.*

## Part 2 | 50%

### Part 2a

Our next goal is to apply 3D World, View and Projection mathematics to our new shape. In the interest of time (and since we don’t have to go download anything) we will use Gateware’s built-in math library. (not strictly required, though this guide assumes you did)

To enable it, go to main.cpp and **#define GATEWARE\_ENABLE\_MATH** above the “Gateware.h” include. Gateware has a 4x4 matrix struct called **GMATRIXF** it is part of the **MATH** namespace, add one to the Renderer class.

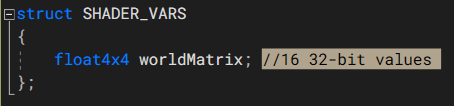
You will also need an interface proxy called **GMatrix** to access the math routines. In the constructor call **Create()** on the proxy to enable it. (Not strictly necessary for the math libraries but a good habit to get into)

After initialization use the matrix operations 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.

### Part 2B

In C++ make a struct called SHADER\_VARS (or similar). And place one GMATRIXF inside it to represent your world matrix. Once that is done transfer the initialized World matrix from the last part to an instance of this structure in the **Render()** function so you can eventually send it the vertex shader.

Declare an almost identical version of this structure in your **HLSL** vertex shader. A single 4x4 matrix can be represented in the language by “float4x4” or just “matrix”.



Now switch the keyword **struct** with **cbuffer**(constant buffer), this tells HLSL that you intend to supply the data from outside the GPU. In the next two parts, we will supply this data using something called a **RootSignature**, this is a block of memory used to communicate both variables(**RootConstants**) and resource links(**RootDescriptors & RootTables**) to a specific shader.

### Part 2c

Before using the GPU-matrix we just declared in the shader, we must upload our CPU World matrix data to the shader’s GPU memory block. To do this we will use something called **Root Constants**. Root constants are a way to upload a minimal amount of CPU memory to a shader without having to go to the trouble of allocating a separate buffer for **uniform**(shader variable)**data** or a **ID3D12DescriptorHeap** to reference that memory in D3D12.

Create a **CD3DX12\_ROOT\_PARAMETER** structure above the existing **CD3DX12\_ROOT\_SIGNATURE\_DESC.** Use its **InitAsConstants(…)** function to describe your custom **SHADER\_VARS** structure, how much room it needs and where it is going. Pass the newly initialized structure to the CD3DX12\_ROOT\_SIGNATURE\_DESC’s **Init(…)** function. (“CD3DX” indicates a wrapper class to simply filling out a D3D12 structure. Remove the “C” and the “X” figure out what underlying D3D12 base structure is being filled out to learn more about it)

***Note:*** *Root Constants are fast & convenient but very limited in size. Only 64 32bit values can be stored in a RootSignature, additionally this space must be shared with any inline descriptors or links to descriptor heaps.*

### Part 2d

In **Render()** we should now be able to call **SetGraphicsRoot32BitConstants()** and give it the address of our structure we made way back in [Part 2B](#_Part_2B). This should upload the data directly to GPU shader memory(no additional buffer required!).

Finally, we add the **HLSL** code required to use the matrix we have provided to the Vertex Shader. In the HLSL reference look-up **shader intrinsics**, these are the math routines built directly into the language. You want the **mul** command, it is used for both matrix to matrix and vector to matrix multiplication.

Fix any compiler errors in your shader and you should see your **grid go flat and move down slightly:**

Graphical user interface, text, application

Description automatically generated

**Important:** By default, the **HLSL** language treats matrix data as **column major**. Most math libraries are **row major**. You will need to do **one** of the following: **transpose**, **mul( Matrix, Vector )**, **#pragma pack\_matrix( row\_major )**

### Part 2E

Now that the grid appears to be following the instructions of our world matrix, let’s use this opportunity to **view the scene** from a different angle so we can get a better look at our grid.

Use the math library from earlier 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 the math library 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.

***Note:*** *You’ve just written a block of code that initializes a view matrix. That sounds like a single responsibility to me! To keep our renderer’s initialization clean, I recommend extracting this code out to a well-named helper function.*

### Part 2F

Our new matrix is no good to us if we can’t use it, thankfully we can just add it to our **SHADER\_VARS** structure so it will be uploaded alongside our world matrix. (Adjust it appropriately)

Once you have successfully transmitted your **view matrix** using the same data pathway as your **world matrix** 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 text, accessory, envelope, businesscard

Description automatically generated

**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.

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.

## Part 3 | 75%

### Part 3a

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 D3D12 API. Create a GMATRIXF 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:**  GDirectX12Surface::GetAspectRatio()

***Note:*** *You’ve just written a block of code that initializes a perspective matrix. That sounds like a single responsibility to me! …You probably get the picture by now. As you continue writing code, any time you’ve finished a block that has a single responsibility, extracting that block out into a helper function will help keep your codebase from becoming a mess. This isn’t necessarily crucial for small solo projects, and you won’t be docked points for messy code, but a clean codebase is much easier to debug and work in. This will be especially important when you move on to DEV4 and join a team.*

### Part 3B

Like our **view matrix** we now have another matrix we wish to apply to our vertices, to do this we will need to upload it or combine it with the existing view matrix. This works because matrices are **Associative.** For example: **(matA \* matB) \* matC == matA \* (matB \* matC)**

We can use this to our advantage in this situation by multiplying the separate view and projection matrices temporarily into a single **viewProjection** matrix and sending that instead.

You can also just send the projection matrix by itself in addition to the World & View since there are **256 bytes** available in the Root Signature. (Room for 4 matrices) If you do this be sure to adjust your Root Constant settings.

Unlike some other APIs, D3D12 provides full access to your GPU’s Root Signature memory. The downside is that this memory is shared with pointers to **descriptor heaps** and **descriptor tables**. So… the more of it you use, the less you can link to other shader resources.

Background pattern

Description automatically generated

### Part 3C

Great! We are now seeing a mathematically correct 3D environment for the first time. Let’s make it a bit more interesting by adding some **walls** to our **floor**. ☺

Create **five** additional **world matrices** using the same methods from [Part 2A](#_Part_2a). 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.

### Part 3D

After drawing the current grid (A.K.A the floor) draw the remaining sections by **updating the world matrix** in the vertex shader using **SetGraphicsRoot32BitConstants** 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%

### Part 4a

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. (*You will still need to add the code for the controller however*)

To start we will need to create two interfaces to access user input data. Go to main.cpp and **#define GATEWARE\_ENABLE\_INPUT** above the “Gateware.h” include. In Renderer.h add the **GInput** and **GController** proxy objects to your class as member variables.

Once you have added these items to your class definition, go to the constructor and **Create()** both objects.

### Part 4B

At this point we should hopefully have access to reading state from the keyboard, mouse, and a game controller. Before we use this information lets ensure we keep the code somewhat clean as we will be adding a decent amount of state query and math code to move the camera around.

Add a public **UpdateCamera()** function to our Renderer class. This will be used to isolate the user input and camera manipulation code. At the top of this function use **std::chrono** to query the amount of time that passes from one call of this function to the next. If you’re unsure how to use the standard libraries to achieve this, you can also grab the **XTime** class from CGS, just be aware that unlike std::chrono this class is Windows only.

The last thing to do is call this function from **main.cpp** right before rendering. This ensures the user has a chance to move the camera each frame before we render our 3D scene.

### Part 4C

To correctly manipulate our existing view matrix, it will need to be placed temporarily be in **world space** otherwise all the movements will seem to be inversed from normal. As you might imagine this can be resolved by grabbing a copy of the view matrix after it has been **inversed**. Once we are fully done manipulating the matrix be sure to place it **back into view space** by taking the inverse of our newly manipulated **camera** (A.K.A inversed view) and assigning the actual view matrix to that.

### Part 4D

We start with a very basic movement, just moving the camera directly **up and down on the Y axis**. Open the **Gateware docs** look over all the available input codes in the **Input** namespace. Alternately you can search Gateware.h for **GInputDefines.h** where all the codes are listed.

Inside the **UpdateCamera** function create a single float designed to represent how much we wish to change the **Y** value this frame and initialize it to zero. We can also create a **const float** called **Camera\_Speed** that represents how far we want the camera to be able to move over one second. (*I settled on* ***0.3*** *units per second*)

To implement camera motion, read the following values from the user input using the .**GetState()** functions:

Total\_Y\_Change = SPACE\_KEY\_STATE – LEFT\_SHIFT\_STATE + RIGHT\_TRIGGER\_STATE – LEFT\_TRIGGER\_STATE

Camera.Position.Y += Total\_Y\_Change \* **Camera\_Speed** \* Seconds\_Passed\_Since\_Last\_Frame

A picture containing shape

Description automatically generated

*You should now be able to make the camera move up or down with Space/Shift or the triggers on your controller.*

### Part 4E

While moving up and down is fairly simple no matter which way we are looking; going **forwards and backwards** and **strafing side to side** will be a bit more complicated. This is because this movement changes based on the orientation of our camera.

On **CGS day four** I covered the fundamental difference between **Local** matrix operation vs. **Global** matrix operations. If you don’t remember this section of the video, I highly recommend you go back and re-watch it. (*It was only about 15 minutes*) In this scenario we will need to use **Local Translation**to achieve the desired effects.

To implement local translation, read the following values from the user input using the .**GetState()** functions:

PerFrameSpeed = **Camera\_Speed** \* Seconds\_Passed\_Since\_Last\_Frame

Total\_Z\_Change = W\_KEY\_STATE – S\_KEY\_STATE + LEFT\_STICK\_Y\_AXIS\_STATE

Total\_X\_Change = D\_KEY\_STATE – A\_KEY\_STATE + LEFT\_STICK\_X\_AXIS\_STATE

TranslationMatrix( Total\_X\_Change \* PerFrameSpeed, 0, Total\_Z\_Change \* PerFrameSpeed)

Camera = MatrixMultiplication( TranslationMatrix, Camera )

A picture containing background pattern

Description automatically generated

*Forward/Backward and Left/Right Strafing camera behaviors should now be available to your camera system.*

### Part 4F

You can probably guess the last thing we will need for a fully functional 3D Camera. That’s right… **rotation!**

We’re going to start by adding the ability to **tilt the camera up and down:**

Thumb\_Speed = PI \* Seconds\_Passed\_Since\_Last\_Frame

Total\_Pitch = FOV \* MOUSE\_Y\_DELTA / SCREEN\_HEIGHT + RIGHT\_STICK\_Y\_AXIS\_STATE \* -Thumb\_Speed

PitchMatrix( Total\_Pitch )

Camera = MatrixMultiplication( PitchMatrix, Camera )

Shape

Description automatically generated

*Tilting the Camera Up and Down should no longer be an issue.*

### Part 4G

All that is left is to allow the camera to **turn left and right**. On the Y axis **global rotation** is the more desirable behavior if we are looking to create an **FPS style** camera as opposed to a space flight style camera.

We finish by adding the ability to **yaw the camera left and right:**

Total\_Yaw = FOV \* AR \* MOUSE\_X\_DELTA / SCREEN\_WIDTH + RIGHT\_STICK\_X\_AXIS\_STATE \* **Thumb\_Speed**

YawMatrix( Total\_Yaw )

Camera.SavePosition()

Camera = MatrixMultiplication( Camera, YawMatrix )

Camera.RestorePosition()

A picture containing text, electronics, computer

Description automatically generated

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

***Note:*** *It’s arguably cleaner to move the input handling and camera movement logic out to its own class. This is not required and adds a bit more challenge, but will result in your program being more modular and flexible. An ideal renderer class only handles drawing models – input and movement are good candidates to move out into a separate class. (It will also make the code easier to cleanly add to your level renderer).*

*If you want to do this, I’d recommend getting your input handling and movement code* ***working*** *first (and committing/pushing your changes),* ***then*** *moving the functionality out. This will allow you to have a better idea of what information this class will need to provide (as part of its public interface), and what parts it can cleanly encapsulate.*

# Summary

Nice! 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! Direct3D12 has quite a lot going on, but it is a very powerful and flexible API that many Windows & Xbox compatible games have adopted.

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.

## 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)

<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)

<https://gateware-development.gitlab.io/gcompiler/index.html> (Official Documentation)

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

# FAQ

* 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 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.
* 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(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.