Assignment 1 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 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

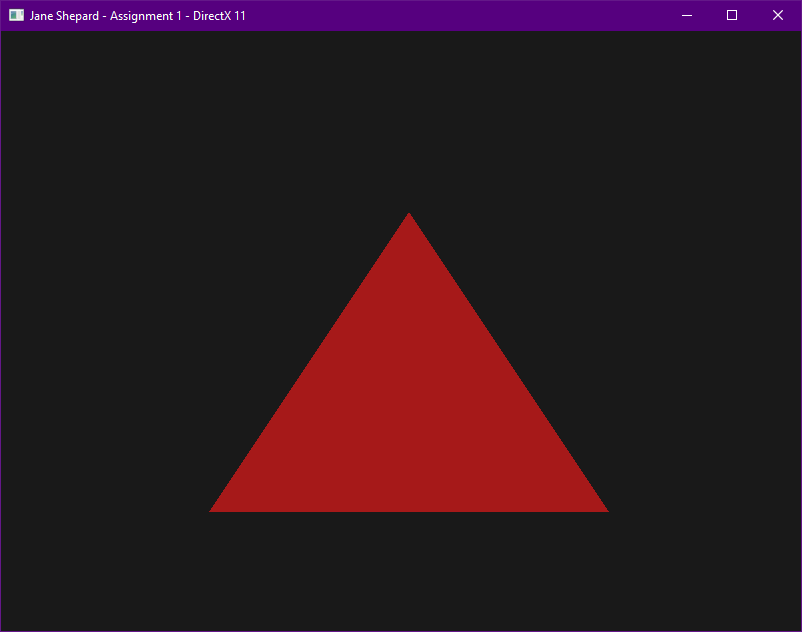
## Part 1 | 25%

### Part 1A

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

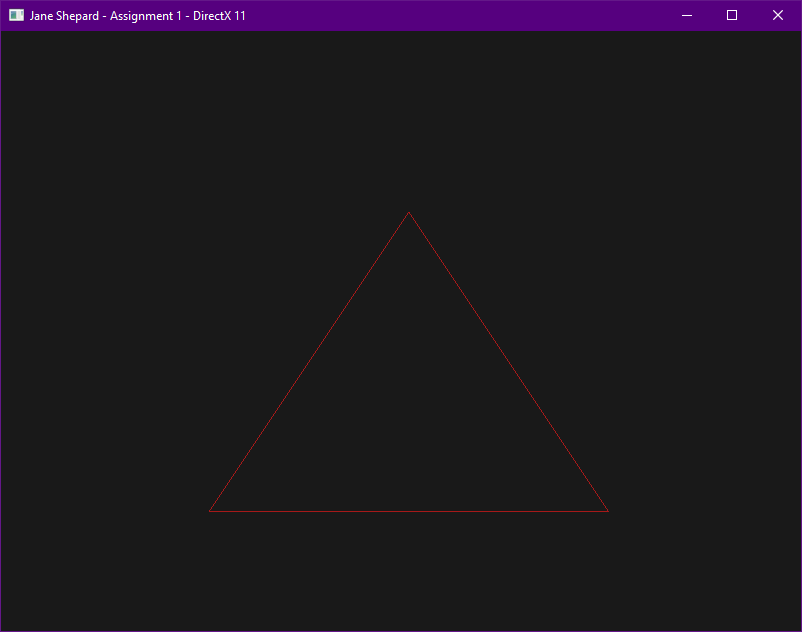
(optional) Change the triangle and background colors to some colors you like.

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



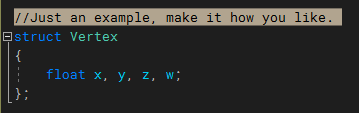
### Part 1B

Next, we’re going to switch the pipeline to draw a list of lines. We will attempt to **draw 3 lines around our current triangle**. To do this, you will need to change the primitive topology type – you may also need to increase the number of vertices that you currently have.



### Part 1C

We’re going to use this opportunity to upgrade our vertex type to be **four** floats instead of **two**. Seeing as we don’t currently have a vertex struct, now is a good time to make one.



For this to work correctly, we will need to apply the **rule of three** to our code.

First, convert the existing vertices to the new struct – add 0 for the z component and 1 for the w component.

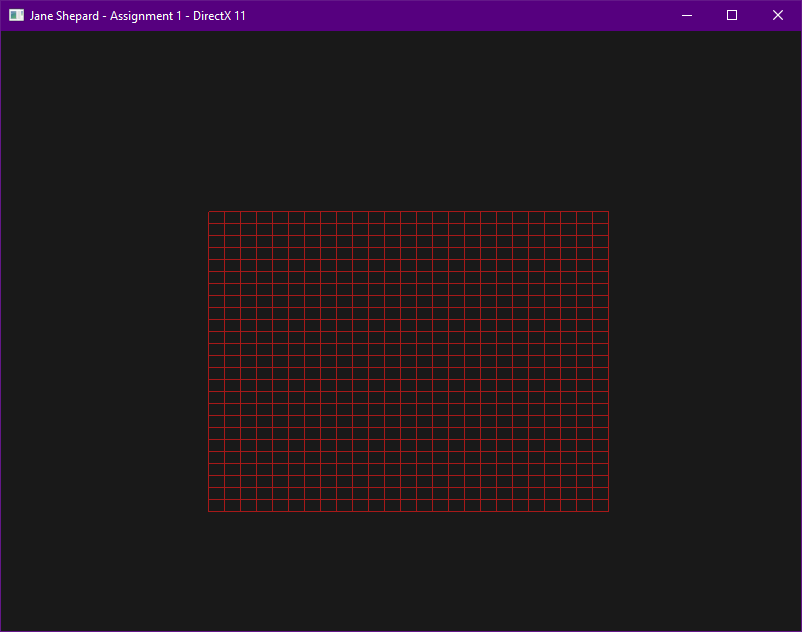
Next, find the **CreateInputLayout** call and adjust the **D3D11\_INPUT\_ELEMENT\_DESC** so that it correctly reflects the layout of your vertex struct. We only have a single attribute at the moment (position), so we don’t need any additional descriptors yet. But if we had other attributes, this is where we would add them.

After the input layout has been corrected, find where the **stride** of your vertices is specified to the API. The **stride** describes how large a vertex is (how many bytes the API should *stride* to step from one vertex to the next). Make sure that the stride you’re providing the API accurately reflects the actual size of your vertex.

The last part of the rule of three is to adjust the **vertex shader** itself. Modify the existing shader to take in the full 4-component vector and return it. We will be making further adjustments to the shader later to do the necessary world/view/projection 3D transformations.

### Part 1D

Now that we can successfully draw 3D lines where we want, we’re going to draw a grid which will eventually serve as the walls to our 3D “room”. To do this, you will need to significantly increase the number of vertices you’re writing into the vertex buffer. The grid will need a density of at least **25 horizontal and 25 vertical squares** – using one or more for loops are recommended to create the required vertices. The 2D grid should span exactly half of **NDC**.



***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 grid. In the interest of time, we will use Gateware’s built-in math library. (Not strictly required, but this guide is written assuming that you did).

Go to main.cpp and **#define GATEWARE\_ENABLE\_MATH** above the “Gateware.h” include – this will enable Gateware’s math library.

Gateware has a 4x4 matrix struct called **GMATRIXF** in its **MATH** namespace. This is the struct we will use for our matrices. Add one to the Renderer class.

There is also a proxy class called **GMatrix** that will give you convenient access to Gateware’s matrix math library. You can add one to the Renderer if you want – be sure to call **Create()** on the proxy to enable it. (This isn’t strictly necessary since all the math functions are static, but setting up proxies is a good habit to get into).

Use Gateware’s matrix operations to create a matrix that is rotated exactly **90 degrees about the x axis** and **translated down the y axis by 0.5f units**. Assign this matrix to your new class member. This will be the first of six unique world matrices.

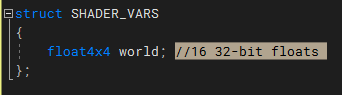
***Note:*** *You’ve just written a block of code that initializes your world matrix. That sounds like a single responsibility to me! To keep our renderer’s initialization clean, I recommend extracting this functionality out into a well-named helper function.*

### Part 2B

Now we need to push this world matrix over to the GPU if we want the shader to be able to use it. To do this, we will use a cbuffer (constant buffer). This is another place where we will have to be mindful of the **rule of three**.

To get started, let’s create a C++ struct to hold the variables we’ll be pushing to the shaders. Name your struct SHADER\_VARS or something similarly communicative. Inside the struct, place one **GMATRIXF** to hold our world matrix.

Next, declare an almost identical struct in your vertex shader. In HLSL, a 4x4 matrix of floats is specified as a “float4x4” or just “matrix”.



Now, switch out the keyword **struct** with **cbuffer**, this tells HLSL that you intend to supply the data from outside the GPU. In the next steps, we will supply this data.

### Part 2C

Now that you’ve declared a struct to hold the data our shaders will need, add an instance of that struct to the renderer. Make sure to initialize the matrix it contains to the world matrix you’ve already created.

### Part 2D

Alright. We’ve designated a spot in our shader to receive data, and we have the data we want to write. Now we need to link our C++ code to that spot. The C++ struct that DX11 provides to assist with this linkage is **ID3D11Buffer** – this struct is also used for the vertex buffer.

Add a **ComPtr<ID3D11Buffer>** to the renderer class for the constant buffer. Then add a call to CreateBuffer() – this will tell the API to prepare a block of memory on the GPU for our buffer. To call this function, you will need to provide a **D3D11\_SUBRESOURCE\_DATA** and a **CD3D11\_BUFFER\_DESC**. You can review the **CreateVertexBuffer**() function that’s provided in the template for an example of how to use these structs. Make sure your **CD3D11\_BUFFER\_DESC** is correctly describing a constant buffer that the CPU can write to.

### Part 2E

Now that we’ve created a constant buffer on the GPU and populated it with data, we must tell the API to link it to the shaders that will use it. For now, the only shader that will use this constant buffer is our Vertex Shader, so it’s the only one we need to link. We can do this via a call to **VSSetConstantBuffers()** – this method is part of the ID3D11DeviceContext.

***Note****: A similar [X]SSetConstantBuffers() method exists for each type of shader – pixel (PS), compute (CS) , domain (DS), etc.*

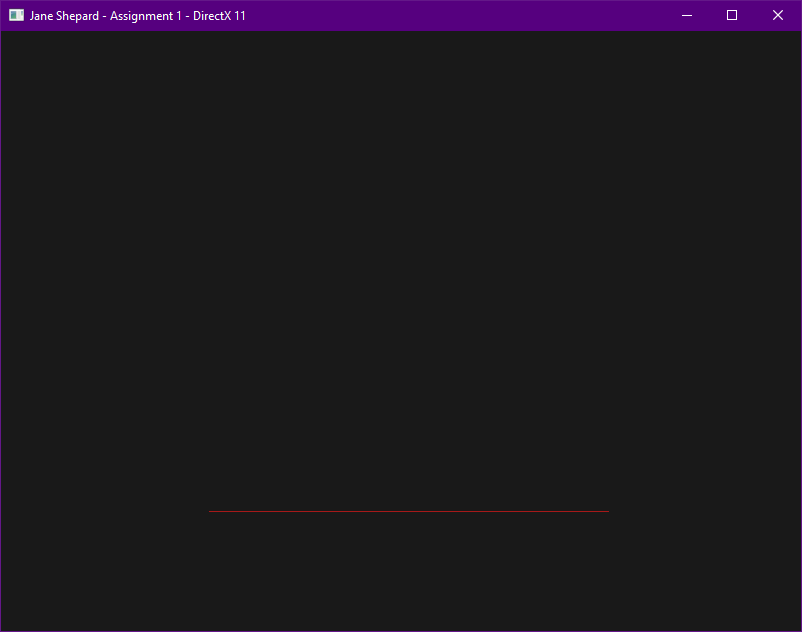
### Part 2F

Finally, we can write the **HLSL** code to use the matrix we’ve provided to the Vertex Shader. In the HLSL reference, look up **shader intrinsics**, these are the math functions built directly into the language. You want the **mul**() function – it is used for both matrix-matrix and vector-matrix multiplication.

Once your Vertex Shader is working correctly, you should see your grid go flat and move down slightly.

***Important:*** *Matrices in HLSL are treated as* ***column-major****, but the matrices made by Gateware are* ***row-major****! You will need to correct for this – you can do so by:*

* *Adding* ***#pragma pack\_matrix(row\_major)*** *to your shader (recommended)*
* *Transposing your matrices before uploading them to the GPU.*
* *Doing your math in reverse order - (matrix \* vector) instead of (vector \* matrix).*



***Note:*** *The \* operator does exist in HLSL for vector and matrix types, but it is not used for linear algebra transformations. It does component-wise multiplication - (x \* x, y \* y, z \* z). This is the correct behavior for multiplicative* ***color*** *mixing, but not spatial transformation.*

### Part 2G

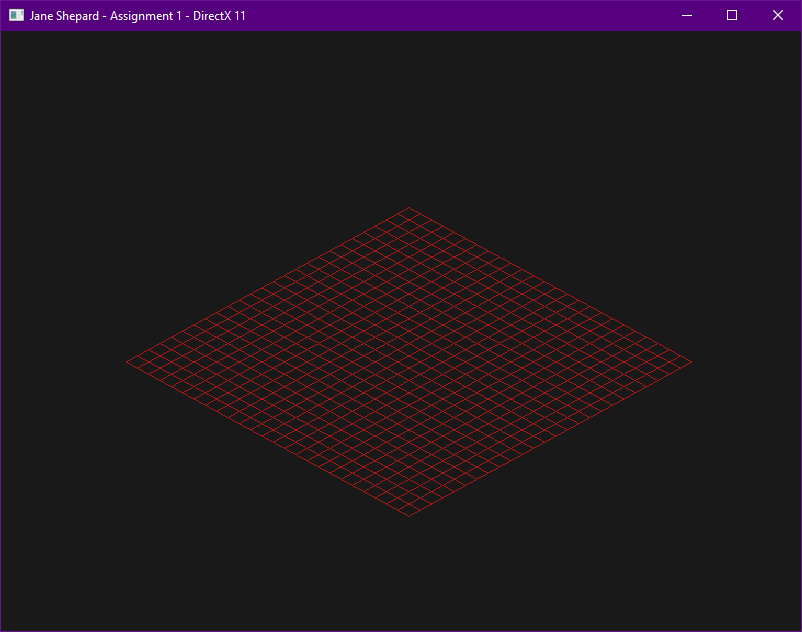
Now that we have a world matrix set up, let’s add our view matrix. Start by adding a space in the SHADER\_VARS for one. Do this in your C++ struct and in your HLSL cbuffer.

Next, add another GMATRIXF to your renderer class – this will be our view matrix. Initialize it so it’s positioned at **0.25x, -0.125y, and -0.25z** and rotated to **Look At** the center of your grid (after it’s been moved into place). There’s a function in Gateware’s matrix math library to do this, see if you can find it!

Now initialize your SHADER\_VAR instance’s view matrix to the view matrix we’ve just made.

Finally, edit your Vertex Shader so it uses the view matrix we added to its cbuffer.

Once you’ve done this, your grid should look something like this:



***Important:*** *Since we haven’t set up a projection matrix yet, the conceptual near and far planes don’t exist. Parts of your grid may fall out of NDC at this point and be clipped – this is okay, we’re going to be implementing a projection matrix in the following part.*

## Part 3 | 75%

### Part 3A

In this section we are going to learn how to add perspective to our scene and 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 it and initialize it using the following parameters:

**Vertical Field of View:**  65 degrees

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

**Near Plane:**  0.1 units

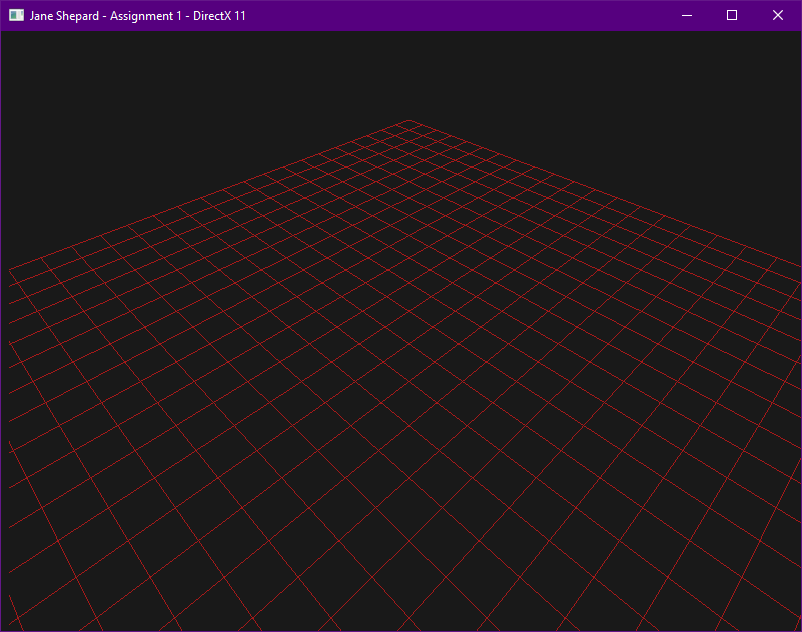
**Far Plane:** 100 units

***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

Next, we need to add this projection matrix to the constant buffer. The process is the same as we did with the view matrix. Once you’ve gotten the projection matrix successfully uploaded to the GPU, add it in to your Vertex Shader.

Once you’ve applied your projection matrix, your grid should look something like this:



***Note:*** *Instead of using two separate matrices, you can combine your view matrix and projection matrix into a single viewProjection matrix before uploading to the GPU – this is technically more optimal since you would only be doing this matrix multiplication once per frame instead of for each vertex each frame – but this is not required. This guide uses separate matrices for simplicity’s sake.*

### Part 3C

Next, we will set up our grid to be drawn five more times. We will not need additional vertices for this – all that’s needed is five more world matrices. Add five more world matrices to your class – you should set them up so you have four walls and a ceiling added to your floor. The final result should be a cube made of grids.

***Note:*** *I recommend using an array or vector to hold your world matrices. This is not strictly required, but it will save you some work later on.*

### Part 3D

Now that we have our additional world matrices, we need to draw our grid five more times, switching out which world matrix is used each time.

To do this, we will use a **D3D11\_MAPPED\_SUBRESOURCE**. “Subresource” is D3D11’s generic term for a resource that lives on the GPU. Declare a local **D3D11\_MAPPED\_SUBRESOURCE** instance inside of **Render()**, then call **Map()** on the device context and map your constant buffer. This locks down the constant buffer so we can write to it – mapping is similar to locking a mutex.

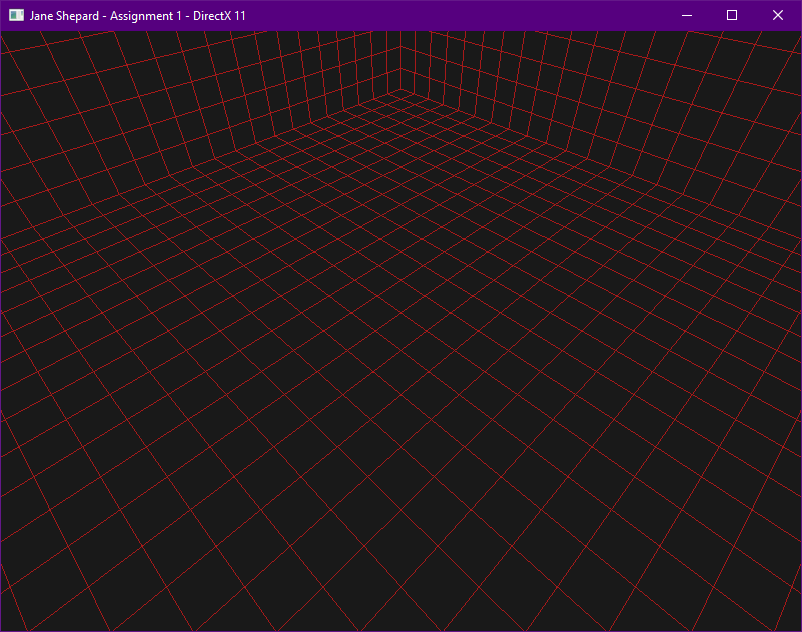
The call to **Map()** should contain your constant buffer and the local mapped subresource instance you created. Look over the official documentation for what to put for the other parameters.

Once we’ve mapped our constant buffer, we can update the world matrix in our SHADER\_VARS instance and use **memcpy** to overwrite the contents of the buffer. The address of the location we want to write to can be found in our local **D3D11\_MAPPED\_SUBRESOURCE** instance.

After the **memcpy**, call **Unmap()** on the device context. This tells the API that we’re done writing to the subresource and it’s safe to use once more. This is similar to unlocking a mutex.

Once the constant buffer is updated and unmapped, we can call **draw()** again to draw our grid using the new world matrix.

Do this process (map, memcpy, unmap, draw) for each of the six world matrices. Once you do so, your grids should look something like this:



***Note:*** *If your world matrices are contained in an array or vector, you can use a for loop for this process. If you have six different named vertices, you will have to copy and paste the code six times. Not the end of the world, but it scales poorly.*

## Part 4 | 100%

### Part 4A

In the final section of this assignment, we will 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*)

We will need two interfaces to access user input. Start by going to main.cpp and **#define GATEWARE\_ENABLE\_INPUT** above the “Gateware.h” include – this will enable Gateware’s input library. Next, add a **GInput** proxy and a **GController** proxy to your class. Once you’ve done so, go to the where your class is initialized and call **Create()** on both proxies.

### Part 4B

At this point we should have access to read the state of the keyboard, mouse, and game controller. Now, we’re about to add a sizeable amount of state querying and math to move the camera around. To help keep our code tidy, let’s make a function to wrap this code.

Add a public **UpdateCamera()** function to your Renderer class. This will be where the code related to getting and handling user input will live. At the top of this function, use the **std::chrono** library to query the amount of time that’s passed since this function was last called – this is typically referred to as *delta time*. If you’re struggling to use the chrono library to do this, you are also allowed to use the **XTime** class from CGS. Just be aware that std::chrono is multi-platform, but XTime is windows-only.

Lastly, go to main.cpp and call our new function right before rendering. This ensures the user’s input will be applied to every frame before rendering.

### Part 4C

To correctly manipulate our existing view matrix, it will need to be placed in **world space** temporarily – otherwise, all the movements will be inversed. This can be resolved by grabbing a copy of the view matrix after it’s been **inversed**. Once we are finished manipulating the matrix be sure to place it **back into view space** by inverting our newly manipulated *camera* matrix (AKA inversed view matrix) and assigning the actual view matrix to that.

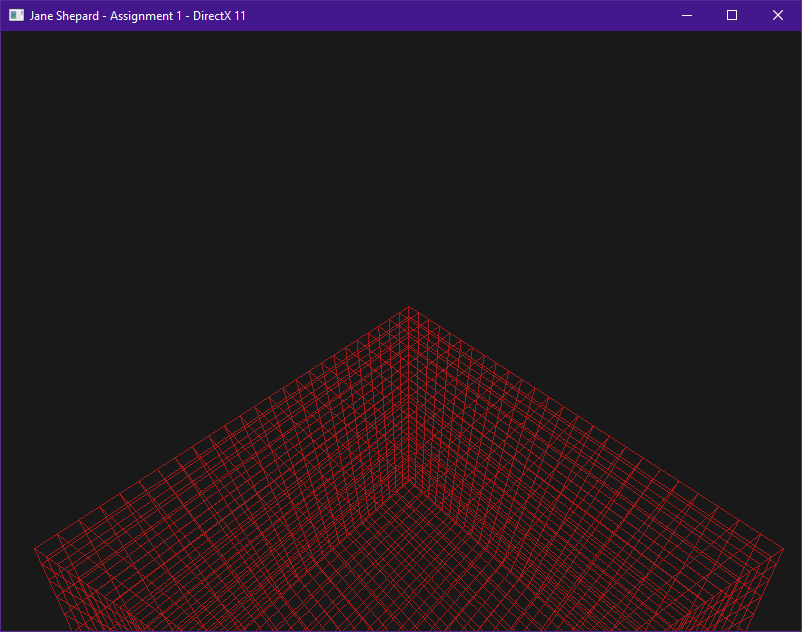
### Part 4D

Let’s start with a very basic movement, just moving the camera **up and down on the global Y axis**. Open up the **Gateware docs** and look through the input codes in the **Input** namespace. Alternately, you can search Gateware.h for “**GINPUTDEFINES\_H**” – the input codes are listed underneath.

Inside the **UpdateCamera()** function, create a float to represent how much we wish to change the camera’s **Y** position this frame – make sure to initialize this float to zero. We can also create a **const float** for our camera’s speed (this would represent how far we want the camera to move over one second – I settled on **0.3** for this value).

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



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

### Part 4E

While moving up and down globally is fairly simple, going **forwards and backwards** and **strafing left and right** will be a bit more complicated. This is because the desired movement depends on the orientation of our camera.

In **CGS**, you learned about the fundamental difference between **local** and **global** matrix operations. If you don’t remember, I highly recommend you go back and watch the **day four** recorded lecture, part 3 (*it’s about 20 minutes*). In this scenario, we want to use **Local Translation** to achieve the desired effect.

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

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

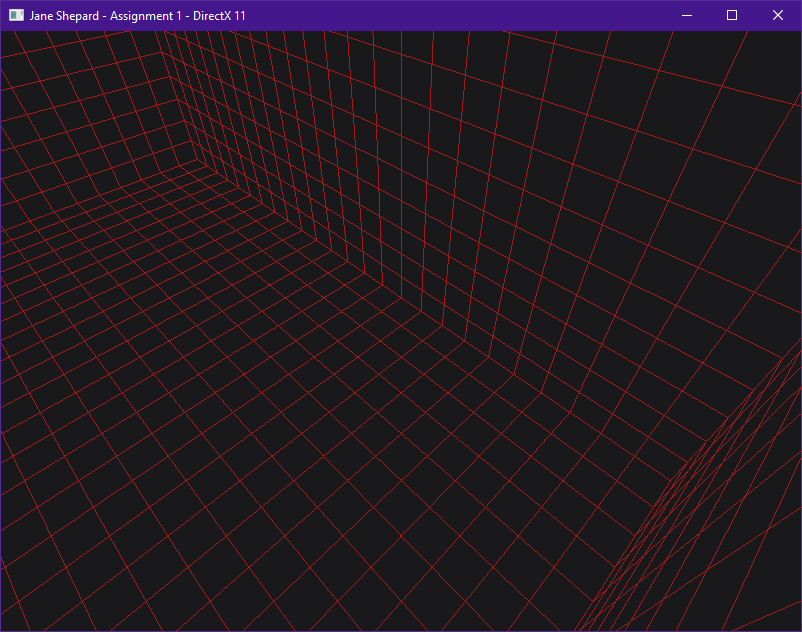
Total\_Z\_Change = W\_KEY\_STATE – S\_KEY\_STATE + LEFT\_STICK\_Y\_AXIS\_STATE

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 = MatrixMultiply( TranslationMatrix, Camera )

***Note:*** *There is a function in Gateware’s math library to make translation simpler – I encourage you to use it.*



*Your camera should now be able to move forward/backward and strafe left/right.*

### Part 4F

You can probably guess the last thing we’ll 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**:

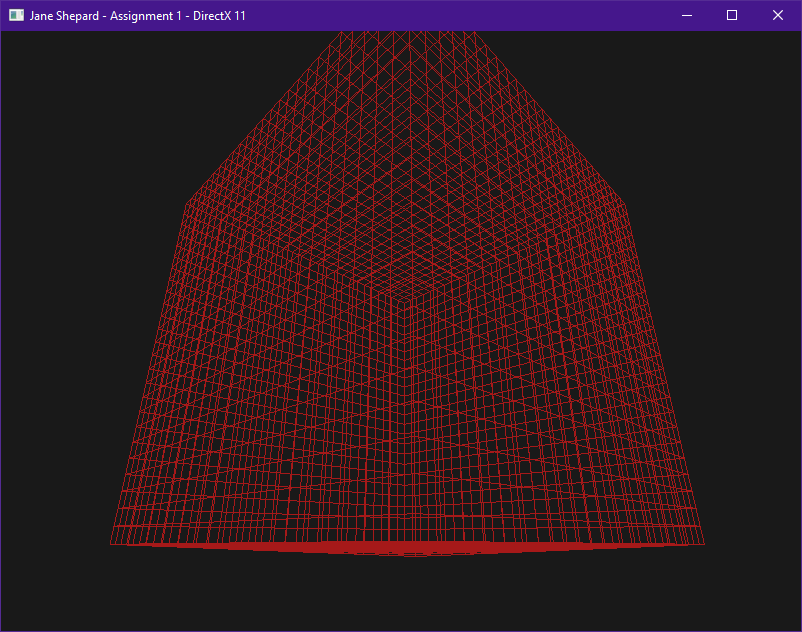
Thumbstick\_Speed = PI \* Seconds\_Passed\_Since\_Last\_Frame

Total\_Pitch = FOV \* MOUSE\_Y\_DELTA / SCREEN\_HEIGHT + RIGHT\_STICK\_Y\_AXIS\_STATE \* Thumbstick\_Speed \* -1

PitchMatrix( Total\_Pitch )

Camera = MatrixMultiplication( PitchMatrix, Camera )

***Note:*** *Like translation, there are several functions in Gateware’s math library to assist with* ***rotation****.*



*You should now be able to pitch your camera up and down.*

### Part 4G

All that’s left is to allow the camera to **turn left and right**. For the Y axis, we want global rotation for an **FPS-style** camera. Local Y rotation is more akin to a flight-sim style of camera.

Let’s add the ability to **yaw the camera left and right**

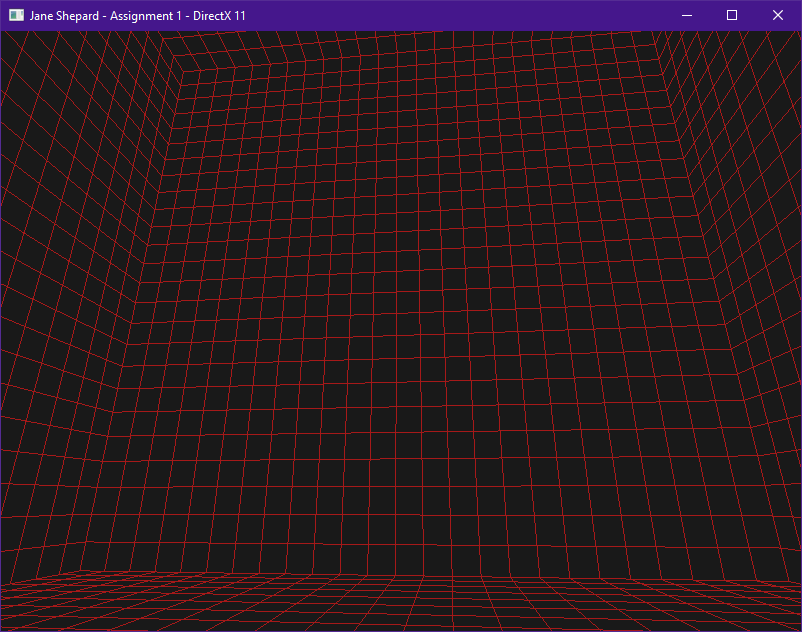
Total\_Yaw = FOV \* AspectRatio \* MOUSE\_X\_DELTA / SCREEN\_WIDTH + RIGHT\_STICK\_X\_AXIS\_STATE \* **Thumb\_Speed**

YawMatrix( Total\_Yaw )

Camera.SavePosition()

Camera = MatrixMultiplication( Camera, YawMatrix )

Camera.RestorePosition()



*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 code it can cleanly encapsulate.*

# 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://learning.oreilly.com/library/view/practical-rendering-and/9781439869765/> (Official D3D11 Coursebook)

<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 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 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) originally had 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.