Assignment 1 OpenGL

# 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 OpenGL API

1. If you are on Windows, your graphics drivers will already support some version of OpenGL.
2. With that said, I recommend you grab a copy of [GLview](https://www.realtech-vr.com/home/glview) to see what features you have available.

Graphical user interface, text, application

Description automatically generated

## 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 with medium confidence

### Part 1b

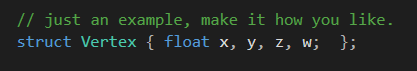
Next, we are going to switch the **MODE** to be able to draw multiple lines. We will attempt to **draw 3 lines around the triangle**. To do this successfully you may need to increase the number of vertices you currently have.

A picture containing 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 **glVertexAttribPointer(…)** routine 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 calling this function again is how we would)

The last part of the rule of three is to adjust the **vertex shader** itself. Modify the existing shader to now take in the full 4 component vector and send it to the rasterizer via the **gl\_Position** global attribute. We will need to make more [GLSL](https://www.youtube.com/watch?v=uOErsQljpHs) code adjustments later to do the necessary world/view/projection 3D transformations.

### 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 **VertexBufferObject** 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 graphical user interface

Description automatically generated

***Note:*** *You’ve just written a block of code that creates a grid. Sounds like a single responsibility to me! Extracting this code out into a well-named helper function will help keep your codebase clean and navigable.*

## 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 six unique World matrices.

***Note:*** *You’ve just written a block of code that initializes our 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

Declare a matrix variable globally in your **GLSL** vertex shader. A single 4x4 matrix can be represented in the GLSL language by the type **mat4**.



Now place the **uniform** keyword in front of it, this tells GLSL that you intend to supply the data from outside the GPU. In the next part, we will supply matrix data from C++ to the shader code so it can be applied here in GLSL before rasterization occurs.

### Part 2c

In **SetUpPipeline()** use the function **glUniformMatrix4fv(…)***.* To send the C++ **world matrix** we setup in PART 2A to the GPU shader **uniform**. Essentially, OpenGL has a variety of “glUniformXYZ” style functions that let you overwrite the variables found in the shaders **between draw calls**. While no longer considered a particularly efficient way to transmit shader variables, there is no arguing how convenient this method is compared to more modern APIs.

Pay special attention to the **first argument** of the above function. To overwrite the correct **uniform,** it is crucial that you **query its location** within the shader using the function **glGetUniformLocation(…)**. Ignoring this step can lead to confusing/incorrect behavior in your shader code.(It is faster to do this only once during initialization)

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

***Note:*** *The block of code you’ve just added to SetUpPipeline() should now be successfully writing to the worldMatrix uniform. 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 2d

Finally, we add the **GLSL** code required to use the matrix we have provided to the Vertex Shader. Unlike HLSL where you need to use the **mul** function to do matrix/vector multiplication, in GLSL the **overloaded multiply operator** works just like you would expect. Try it out by multiplying your local space **vec4** into world space!

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

A picture containing text

Description automatically generated

***Tip:*** *OpenGL/GLSL Samples you see online may appear mathematically backwards(multiplication ordering) to what you are used to. This is likely because they tend to use the* ***GLM*** *math library which is a* ***column major*** *math library.*

### 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**. (OpenGL’s NDC space is surprisingly also Left-Handed)

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.

### Part 2F

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. (You can declare temporary variables in the shader if desired)

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

A picture containing shape

Description automatically generated

**Important:** The conceptual **near and far planes** do not exist yet, so anything outside the **Z range of -1z to +1z** will not be drawn. Because of this we will need to choose camera values between **-0.5f to +0.5f**, otherwise we can expect the clipper to eliminate any geometry that is shifted outside of NDC.

To create this image, I placed my camera at the center of the grid(after it was moved down). I then rotated it **-45** degrees to the left then **-45** down. Finally, I translated the camera backwards **-0.5f** units on the **local Z** axis before taking its **inverse** to get my view matrix.

## 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 **right-handed perspective projection matrix** specifically for the OpenGL 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:**  GOpenGLSurface::GetAspectRatio()

### 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. (More efficient for the shader too!)

You can also just **send the projection matrix** by itself in addition to the World & View since there is still plenty of room in the shader for uniform data. OpenGL guarantees there will be room for at least **1024 floats**(or 128 vec4), with some hardware reserving even more space for direct uniform data.

***Tip:*** If we require even more uniform storage, there are **UBO**s(16KB guaranteed) and **SSBO**s(128MB guaranteed).

Text

Description automatically generated with medium confidence

Uh oh… where did our grid disappear to? Keep in mind we have now switched to using a classic “OpenGL” style Right-Handed projection for movement. This means that while the X & Y axis work like you might expect, the Z axis is now flipped around! This means to travel forward we follow the negative Z axis, and to go backwards we move along the positive Z axis!

If you remember earlier in PART 2F, we translated our camera along the negative Z axis to move away from the grid so we could see it from above. Now that we are using a Right-Handed projection the **reverse** applies. Go back and change the direction of travel and we should see our grid again, but now with full 3D perspective!

Background pattern

Description automatically generated

***Note:*** If you are used to Left-Handed systems this may take a moment to adjust to. Aside from -Z axis travel, also be aware that positive rotations are now counterclockwise along an axis instead of clockwise.

***Tip:*** If you must use Left-Handed coordinates in OpenGL it is possible, check out GMatrix for the LH projection.

### 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 **glUniformMatrix4fv(…)** 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 let’s 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 background pattern

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’s 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 )

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 )

A picture containing text

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 graphical user interface

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! OpenGL isn’t the most efficient or most feature rich graphics API anymore, but as you can see it is still plenty capable of the core 3D work most games need. The flip side of the coin is that the API is quite clean and minimal. If you don’t think so, try Vulkan or D3D12 and get back to me ☺. Now don’t get me wrong those are powerful & useful APIs, but not every project needs access to the latest greatest GPU features or every last drop of performance. If that describes your project, OpenGL could be the way!

The second introductory assignment in this course will have you loading your first 3D Model and applying a basic lighting algorithm 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.

## OPENGL API

<https://www.khronos.org/opengl/wiki/> (Official wiki for OpenGL)

<https://www.khronos.org/registry/OpenGL-Refpages/> (Official Documentation)

<https://docs.gl/> (Unofficial but fantastic, has example code snippets, even has a Dark mode!)

<https://github.com/g-truc/ogl-samples> (Official GitHub API Samples)

<https://learning.oreilly.com/library/view/opengl-superbible-comprehensive/9780134193120/> (THE OpenGL Book)

## GLSL GRAPHICS LIBRARY Shading Language

<https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language> (official wiki for GLSL)

<https://shadered.org> (opensource HLSL & GLSL shader IDE, excellent for writing shaders with immediate feedback)

<https://www.shadertoy.com/> (community for prototyping & sharing GLSL shaders)

## 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 OpenGL API correctly?
  + Aside from reading the docs and making sure the code compiles, we have enabled run-time debug output in the OpenGL API (In Debug mode only). Be sure to pay close attention to the **console window** when running the program. Some messages are non-critical but the ones that are will say **\*\* GL\_ERROR \*\***.
* Visual Studio is treating my GLSL shaders like plain text, 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.
  + OpenGL will attempt to 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.
  + Once your shaders get very complex, I recommend using a dedicated shader IDE like [ShaderEd](https://shadered.org/).
  + If you want your syntax highlighting and intellisense back, there *are* a few community-made Visual Studio extensions for GLSL. These are very useful, but they may not be available for your version of Visual Studio – at time of writing, there are none for VS 2022.
* 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 OpenGL/ES interface to Gateware, setting up a 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.