Assignment 2 OpenGL

# Introduction

In the final guided API assignment in this course, 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 **uniform buffer** and then have it respond to a directional light source with a specular component.

# 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 2

## 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

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)

Finally make your way back to the actual source code and **include your new header file**. (Optional) If you would like the file to be **permanently included as part of the actual solution filter**, you will need to edit the **CMakeLists.txt** file and **rebuild** the project. (Look carefully, you will see the other source files. And yes, it is [possible](https://www.jetbrains.com/help/clion/cmakelists-txt-file.html) to put them all in a single list and use that instead)

### Part 1C

With the model data now available to us, we turn our attention to replacing our existing 2D NDC triangle with the new data. To do this we will need to abide by the **rule of three.** First, find the code where the triangle is currently being copied to a GPU **vertex buffer** and **replace the data** with all the vertex data from the **model header** file.

After doing this you will probably notice that your triangle appears odd. (If it even appears at all!)

Chart, line chart

Description automatically generated

### Part 1D

Let us keep in mind that out new 3D model contains **many triangles**, not just the one. Go to where the triangle is being drawn and adjust it to **draw** the **correct number of vertices** that are listed in the model data.

A picture containing chart

Description automatically generated

As you can see, **the current code is designed to draw 2D NDC triangles.** Though we are drawing the full amount of vertex information, the API does not understand that this data is meant to be used as a **full 3D model that includes an XYZ position, UVs and even normal data.**

### Part 1E

If you look inside the model header, you will notice all the vertex data comes in the form of an **OBJ\_VERT** structure. This is the **first part of the rule of three**, and we will need to match the rest of rules if we are to be successful.

Next, we will need to adjust the **glVertexAttribPointer(…)** functions to correctly match the binary format of the **OBJ\_VERT** structure so the data is read in correctly. This will be the second part of the rule of three. You will need to exactly match the **order**, **byte size**, **format and byte offset** of each vertex attribute in **OBJ\_VERT**.

Take note that this routine must be called once **for each** individual **attribute** in the vertex structure. For example: If a vertex structure had both a position and a color, we would need to call glVertexAttribPointer(…) two times. Additionally, each unique attribute must be individually **enabled** using **glEnableVertexAttribArray(…)**.

Diagram

Description automatically generated

Its starting to look like something… but still very much off from what we are expecting!

***Note:*** *Keep your eyes glued to the console while working with OpenGL. If the API detects it is being used in an improper or unexpected way, it will print error messages there. Resolve any errors before continuing!*

### Part 1F

For the **last part of the rule of three** we need to adjust our vertex shader. Keep in mind the shader was originally written to draw a 2D NDC triangle. We will at least modify it enough so that the **correct data is coming in** and the **full 3D position is going out**.

Adjust the incoming vertex so that **all three components(position, UV & Normal)** are now received by the shader. Study the syntax of the **GLSL shader language** to make these changes. Tweak the **output data** so that it uses the full **XYZ** component of the position, setting the **W** to its standard homogenous starting value.

Diagram

Description automatically generated

Hmm… no visual improvement, still feels like we are missing something… nonetheless it is critical we do this step!

***Note:*** *It is extremely likely that you will encounter GLSL syntax errors in the console while doing this step. Use the output of glGetShaderInfoLog to figure out what you are getting wrong with the syntax!*

### Part 1G

Even though we seem to have the **rule of three** correct now, it turns out we are still missing a **major** piece of the puzzle! Take a moment to go into the model header file and scroll past the **vertex array**.

Text

Description automatically generatedUh oh… Looks like we missed an entire section of data! **Index data** is a critical part of almost all 3D model files. This data is used during the **vertex assembly** process by the GPU to efficiently **reuse** existing vertices shared by multiple primitives (lines & triangles).

While it is possible to make and draw a shape without index data, GPUs are optimized to render with them and pretty much all 3D model files require it to be used one way or another. Thankfully, all modern graphics APIs can accept **index buffers** and draw using them.

To make an **index buffer** in OpenGL, add one new **GLuint** to the class to hold our index buffer object. Then call **glGenBuffers(…)** again followed by **glBindBuffer(…)** and **glBufferData(…)**. However, this time use **GL\_ELEMENT\_ARRAY\_BUFFER** instead of just GL\_ARRAY\_BUFFER and of course pass the index data from the model instead of the vertex data. Don’t forget to add the appropriate **glDeleteBuffers (…)** call in the destructor for our new buffer. (Otherwise, we will leak memory)

***Tip:*** *Beware of copy-paste errors here when referencing the vertex buffer versions.*

***Note:*** *You just wrote a block of code that creates an index buffer. Sounds like a single responsibility to me! To keep your initialization code clean I recommend extracting it out into a well-named helper function.   
I won’t bother you with more reminders since you had plenty of them in assignment 1. In any case, I recommend you continue to practice and grow your clean coding skills throughout this assignment. You will not be docked points for messy code, but clean code* ***is*** *easier to expand and debug. You will likely have an easier time with this assignment and future programming if you code cleanly.*

### Part 1H

With our **index buffer** now allocated and populated, we can use it to draw the model as intended. First you will need to **glBindBuffer(…)** our newly created index buffer (be sure to use the correct flags). Then you will need to switch to using a different **draw operation** that supports **indexed/element**-based geometry submission. If you get it right, it should look something like this:

Logo

Description automatically generated

***Hint:*** *No need to provide an index pointer to the* ***Draw*** *call as glBindBuffer(…) is what links the index buffer object.*

If you did this correctly, the 3D model will **appear offset**! This seems counter intuitive but remember that without a **View & Projection Matrix** the camera is technically located at the **origin with no perspective**. Because of this, the triangles are still being interpreted as **NDC space** coordinates. Since the model was created to be sitting above the origin, this is technically an accurate visualization. (at least so far)

Graphical user interface

Description automatically generated

**(Optional Step)** GPU/API Debuggers like [**RenderDoc**](https://renderdoc.org/) are crucial for programmers writing graphics code. In the above screenshot I have used the tool to **inspect** the indexed draw call we just wrote. Even if the running program showed **nothing** but a blank screen, I could still tell the geometry is **loaded correctly** because **RenderDoc** has a **visual inspector** that lets me look at any **geometry** we have already copied onto the card.

***Tip:*** *To enable RenderDoc capture and debug of the program, you must direct the program to your executable file.*

Now that we know the geometry is there, we can move on to making our vertex shader 3D just like we did in the previous assignment. However, before we do that it would be nice to fully see our model on-screen. To do so, we can **temporarily** adjust the **vertex shader** so it **shifts all the Y coordinates down by -0.75f.**

Logo

Description automatically generated

***Note:*** *I chose the(-****0.75f****) number 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 in the next part.*

## Part 2 | 50%

### Part 2a

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:*** *Most OpenGL examples prefer a right-handed coordinate system; this lab will make that same assumption. If desired Gateware includes a projection matrix for OpenGL that can allow you to the API left-handed instead.*

### Part 2B

OpenGL has **three primary ways** to communicate **variable data** to running shaders: **Uniform Update Functions**, **Uniform Buffers,** and **Storage Buffers**. **glUniformX(…)** is a convenient way to move CPU data to the shaders right before a draw call. Unfortunately, it is a less than ideal way to move large amounts of data to your shaders cleanly.

In addition to matrix variables, we are going to want to upload the information about **materials & lights** to both the vertex shader and pixel/fragment shader so we can correctly visualize this model as intended. For this reason, we are going to use a **Uniform Buffer Object** (A.K.A a UBO). Though not quite as easy to use as **glUniformX(…)**, they can transfer a more useful amount of memory in one go. The first section of this buffer will hold the data for our overall scene, the second area will hold information about each sub-mesh in the model.

A **Storage Buffers** **Object** (A.K.A an SBO) allows you to store significantly more than a UBO. If you potentially have megabytes or even gigabytes of non-vertex/non-texture data, then a Storage Buffer may be the correct choice. The trade-off is potentially slower memory access; API complexity is much like Uniform Buffers.

Before we dive into their creation lets **organize the data** we intend to send to our shaders. The following structure should cover most of our needs when rendering the Logo:

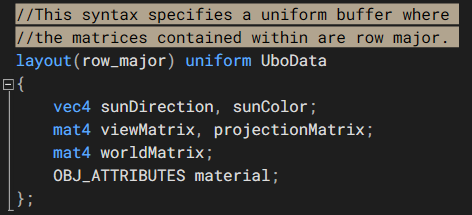
Text

Description automatically generated

***Note:*** *It would be more optimal to split the above data into separate buffers, we chose to bundle them here for convenience and simplicity. Optimized OpenGL code should approach this differently. (Ex: Instanced rendering)*

***Tip:*** *You will notice above shader data is also 16byte**aligned. Structures on the CPU may not line up with the equivalent version on the GPU if you are not careful about aligning data to 16byte register boundaries.*

You will also need to **mirror this data in both GLSL shaders**. Be sure to match the order and size of the variables using the language’s built-in types. Since there is no **OBJ\_ATTRIBUTES** type in GLSL you will need to **make your own**, again mirroring the **size and order** of the data in the C++ struct.



***Note:*** *The use of row\_major in the layout of this uniform buffer tells OpenGL to read in the data from C/C++ and transpose it during shader upload. It is important to be aware it is still considered column\_major inside the shader.*

Complete this step by making a class **instance** of your new C++ structure and **initialize** everything inside to all the math variables we made at the start of this process. The model we are attempting to draw contains more than one sub-mesh, for now we will just focus on drawing the first part. Transfer the first **material** from the model’s **header file** to the material of the UBO\_DATA. We will end up using the same world matrix for both sub-meshes for now.

***Note:*** *Though you could represent them with a* ***64bit double*** *type, I chose to ignore the material string pointers in GLSL since they are only theoretically used during initialization. Therefore, structs only contain* ***OBJ\_ATTRIBUTES.***

### Part 2C

Now that we have the data required to draw our 3D model created, we will need to get it onto the GPU. As we discussed earlier, we will do this using a UBO (Uniform Buffer Object).

Creating one is very similar to how we created our Index Buffer. Start by creating another **GLuint** to represent the new buffer inside the class. Then use the same functions you used to create and fill the Index Buffer for making the UBO, but this time pay attention that you select the appropriate **target** parameter (it should be self-evident).

The initial data we are going to be copying over will come from the instance of the UBO\_DATA you filled out in the last section. When copying the data be sure to use the **GL\_DYNAMIC\_DRAW** flag since this data is meant to be changed often from the CPU side.

***Tip:*** *Don’t forget to free the memory of this new buffer during shutdown!*

### Part 2D

Alright… so UBOs are great but they are a *comparatively* recent addition to the OpenGL API. Using modern OpenGL often necessitates querying for new functionality that may not be part of the base API. If you look inside the OpenGLExtensions header, you will see many handles to functions that came from later editions of the API.

Unfortunately, there are currently no functions related to the use of UBOs available in the template. This means we will need to directly query them from the OpenGL driver.

Here is a list of the functions that need to be added:

* **glGetUniformBlockIndex**
* **glBindBufferBase**
* **glUniformBlockBinding**
* **glBufferSubData**

Be sure to follow the spelling **exactly** and follow the code pattern of the other functions queried in this way. Once you have declared the function handles in the header, you will need to also add them to the **QueryOGLExtensionFunctions()** function located directly below. This is what links the optional API function pointers directly to OpenGL’s runtime.

### Part 2E

With the UBO created and our new functions linked, lets connect our uniform buffer to the OpenGL pipeline!

Go to the Render function and somewhere before calling OpenGL’s draw routine lets grab a handle to the **uniform block** located inside the shader. You can do this using the function **glGetUniformBlockIndex(…)**, you will need to pass it the active shader program and the **name** of the **uniform** block located inside the shader.

We will use the **uniform block index** returned from this function later to connect our UBO to the shader code.

### Part 2F

Next, we are going to take our UBO and **bind** it to a specific UBO slot on the graphics card. You can do this using the function **glBindBufferBase(…)**. Once again select the appropriate **target** for what we are connecting and provide the handle to our buffer. Video cards can support multiple UBOs simultaneously, but since we have only one let’s just use the first slot for now.

### Part 2G

To wrap up this section connecting the buffer to the pipeline we will call **glUniformBlockBinding(…)**. As the name implies, you can use this function to connect the **buffer** we just **bound** to the pipeline to the **uniform block index** we found a little earlier.

If you do this successfully, then our **UBO** should be **bound** to the **uniform** object in the **shader** containing the same format/structure of data.

### Part 2H

If you run your program, you may notice it looks… the same. Keep in mind while we may have hopefully attached our **uniform** data to the shader, that does not mean OpenGL will somehow automagically use it for us.

In theory you now have access to a **world**, **view** & **projection** matrix. Use them in the **vertex shader** to transform the incoming vertex into **projection space** just like you did in the first assignment:

A picture containing logo

Description automatically generated

***Tip:*** *Don’t forget to remove that temporary vertex adjustment we did earlier, we shouldn’t need it after this.*

Excellent, our shader code now has a way to access a mesh’s transform and material data! 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%

### Part 3A

First thing we should do is get rid of that hardcoded color and **replace it with a color from our actual materials**. To do so, copy the **GLSL** code related to your **UBO** and make sure it also available to the **fragment/pixel shader**.

**Replace the outgoing color** with the **diffuse color** from the **material**. It should just be a solid white color.

Graphical user interface, text

Description automatically generated

This appears to be the **correct color of the Text** of the Full Sail Logo. Unfortunately, the **Logo itself** should be closer to an **orange** color.

### Part 3B

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 switch the material used 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.

***Hint:*** *The last argument to glDrawElements(…) can act as a byte offset to where your indices should start drawing.*

### Part 3C

Ok… so now we have separate draw calls for each mesh. Now we need a way to not hard-code the specific material that is supposed to be being used for a given mesh.

A little earlier you added the capability to use a new function: **glBufferSubData(…)**. You may have been wondering what it’s for since we have not used it yet. Well, it’s about to come in quite handy!

This function **locks down** and **writes to** a GPU **resource** like a UBO **safely**. Essentially: glBufferSubData waits for the GPU to stop accessing a buffer, locks it down, writes to it, then releases it so it can once again be used by the GPU.

Use this function to overwrite the memory of the currently bound UBO with an updated instance of your **UBO\_DATA** variable. However, this time be sure to update the material member of the structure to be equal to the current material of the mesh in the drawing loop. If you do this correctly, it should look like this:

Logo

Description automatically generated

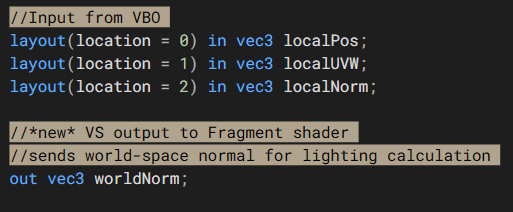
***Note:*** *glBufferSubData can be used to update a portion of a buffer, or the whole thing. We’re overwriting the whole UBO here for simplicity’s sake, but you could use the offset and size to only overwrite the parts that need to be updated if you wanted.*

## Part 4 | 100%

### Part 4a

After you get to grips with how to upload and access static data with a graphics API; you then get to the fun part, **playing with shader code!** 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.

We will start by applying a basic **directional light source** the surface of our model. To make this possible we will need to **output a world space normal** from our vertex shader for use in the **fragment**/**pixel shader**.



In the **vertex shader** make an **out** variable that will receive the transformed normal. The **out** GLSL keyword specifies that this variable will move into the next stage of the rendering pipeline.

***Note:*** *I did not include a* ***UV coordinate*** *in the output 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****.*

***Note:*** *We’re using three separate inputs to take in each attribute of our vertices separately. If you wanted, you could create a struct in GLSL that contains all of the attributes and use that instead. Just make sure the ordering and alignment of your components matches!*

### Part 4B

Now **adjust the vertex shader** so it will initialize the variable you defined in the last step. Do not forget to **transform the outgoing normal into world space** since our lights are also defined there.

You will also now **adjust the fragment/pixel shader**, so it now has an **in** variable to match the vertex shader’s new **out** variable. After you do this, everything should **still compile and draw** like it did before. (We will use the new data in the following steps)

***Hint:*** *Transforming a direction vector vs. a position vector by a 4x4 matrix is slightly different… do you recall how?*

### Part 4C

Now we should have everything we need to apply a **directional light formula** to each of our pixels. Assuming you did not memorize this formula, it was 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)

Text

Description automatically generated

***Tip:*** *Normal data coming from the* ***rasterizer*** *has been* ***interpolated*** *across a primitive. This means it* ***may no longer be normalized*** *when it reaches a particular pixel. Thankfully, there are* ***GLSL***[*functions*](https://docs.gl/sl4/acos) *you can use to renormalize it.*

***Hint:*** *If the lighting is not working correctly, it could be due to not enabling or correctly offsetting the additional vertex attributes. Double check the arguments of the functions you called in* [*Part 1E*](#_Part_1E)*.*

### OPTIONAL

Although [RenderDoc](https://renderdoc.org/) is a great tool, it unfortunately currently **lacks GLSL debugging** support. This means the ability to debug your shaders will either need to be done the classic way (display variables as colors). Or you will need to download one of a handful of other programs that can do it: <https://www.khronos.org/opengl/wiki/Debugging_Tools>

The ones available to you will depend on your current platform and what graphics card you are running. For example, if you are on Windows and running an Nvidia graphics [Nsight VS Edition](https://developer.nvidia.com/nsight-visual-studio-edition) may be an option for you.

Some of these tools (like the one above) may be a bit painful to get installed/working, but if they allow you to step through your shader code it will be worth the effort!

### Part 4D

In Part **2A** you created a **Y rotation matrix** that slowly rotated **over time**. It is now time to put that matrix to use! On the CPU side **set** the **second mesh’s world matrix** in the **UBO\_DATA** to be **equal** to this constantly **rotating matrix**. (This matrix will be used for the top mesh of the Logo)

After you do this, you should see the top part of **Logo spinning**. The lighting should change based on the rotation of the logo. If it does not, make sure you are correctly transforming the **normal** attributes into **world space**.

Text

Description automatically generated

***Tip:*** *Ideally updating GPU shader variables is an operation that happens once for each frame. For simplicity we update data between draw calls, but it would be better if we had arrays of matrices/materials in the shader.*

### Part 4E

The final part of this assignment will just have us cleaning up and enhancing the lighting effects. We will start by finding the section in the code where we our **UBO structure** is and **adding two new vectors**.

We will 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 fragment shader** to compute the **total** amount of light striking a pixel before multiplying it by the **surface color**. If you do this correctly it should look like the below picture. (*If you need a refresher on the ambient term, again check CGS day 7*)

Text

Description automatically generated

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

### Part 4F

The last step in our journey is to 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**.

Text

Description automatically generated

***Hint:*** *You will need to adjust the vertex shader to also output the world position of the vertex for specular to work.*

(Optional) instead of using the classic **half-vector method** provided in the slides, you can instead compute the exact vector reflected from the surface and compare that to your view vector. This will get you a much cleaner and more accurate specular reflection as shown below:

Text

Description automatically generated

***Tip:*** *GLSL has a* ***reflect*** *function built directly into the language. It has many useful applications in graphics.*

# Summary

Great Job! OpenGL may not be the most modern API, but as you can see it is still plenty capable of the rendering tasks required by most games. From here its time to start thinking about which of the 3D hardware APIs you want to use for your Level Renderer. They all have advantages and disadvantages. APIs like Vulkan and D3D12 are more complex but also more efficient and more in-demand on a resume. APIs like D3D11 and OpenGL are older/simpler but still commonly used, and often are plenty good enough if you don’t need bleeding edge performance and features. (Ex: 2D/Mobile/Cell Shaded games)

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.

## 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)

## 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)

*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 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 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 stuck on Part 2 and do not understand how uniform buffers work. Any advice?
  + Uniform Buffers can be a little confusing at first, but there some great resources out there explaining how they work. Try to figure things out using the docs, but if you are stuck reading [this article](https://www.geeks3d.com/20140704/gpu-buffers-introduction-to-opengl-3-1-uniform-buffers-objects/) could really help!
* 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.