Lab 3 Vulkan

# Introduction

In this Lab we will build a 3D grid procedurally and then use the core mathematics of computer graphics (world/view/projection) to manipulate the shape in three dimensions. These matrices will be **uploaded** to the Shaders using something called a **Uniform Buffer**. This will allow for significantly more storage room than the **Push Constants** from Lab 2. To utilize this new buffer, we will also need to go over Vulkan **Descriptor Sets**. Finally, we will integrate a 3D Camera to help us navigate the scene.

# Getting Started

The method of getting started with this and most future labs should be identical to the first lab assignment. The main difference is that you now have Vulkan installed, so reinstalling it should not be required.

However, all the other steps still matter. In particular: Cloning this repository so your progress can be saved. If you don’t remember all the steps, please review the getting started section from Lab 1.

# Assignment 1

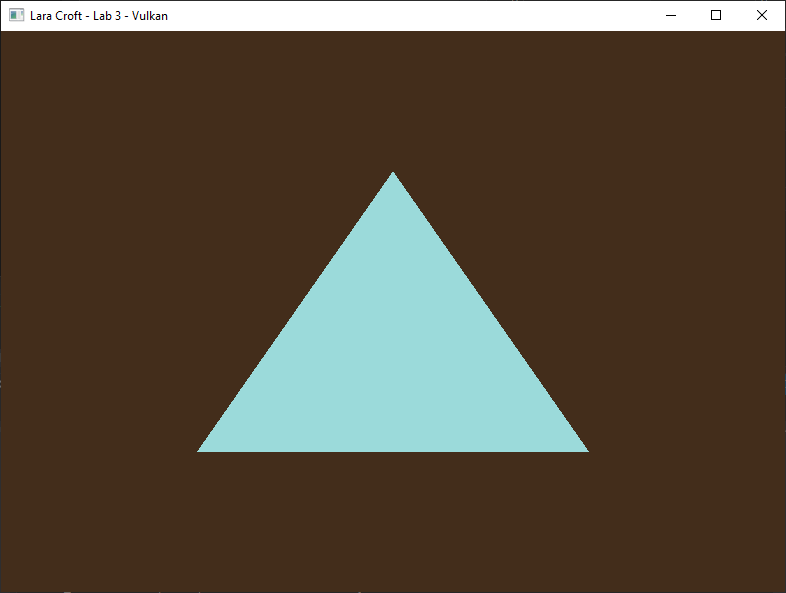
## Part 1 | 25%

### Part 1a

Choose some colors you like(optional). Study the code and familiarize yourself with where things are.

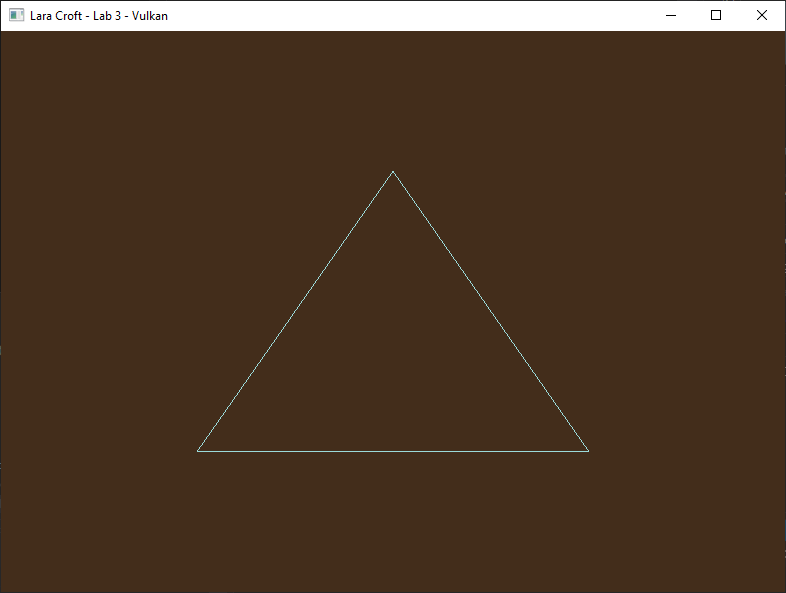
Use the “SetWindowName” function from GWindow to place your name and API variant at the top.

Lab 3



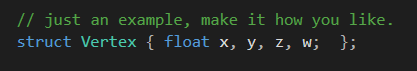
### Part 1b

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



### 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.



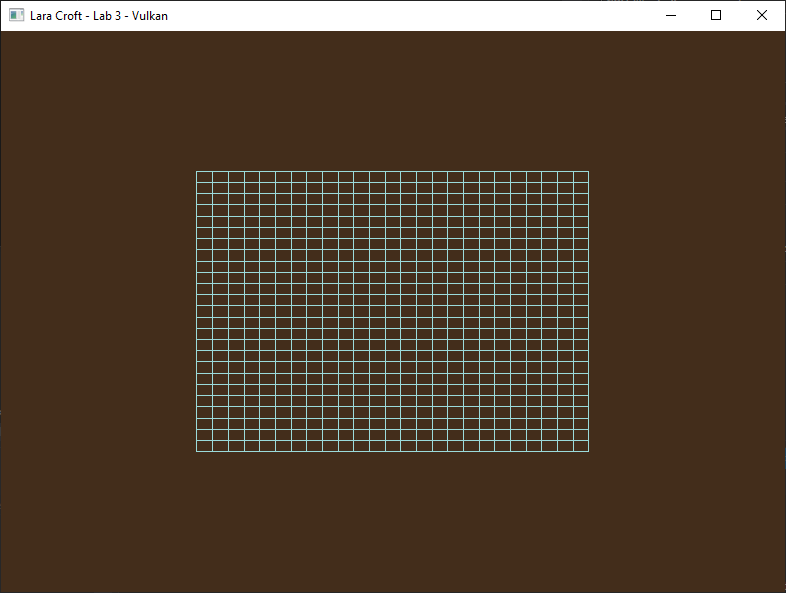
For this to work properly you will need to apply the **rule of three** to the code. First convert the existing triangle to the structure, adding zero and one for the Z and W values respectively.

Next find the **VkVertexInputBindingDescription** and adjust it so that it will accept the additional data you are now passing to the video card. You will also need to adjust the **VkVertexInputAttributeDescription** so that the format in use is again compatible with the extra data being passed per-vertex.

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

### Part 1D

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



***Note:*** *You’ve just written a block of code that creates a grid of vertices. That sounds like a single responsibility to me! To keep your initialization code clean, I recommend extracting it out to a well-named helper function.*

## LAB CHECKPOINT | MANUAL COMMIT | DO NOT SKIP

To receive credit for your assignments, you must manually commit to your Repo 4 times (Once at the end of each Part/Section). If you skip this commit (or disable auto commits) you will not receive credit for the previous section.

1. Change the lab’s Title Bar so it says where you are in the lab.
   1. For Example: “John Smith – Lab 1 – Part 1 Complete”
2. Take a Screenshot of the lab window running showing your work. (Windows Key + Shift + S)
3. On Windows the screenshot is in your clipboard, open MS Paint and (Ctrl + V) to load it in.
4. Save the image with the same name as your title bar in the root folder of this repository.
5. Open GitHub Desktop for this repo. You should see the new image as a pending change.
6. Create a commit message with the same name as the title bar & commit your changes.

If this section is not perfect but you need to continue forward, you must still do this to get partial credit!

## Part 2 | 50%

#### OVERVIEW

Vulkan has **three primary ways** to communicate **variable data** to running shaders: **Push Constants**, **Uniform Buffers** and **Storage Buffers**. Push Constants are an extremely convenient and easy way to move CPU data to the shaders in a draw call. Unfortunately, they have serious limitations. Graphics cards are only required to support **128bytes** of data via this method. Essentially this is enough room for two 4x4 float matrices and nothing else.

Last time, before using the GPU-matrix we just declared in the shader, we uploaded our CPU World matrix data to the shader’s GPU memory block via **Push Constants**. Unfortunately, this time around we are going to need a lot more space than the 128 bytes provided by Push Constants.

To circumvent this limitation, we will use something called a **Uniform Buffer**. This involves allocating a separate buffer for **uniform**(shader variable)**data** and a **VkDescriptorSet** required to reference that memory in Vulkan.

**Uniform buffers** are a compromise between Push Constants and Storage Buffers. They are very efficient like Push Constants but also hold much more data. (Up to 16KB guaranteed, often up to 64KB) Still 16KB is nothing compared to the **gigabytes** a storage buffer can hold, and they lack the simplicity and convenience of Push Constants.

### Part 2a

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

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

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

After initialization use the matrix operations to create a matrix that rotates exactly **90 degrees around the** **X axis and translates down the Y axis 0.5f units**. Assign the combined matrix to a new class variable, this matrix will be the first of four unique World matrices.

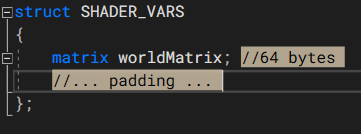
***Note:*** *You’ve just written a block of code that initializes a world matrix. That sounds like a single responsibility to me! To keep your initialization code clean, I recommend extracting it out to a well-named helper function.*

### Part 2B

Make a struct called SHADER\_VARS (or similar). And place one GMATRIXF inside it to represent your world matrix.

The minimum size of the shader data you can send to Vulkan is **128 bytes**. This means you will need to add padding to the structure, so it is at least 128 bytes in size. Once that is done transfer the initialized World matrix from the last part to an instance of this structure in the **Render()** function so you can eventually send it the vertex shader.

Declare an almost identical version of this structure in your **HLSL** vertex shader. A single 4x4 matrix can be represented in the language by “float4x4” or just “matrix”. You may need to adjust the type of value used for padding so that the **shaderc** compiler is happy. Just be aware the different types are different byte sizes so adjust the amount accordingly.



Now switch the keyword **struct** with **cbuffer**(constant buffer), this tells HLSL that you intend to supply the data from outside the GPU. In the last assignment, we used the HLSL attribute **[[vk::push\_constant]]** to tell Vulkan this is where our “*Push Constants*” data would go. However, this time DO NOT add this above the cbuffer.

### Part 2c

Now that we have the data required to draw our 3D model created, we will need to get it onto the GPU. We can do this by creating a **uniform buffer** which is very similar to the vertex and index buffers you already created.

There is one catch however… because this buffer will be **updated each frame**, we must take care to **avoid synchronization issues** between the GPU and CPU. Unlike older 3D APIs like D3D11 and OpenGL, in Vulkan this **synchronization is not handled for you** automatically.

That is the bad news, the good news is that the Gateware template is already **synchronizing on a per-frame basis**. What this means is that if we **create a storage buffer for each simultaneous frame** that can be processed, we will **not run into any sync issues** as each in-flight frame will have its **own uniform buffer** it can use directly.

Using the **same types you used for vertex and index buffers**, go ahead and allocate a **std::vector** of each for your **uniform buffers**. In the next part we will **resize()** them based on the **maximum number of frames** reported by **GVulkanSurface**.

### Part 2D

Once you determine the **maximum number of active frames\***, use a **for loop to initialize all** the **uniform buffers** the exact same way. Again, you can pretty much copy how this is done for the vertex and index buffers, however we will change the data going in to be a copy of the struct you initialized from the end of step **2B**. (Do not forget to also adjust the buffer size and USAGE\_BIT to match)

Even though we made multiple buffers, in the next section we will focus on just linking the first one to the pipeline.

To finish this step, we will also be sure to **free** these new **uniform buffers** on program completion.

***\*Tip:*** *Vulkan refers to frames displayed to the end user as swap chain images.*

### Part 2E

Buckle-up, this is where Vulkan starts to get a bit more gnarly than the simpler APIs. Next thing we need to do is describe to the existing **VkPipelineLayoutCreateInfo** that it will be using a descriptor set to supply external data to the shaders(in this case our storage buffer).

The interface that does this is called a **VkDescriptorSetLayout**. To make one you need two things, a **VkDescriptorSetLayoutCreateInfo** which itself needs a **VkDescriptorSetLayoutBinding**.

The VkDescriptorSetLayoutBinding should only have 1 descriptor as that is all we need for now. The **type** of descriptor should be used for **uniform** buffers. The **stage** it is assigned to should be the **Vertex & Fragment**(Pixel) shaders, as that is who needs this data. Don’t forget to fill out the other values even if we are not using them.

Next, we make the VkDescriptorSetLayoutCreateInfo which tells Vulkan how many bindings we have and where they are. The rest of the arguments can be set to nullptr or whatever their required defaults are. (Read the docs)

Ok… now we can finally call **vkCreateDescriptorSetLayout**. Add a permanent handle to a VkDescriptorSetLayout in your class, we will need it so we can **free its memory** at the end of the program. Speaking of, go ahead and take care of that now once you have created it.

The final step in this section is to tell the existing **VkPipelineLayoutCreateInfo** that you have a usable descriptor set layout now.

### Part 2F

Well, all of that was just to tell the pipeline “Hey! Descriptors are coming!”. Now we need to supply said external descriptors. These external descriptors are called **VkDescriptorSet**(s), but before we can make one, we need something called a **VkDescriptorPool**.

“Pools” are how Vulkan efficiently reserves memory on the video card, there are many different kinds. A VkDescriptorPool is used to reserve descriptor sets, **add one to your class** and add code to **free it** during clean-up. You can then create one using **vkCreateDescriptorPool**.

To do this you will need to supply a **VkDescriptorPoolCreateInfo**(starting to see a pattern here?). We need a very shallow pool, so it should only take 1 descriptor and no special flags other than the required defaults. Remember, we are trying to link our new **uniform buffer\*** to the shaders. Set the **VkDescriptorPoolSize** it wants appropriately.

\****Note:*** *Again, just focus on using one buffer for now. The others will be used later to solve any sync issues.*

### Part 2G

Are we there yet? I know, I know… this is getting a bit absurd, but the good news is once you do this once; you pretty much have a blueprint on how to upload most any non-geometry resources to Vulkan.

Anyway, our new **VkDescriptorPool** can actually allocate some of those **VkDescriptorSet**(s) we have been wanting for a while now. To do this we can use the function **vkAllocateDescriptorSets**. Before we do so, lets add an actual VkDescriptorSet to the class. You **do not have to free** the memory for the descriptor set as it is part of the descriptor pool, though it is possible to do so if you need to.

Like most things in Vulkan, you must describe the thing that you wish to create/allocate in the API. In this case you do so using a **VkDescriptorSetAllocateInfo** structure. Thankfully, the arguments to this structure are self-explanatory at this point, use the docs to fill them out.

### Part 2H

So, the good news is we have everything we need allocated now. The bad news is none of them know about each other. To correct this issue, we will start by linking our new **VkDescriptorSet** to our **uniform buffer**.

To do this you will need a **VkWriteDescriptorSet** and a **VkDescriptorBufferInfo** structure to describe what you are trying to do. Filling their members out is obvious for the most part. Keep in mind, we are connecting one **uniform buffer** and we do want access to **all** of it.

Once you have filled everything out use **vkUpdateDescriptorSets** to tell the **VkDevice** to link them together.

*Tip: Consider splitting up the code you have just written for these last few parts into their own functions.*

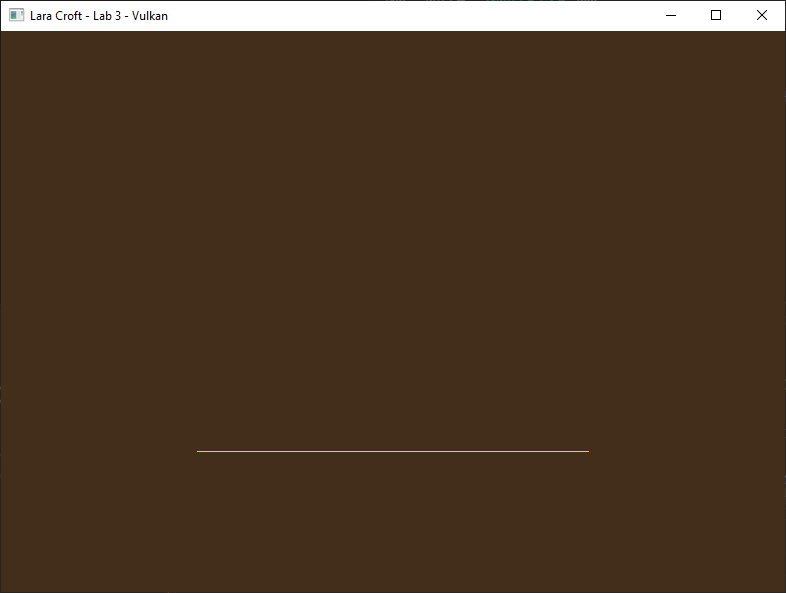
### Part 2I

Yes… this is the last part; and yes, we will finally see a visual change after this! There are only two things left: **Connecting** the descriptor(s) to the command buffer and finally **using** the model’s matrix data in the vertex shader.

First, inside the **Render()** function we will **connect** the descriptor set to the command buffer using **vkCmdBindDescriptorSets**.It will ask for many of the items we created over the last few sections.

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

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



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

## LAB CHECKPOINT | MANUAL COMMIT | DO NOT SKIP

To receive credit for your assignments, you must manually commit to your Repo 4 times (Once at the end of each Part/Section). If you skip this commit (or disable auto commits) you will not receive credit for the previous section.

1. Change the lab’s Title Bar so it says where you are in the lab.
   1. For Example: “John Smith – Lab 1 – Part 1 Complete”
2. Take a Screenshot of the lab window running showing your work. (Windows Key + Shift + S)
3. On Windows the screenshot is in your clipboard, open MS Paint and (Ctrl + V) to load it in.
4. Save the image with the same name as your title bar in the root folder of this repository.
5. Open GitHub Desktop for this repo. You should see the new image as a pending change.
6. Create a commit message with the same name as the title bar & commit your changes.

If this section is not perfect but you need to continue forward, you must still do this to get partial credit!

## Part 3 | 75%

### Part 3A

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

Use the math library from earlier to create a **View Matrix** so we can see the scene from above(**+Y**), back(**-Z**) and to the right(**+X**). You can do this the same way you did in **CGS day 4** by placing a world space matrix where you want the camera to be and then taking its inverse. (*Tip: there is a function in the math library designed to make this process even easier, see if you can spot it*)

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

***Note:*** *You’ve just written a block of code that initializes a view matrix. 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.*

### Part 3B

Our new matrix is no good to us if we can’t actually use it, thankfully we have plenty of room now in our **SHADER\_VARS** structure (**uniform buffer**) to upload it. (Adjust it appropriately)

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

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

A screenshot of a computer

Description automatically generated

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

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

### Part 3C

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

Let’s start by using the math library to create a **left-handed perspective projection matrix** specifically for the Vulkan 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

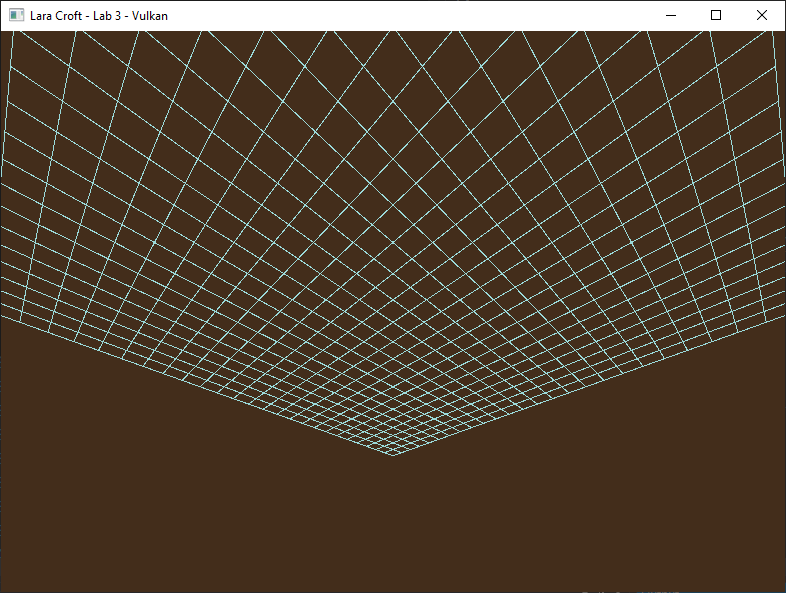
**Aspect Ratio:**  GVulkanSurface::GetAspectRatio()

*Note: The projection matrix used in CGS would not work directly here because Vulkan has a slightly different NDC.*

### Part 3D

We now have a third matrix we wish to apply to our vertices. If we were still using **push constants** (Lab 2) there would be a bit of a problem… The push constants data in Vulkan while quite convenient, is only guaranteed up to **128 bytes**. Since a **GMATRIXF is 64 bytes** and we already have **TWO** of them well… you can see the issue.

The good news is we have a **Uniform Buffer** now. Therefore, we have at least **16kb** guaranteed by the [Vulkan Spec](https://registry.khronos.org/vulkan/specs/1.3/html/chap33.html#limits-minmax) (Up to 64kb). So just add the projection matrix to **SHADER\_VARS** struct and apply it in the vertex shader.



*Well, we appear to have some perspective now but based on the numbers we used earlier something seems… off.*

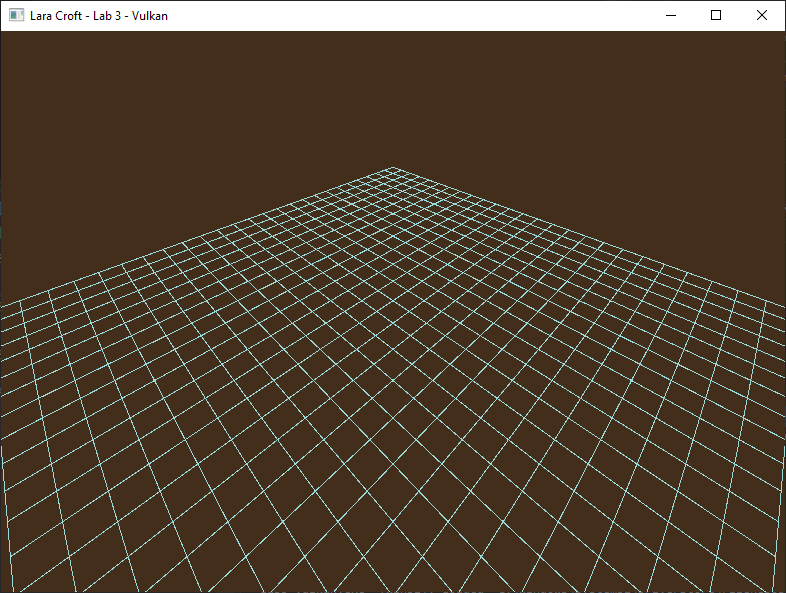
### Part 3E

If you have been paying close attention, you may be wondering why we seem to be **underneath** the grid looking up. You’re not wrong to think this is odd, based on the numbers we were using earlier in a **left-handed** coordinate system **we should be above the grid looking down!**

The reason this happens is because **Vulkan’s NDC has Y+ going down.** The Vulkan projection matrix you built in [Part 3C](#_Part_3C) accounts for this by **negating the Y axis during projection**. So… if that is true why is it still flipped?

If you look at where the **HLSL** shaders are being compiled, you will notice I have a feature turned on during compilation that **automatically has the shaders invert the Y** for us. This was handy to have on initially because it made Vulkan’s NDC space the exact same as the one you first encountered in CGS.

Unfortunately, this feature is now interfering with our projection matrix which was designed to work with Vulkan’s native NDC coordinate system. **Disable it** and things should look more how we had originally expected.



***Tip:*** *You could also leave this compiler feature on and switch to using a “DirectX” style projection matrix instead. This would be the recommended approach if you want math and shaders that work seamlessly across both APIs.*

### Part 3F

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 set up 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.

Adjust your **SHADER\_VARS** structure & **uniform buffer** so it has an array of all 6 matrices. To get your shader to still compile/work, just have it use the [0] matrix for now.

A black background with white text

Description automatically generatedA screen shot of a computer code

Description automatically generated

### Part 3G

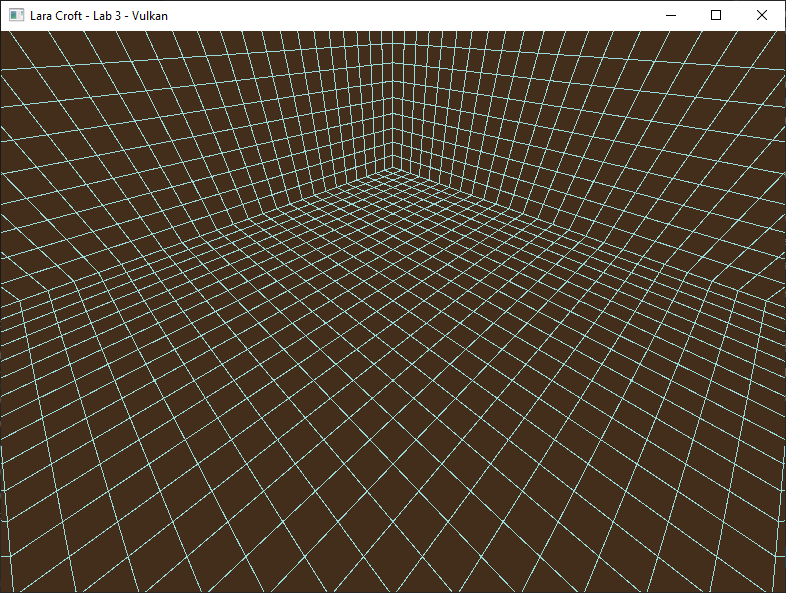
Modern graphics APIs like Vulkan and D3D12 are designed to allow you to render the same geometry data multiple times extremely efficiently. This technique is called **Instancing**, and essentially lets you draw copies of the same mesh simultaneously by utilizing the GPU’s extra bandwidth.

We are going to leverage this feature to have Vulkan draw all 6 walls at the same time. Start by adjusting the **vkCmdDraw** arguments to draw 6 instances instead of just one. When you do this, you will likely notice no change. This is because they are all using matrix [0] and drawing in the exact same spot.

To solve this, adjust the vertex shader to correctly pair the right vertex with the right matrix based on the **SV\_InstanceID** value provided by the graphics pipeline. Essentially, this value is the **index** to which instance the current vertex belongs to. Access it by adding the following input to your Vertex shader:

A black screen with white text

Description automatically generated



You do not need to add the unsigned int to your Vertex structure or adjust the Rule of Three. **SV\_InstanceID** is a **System Value** semantic and is provided by the hardware itself during the drawing process.

If you incorrectly set up your other wall matrices you will probably notice that now. Try hardcoding the matrix index in the vertex shader to find out which matrices work, and which ones don’t for debugging purposes.

***Note:*** *Instancing is the main reason games can draw repeating objects extremely efficiently with low overhead.*

## LAB CHECKPOINT | MANUAL COMMIT | DO NOT SKIP

To receive credit for your assignments, you must manually commit to your Repo 4 times (Once at the end of each Part/Section). If you skip this commit (or disable auto commits) you will not receive credit for the previous section.

1. Change the lab’s Title Bar so it says where you are in the lab.
   1. For Example: “John Smith – Lab 1 – Part 1 Complete”
2. Take a Screenshot of the lab window running showing your work. (Windows Key + Shift + S)
3. On Windows the screenshot is in your clipboard, open MS Paint and (Ctrl + V) to load it in.
4. Save the image with the same name as your title bar in the root folder of this repository.
5. Open GitHub Desktop for this repo. You should see the new image as a pending change.
6. Create a commit message with the same name as the title bar & commit your changes.

If this section is not perfect but you need to continue forward, you must still do this to get partial credit!

## Part 4 | 100%

#### OVERVIEW | IMPORTANT | THIS SECTION IS MOSTLY OPTIONAL

This part of the assignment is mostly about 3D math and building an FPS in-game camera system. Although this is certainly related/tangential to graphics code, it’s not strictly what this class is about. (Learning Vulkan)

If you are falling behind or are just not interested in getting better at 3D math, you can skip most of this section by using the provided **Camera.h** source file. It contains a function that is basically a turn-key camera system.

If you choose this route, it will be up to you to figure out how to use it. However, what I will tell you is that after you get your updated camera matrix, you will have to overwrite it each frame to the appropriate **uniform buffer**.

Regardless of which path you choose; everyone must do **PART 4X and 4Y** below to have their camera updated properly in the **uniform buffers** and not run into buffer/frame synchronization problems.

### Part 4X

Your Camera/View matrix is no longer going to be a fixed unchanging variable. We are going to need to copy the CPU version (which will change each frame) to the GPU **uniform buffer** or nothing will change visually.

Each time the matrix is **updated**, we need to **copy it** to the **uniform buffer** associated with **this frame**. To do this you can use the helper function **GvkHelper::write\_to\_buffer** which will allow you to **send the updated structure data** you just modified to the **current uniform buffer** for use in the shaders.

### Part 4Y

Depending on what graphics card you have, you **might** see some flickering/hitching while moving the camera. This is due to us currently **only using one** of the **uniform buffers** we have for multiple frames. Earlier we made sure to create a separate uniform buffer for each rendered frame to avoid such synching problems. Unfortunately, when we created our **VkDescriptorSet** earlier we only made one. Let’s correct that oversight now.

Start by replacing your single **VkDescriptorSet** with a **std::vector** of them. The idea is that we will have one VkDescriptorSet for each **uniform buffer** you made in **Part 2C**.

Make your way over to **Part 2F** where you made your **descriptor pool**. Adjust the **size** of the descriptor pool and the maximum number of **sets** to have enough room to have **one descriptor for each rendered frame**.

Now we can go back to **Part 2G** where we **allocated** our **VkDescriptorSet** and adjust the code to allocate a descriptor set for each frame instead of just one. (***Hint***: call **vkAllocateDescriptorSets** multiple times)

Starting to see a pattern? Finish setup by going to **Part 2H** and **vkUpdateDescritorSets** so that each of our **descriptor sets point to its corresponding uniform buffer**. (Ex: descriptor[0] -> uniformbuffer[0] etc.…)

Now we can select and **bind** the **appropriate descriptor** and uniform buffer while rendering instead of being forced to share the first one. Find the code in the **Render function** that does this and adjust it as needed.

***Note:*** *Even if you are not experiencing sync issues right now, if you don’t adapt your code to compensate you will experience them at some point when things get more complex. Better to avoid such issues early on.*

### Part 4a (OPTIONAL: USE CAMERA.H TO SKIP to the end)

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 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

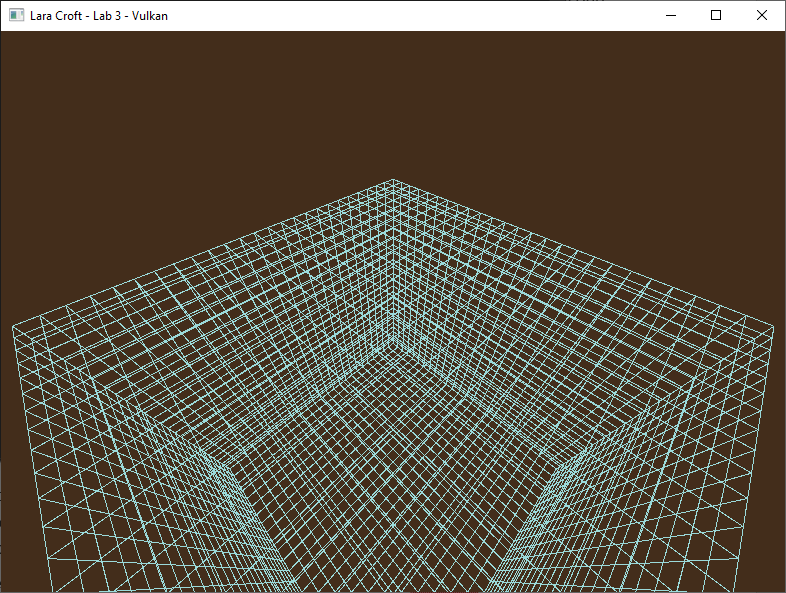
We start with a very basic movement, just moving the camera directly **up and down on the Y axis**. Open the **Gateware docs** to look over all the available input codes in the **Input** namespace. Alternately you can search Gateware.h for **GInputDefines.h** where all the codes are listed.

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



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

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

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

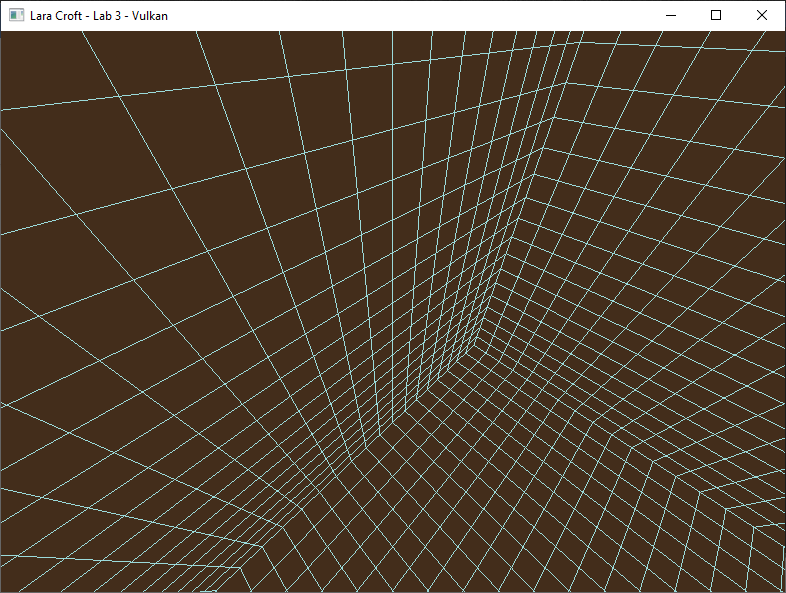
PerFrameSpeed = **Camera\_Speed** \* Seconds\_Passed\_Since\_Last\_Frame

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

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

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

Camera = MatrixMultiplication( TranslationMatrix, Camera )



*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!**

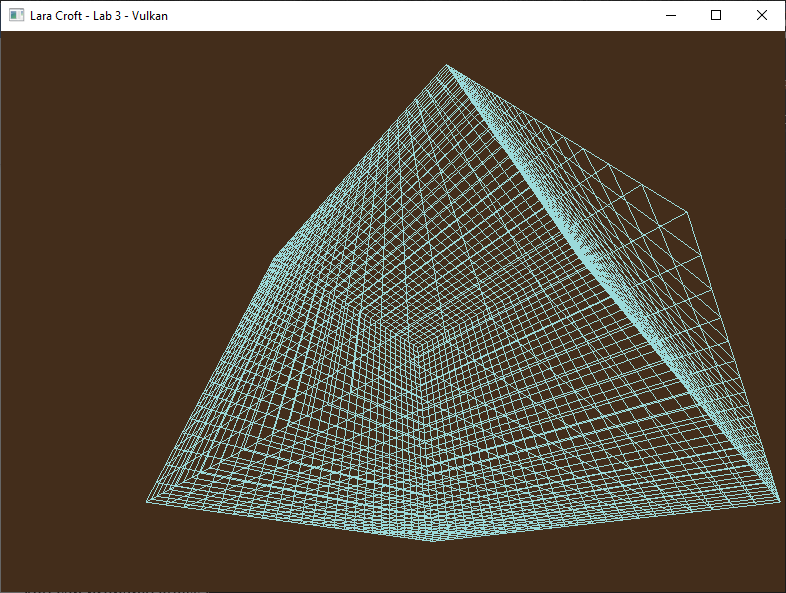
We’re going to start by adding the ability to **tilt the camera up and down:**

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 )



*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:**

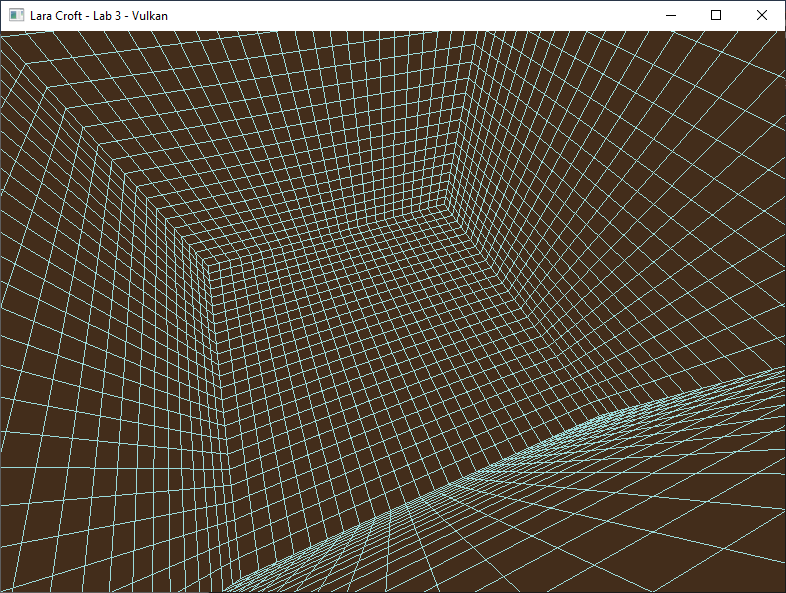
Total\_Yaw = FOV \* AR \* MOUSE\_X\_DELTA / SCREEN\_WIDTH + RIGHT\_STICK\_X\_AXIS\_STATE \* **Thumb\_Speed**

YawMatrix( Total\_Yaw )

Camera.SavePosition()

Camera = MatrixMultiplication( Camera, YawMatrix )

Camera.RestorePosition()



*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 other labs/projects).*

*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.*

## LAB CHECKPOINT | MANUAL COMMIT | DO NOT SKIP

To receive credit for your assignments, you must manually commit to your Repo 4 times (Once at the end of each Part/Section). If you skip this commit (or disable auto commits) you will not receive credit for the previous section.

1. Change the lab’s Title Bar so it says where you are in the lab.
   1. For Example: “John Smith – Lab 1 – Part 1 Complete”
2. Take a Screenshot of the lab window running showing your work. (Windows Key + Shift + S)
3. On Windows the screenshot is in your clipboard, open MS Paint and (Ctrl + V) to load it in.
4. Save the image with the same name as your title bar in the root folder of this repository.
5. Open GitHub Desktop for this repo. You should see the new image as a pending change.
6. Create a commit message with the same name as the title bar & commit your changes.

If this section is not perfect but you need to continue forward, you must still do this to get partial credit!

# 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! Vulkan can be a somewhat intimidating API at first, but as you can see it utilizes many of the same math & concepts you already encountered in CGS.

Your 4th lab assignment in the course will expand your understanding of Shader Resources and Descriptor Sets. You will also load a 3D Model and apply basic lighting algorithms using the flexibility of the pixel shader.

# 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.

## Vulkan API

<https://vulkan.lunarg.com/doc/view/latest/windows/apispec.html>

<https://www.khronos.org/files/vulkan11-reference-guide.pdf>

[ebooks.fullsail.edu (if the link does not work directly, copy it to your browser)](C:\\Users\\lnorr_000\\AppData\\Roaming\\Microsoft\\Word\\ebooks.fullsail.edu (if the link does not work directly, copy it to your browser)https:\\learning.oreilly.com\\library\\view\\vulkantm-programming-guide\\9780134464701\\)

[https://learning.oreilly.com/library/view/vulkantm-programming-guide/9780134464701/](C:\\Users\\lnorr_000\\AppData\\Roaming\\Microsoft\\Word\\ebooks.fullsail.edu (if the link does not work directly, copy it to your browser)https:\\learning.oreilly.com\\library\\view\\vulkantm-programming-guide\\9780134464701\\)

[https:/github.com/SaschaWillems/Vulkan](https://github.com/SaschaWillems/Vulkan) (will not transfer directly, but you can study the code for some insight)

<https://github.com/KhronosGroup/Vulkan-Guide> (nice overview of more specific resources)

## HLSL High Level Shading Language

<https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-reference>

*Note: The above docs often refer to Direct3D APIs. Modern Vulkan can also use the language. You should just study the syntax of the language when using it with Vulkan as other things like compiling are done differently.*

<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

* Peeking at Camera.h, std::chrono<> seems like a useful library. Do you have any more sample code?
  + You can use 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 Vulkan API correctly?
  + Aside from reading the docs and making sure the code compiles, we have enabled run-time debug output in the Vulkan API. Be sure to pay close attention to the console window when running the program. Any non-fatal mistakes you make will be reported by the Vulkan validation layer and printed there.
* I am really lost on part XX and the steps after, Vulkan Descriptor Sets are not making sense to me. Help?
  + Descriptor Sets are without a doubt one of the most challenging parts of the Vulkan API to understand. I included some sample code with this assignment showing how they are used to attach a uniform buffer to the vertex and pixel shaders. Studying this code should help you get through this section more easily.
  + The first 5 minutes or so of [this video](https://youtu.be/d5p44idnZLQ) is an excellent visual break down of Descriptor Sets in Vulkan. If you are struggling to wrap your head around what is going on I highly recommend giving it a quick watch.
* Visual Studio doesn’t seem to be detecting the errors in my shaders, 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.
  + Vulkan uses a binary intermediate language called SPIR-V that higher level shader languages like HLSL and/or GLSL must be compiled into. If there are any issues when converting your code to SPIR-V the **shaderc** compiler will note the error and I added code to print it to the console. Keep your eyes on it.
  + It *is* possible to have visual studio compile your HLSL code - but the output is not compatible with Vulkan, and it cannot compile Vulkan-specific features like push constants. Once your shaders get complex, I recommend using a dedicated shader IDE like [ShaderEd](https://shadered.org/).
* I have no compiler errors or run-time errors, yet nothing seems to be drawing. What do I do now?
  + Check over your code carefully to ensure you did not miss anything obvious such as having the wrong shader or geometry assigned to a pipeline. (or just setting up your vertex data wrong)
  + Problems like this can be difficult to track down, mainly because your C++ code cannot really see what is happening on the GPU. You can download a third-party tool called [RenderDoc](https://renderdoc.org/) to dig much deeper.
  + 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(Derrick Ramirez) did originally have to write the Vulkan interface to Gateware, setting up a modern Graphics API like Vulkan or Direct3D12 from scratch would quickly turn this into a full blown Project and we only have time for one of those this month. ☺
  + If you still really want to learn how to initialize a 3D API with no dependencies, there are plenty of online resources out there(including a few of my own) on how to do exactly that once you complete this course.