Lab 3 Vulkan

# Introduction

In the third assignment in this course, we will expand upon the basic matrix and shader manipulation we learned last time. We will start by building a 3D shape procedurally and then use the core mathematics of computer graphics (world/view/projection) to view the shape in three dimensions. Last, we add fully 3D camera controls.

# Getting Started

## Preparing to use the Vulkan API

1. Download & install the latest graphics drivers from your laptop/video card manufacturer.
2. Download & install the Vulkan SDK for your platform: <https://vulkan.lunarg.com/sdk/home>
3. Reboot your computer. (or type **taskkill /f /im explorer.exe && explorer.exe** into a command prompt)

## Use CMake to build your assigned Lab template

1. Open the directory containing this document in windows explorer and select the path bar at the top.
2. Type **cmd** into the bar and a command prompt should open. Type: **cmake -S ./ -B ./build** enter.
3. This should generate a solution inside a new folder. Open it and set the lab as your startup project.

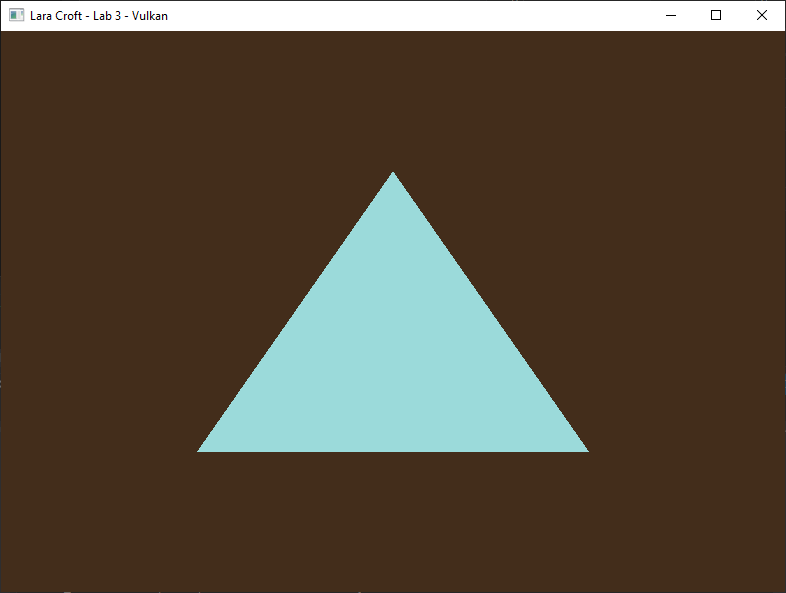
# Lab Assignment 3

## Part 1 | 25%

### Part 1a

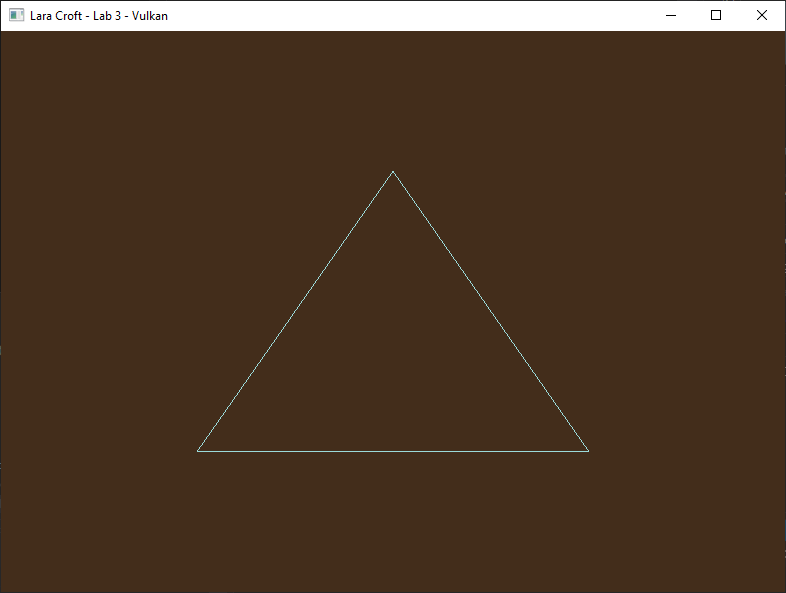
Choose some colors you like(optional). Study the code and familiarize yourself where things are.

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



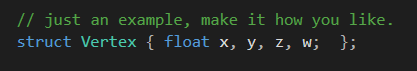
### Part 1b

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.



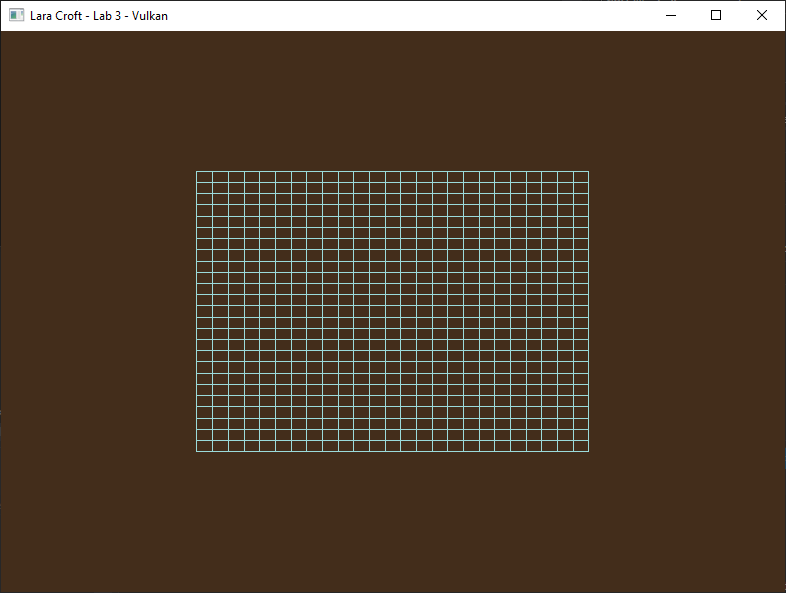
For this to work properly you will need to apply the **rule of three** to the code. First convert the existing triangle to the structure, adding zero and one for the Z and W values respectively.

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**.



## Part 2 | 50%

### Part 2a

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

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

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

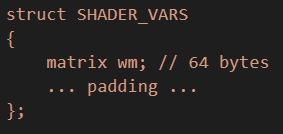
After initialization use the matrix operations to create a matrix that rotates exactly **90 degrees around the** **X axis and translates down the Y axis 0.5f units**. Assign the combined matrix to a new class variable, this matrix will be the first of four unique World matrices.

### Part 2B

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. Though HLSL can be utilized by Vulkan, sometimes we need to give the **SPIR-V** shader *compiler hints* as to how we are going to use something. Place the input attribute **[[vk::push\_constant]]** immediately above your new **cbuffer** structure. This tells Vulkan where your “*Push Constants*” data will go.

### Part 2c

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

Create a **VkPushConstantRange** structure above the existing **VkPipelineLayoutCreateInfo** and fill it out so that it describes your custom **SHADER\_VARS** structure, how much room it needs and where it is going. Link it to the **pipeline layout** creation code. (Use the reference materials to get more details about this structure)

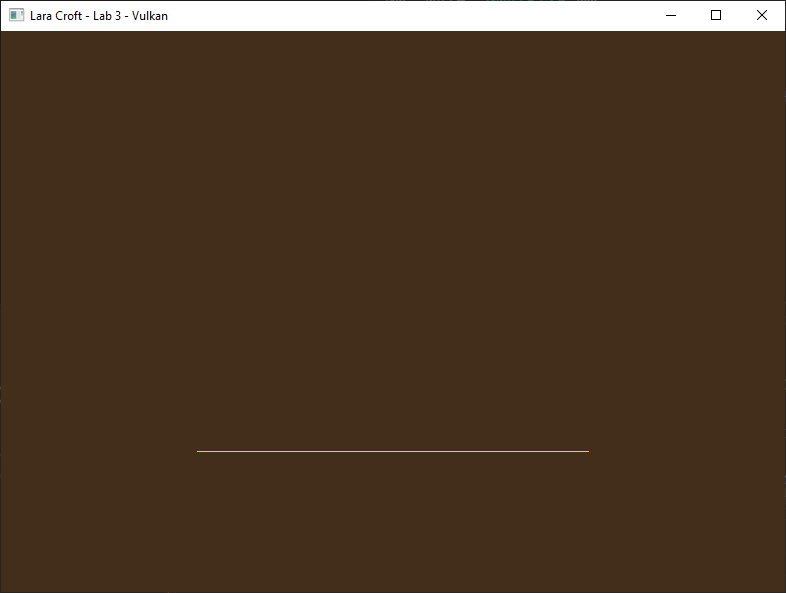
*Note: Push Constants are very convenient but very limited (about 128bytes). A uniform buffer, descriptor layout, descriptor pool and descriptor set are required for significant amounts of data that need to be used by a shader.*

### Part 2d

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

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

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



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

### Part 2E

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

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

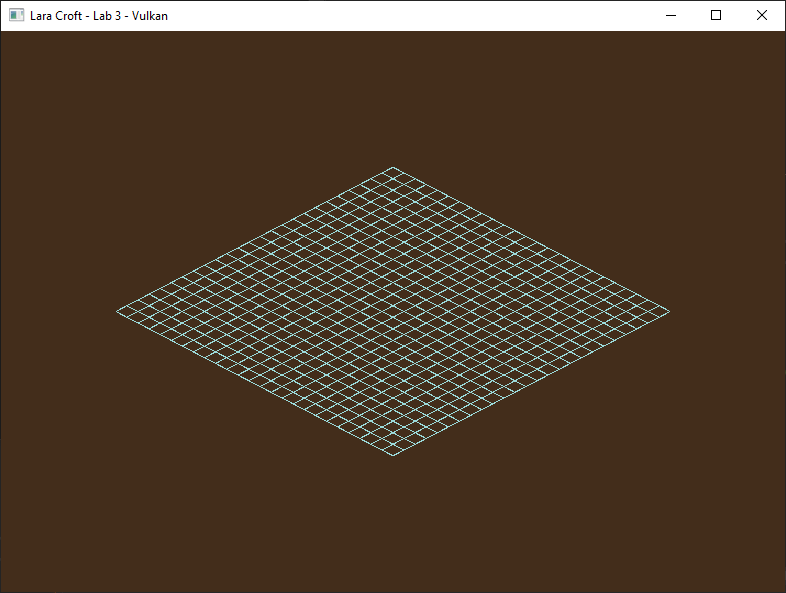
Essentially you want to build a **camera matrix** that has been **moved backwards, up and to the right**. Then you want to **rotate the matrix slightly to the left and down** so its forward(**+Z**) vector is pointing towards the origin.

### Part 2F

Our new matrix is no good to us if we can’t actually use it, thankfully we should have just enough room left in our **SHADER\_VARS** structure to sneak it up there. (Adjust it appropriately)

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

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



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

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

## Part 3 | 75%

### Part 3a

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

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

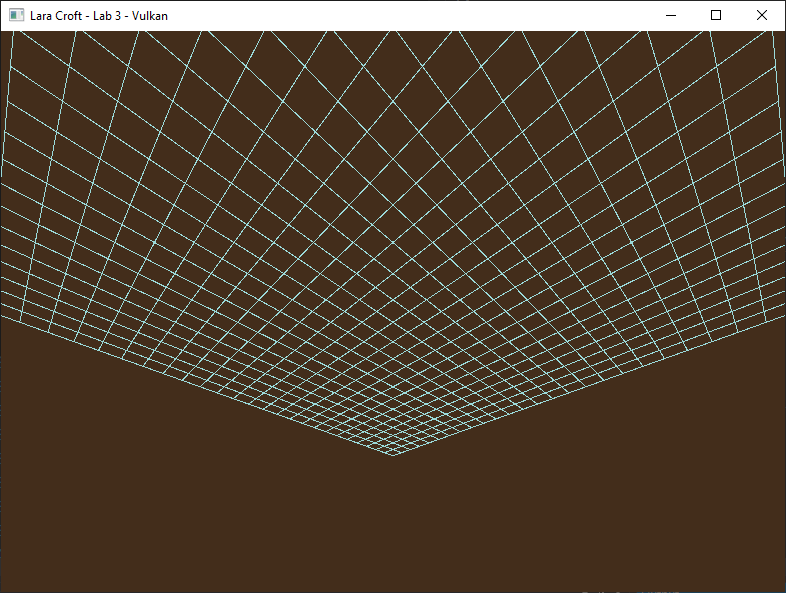
### Part 3B

Like our **view matrix** we now have another matrix we wish to apply to our vertices, unfortunately there is 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.

At this point we could do some extra work to allow for larger data submissions using **descriptor sets and pools**. However, since all we really need for the rest of this lab is one additional matrix; we can get a little creative and save some memory in the push constants by **combing the view and projection matrix** before sending it to the vertex shader.

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 combining the separate view and projection matrices temporarily into a single **viewProjection** matrix and sending that instead.



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

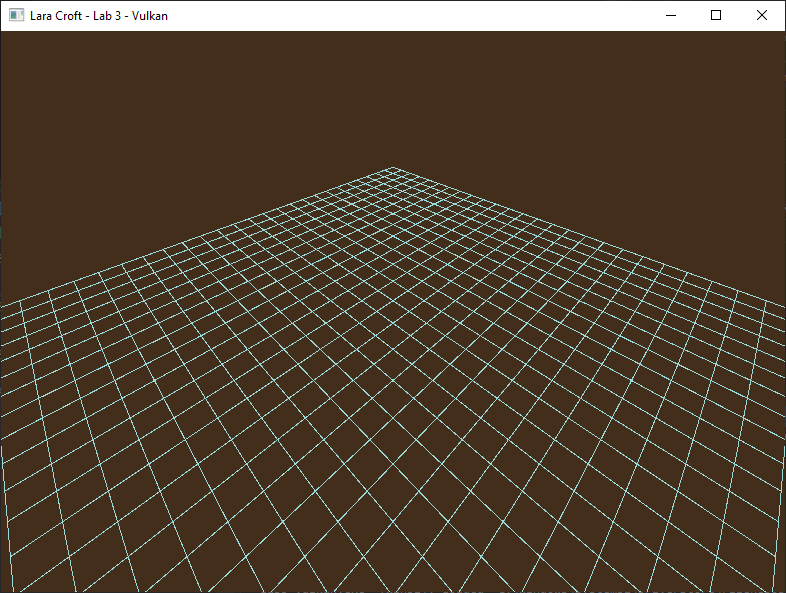
### Part 3C

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 3B](#_Part_3B) 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.*

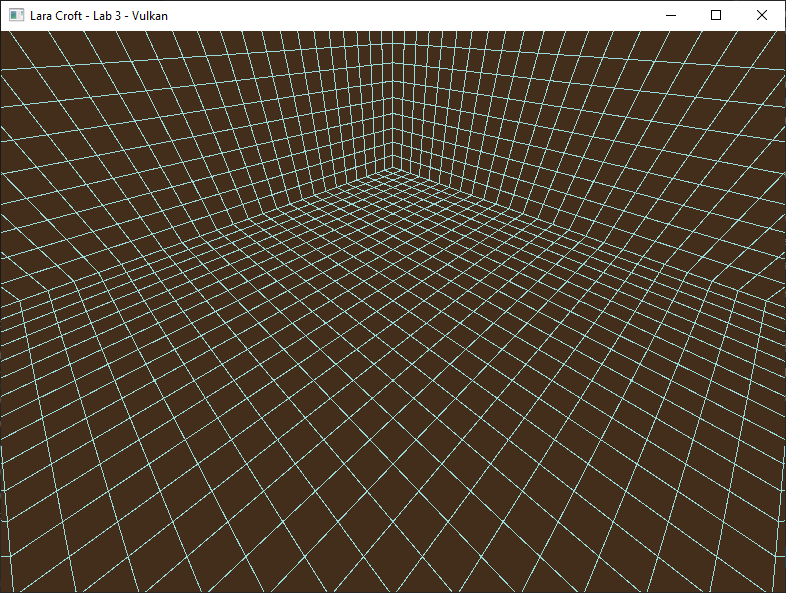
### Part 3D

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 3E

After drawing the current grid (A.K.A the floor) draw the remaining sections by **updating the world matrix** in the vertex shader using **vkCmdPushConstants** before **re-drawing the same grid** in the five new locations/orientations.



***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 lab we will learn to add both **Keyboard and Mouse** support as well as **Game Controller** support via the Gateware API. Having any PC compatible **XBox controller** is recommended for this step, but only a Keyboard and Mouse are strictly required. (*You will still need to add the code for the controller however*)

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

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

### Part 4B

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

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

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

### Part 4C

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

### Part 4D

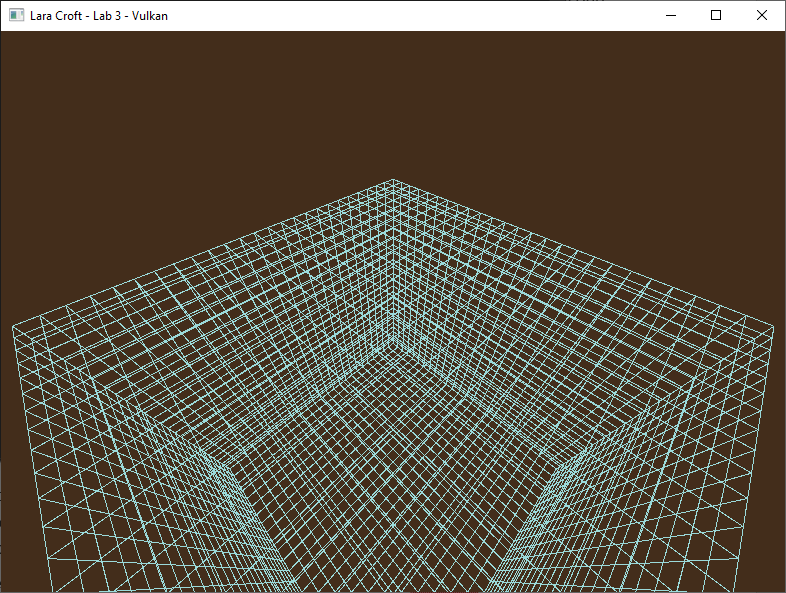
We start with a very basic movement, just moving the camera directly **up and down on the Y axis**. Open the **Gateware docs** look over all the available input codes in the **Input** namespace. Alternately you can search Gateware.h for **GInputDefines.h** where all the codes are listed.

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

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

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

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



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

### Part 4E

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

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

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

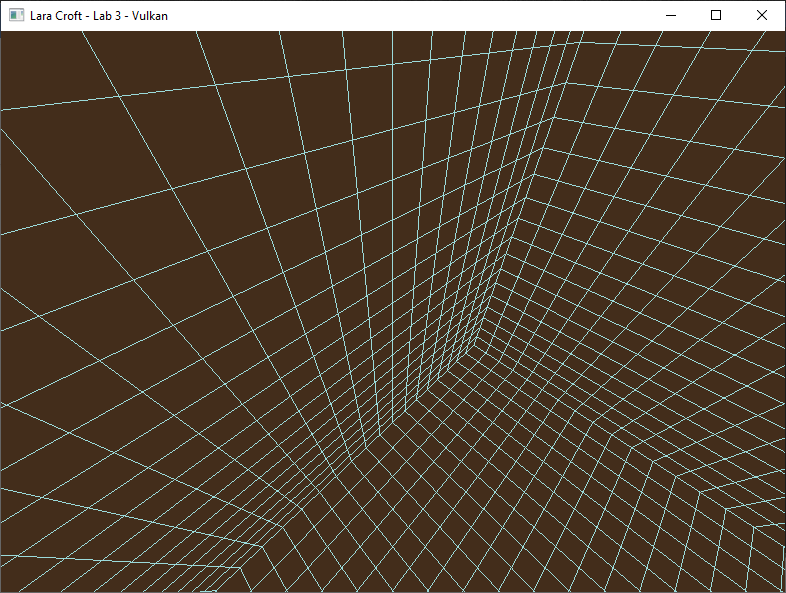
PerFrameSpeed = **Camera\_Speed** \* Seconds\_Passed\_Since\_Last\_Frame

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

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

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

Camera = MatrixMultiplication( TranslationMatrix, Camera )



*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!**

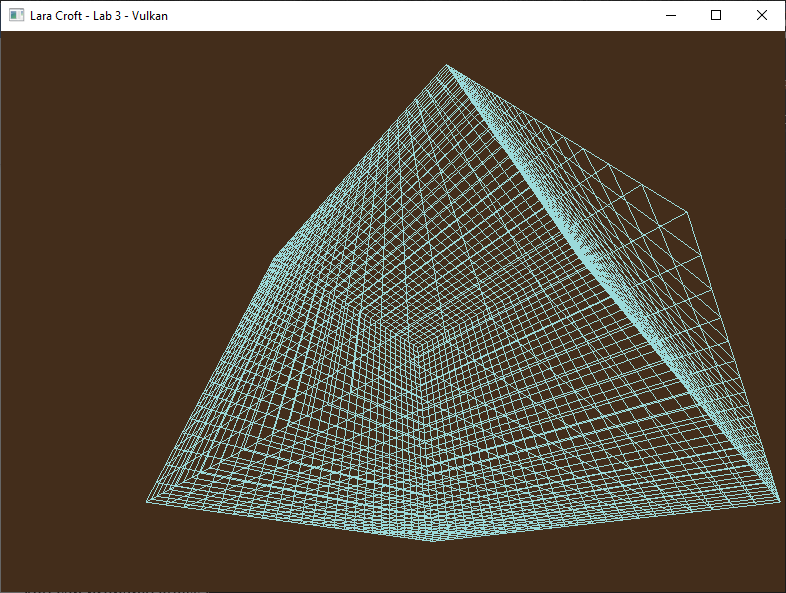
We’re going to start by adding the ability to **tilt the camera up and down:**

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:**

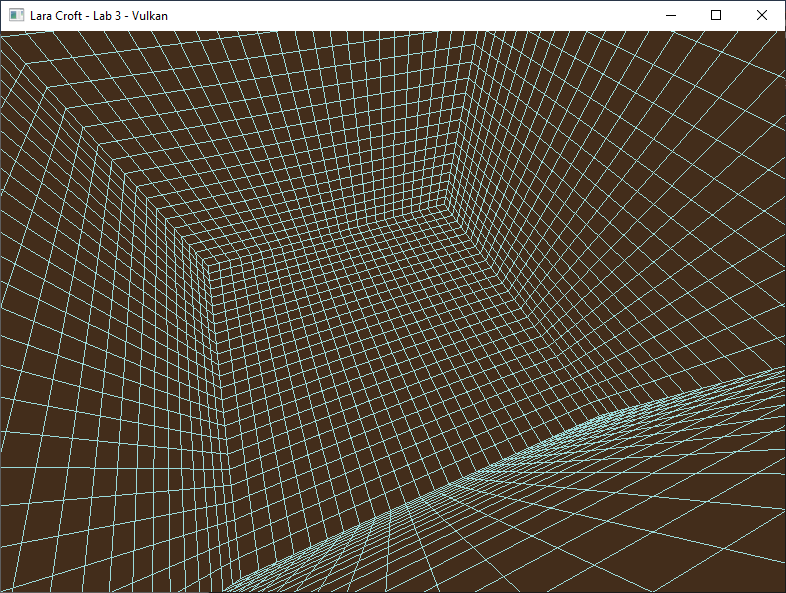
Total\_Yaw = FOV \* AR \* MOUSE\_X\_DELTA / SCREEN\_WIDTH + RIGHT\_STICK\_X\_AXIS\_STATE \* **Thumb\_Speed**

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. ☺*

# 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 shares many of the same concepts of other APIs you have already encountered.

Your final guided assignment in the course will have you loading your first 3D Model and applying a basic lighting algorithm 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)

[..\..\..\Gateware\documentation\html\index.html](file:///C:\Users\lnorr_000\AppData\Gateware\documentation\html\index.html)

*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 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.
* The HLSL shader code appears to just be a string, 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. 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.
  + Once you have installed RenderDoc, in main.cpp uncomment the line "VK\_LAYER\_RENDERDOC\_Capture". 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(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 from a Lab 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.