MASTERY 2 Vulkan

# Introduction

The second mastery assignment in this course will show you how to create and manipulate buffers containing shader variables. Then we will use said variables to manipulate a basic on-screen shape. Additionally, you will continue to familiarize yourself with how to create and submit geometry to a 3D graphics API.

# 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 API 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 it as your startup project.

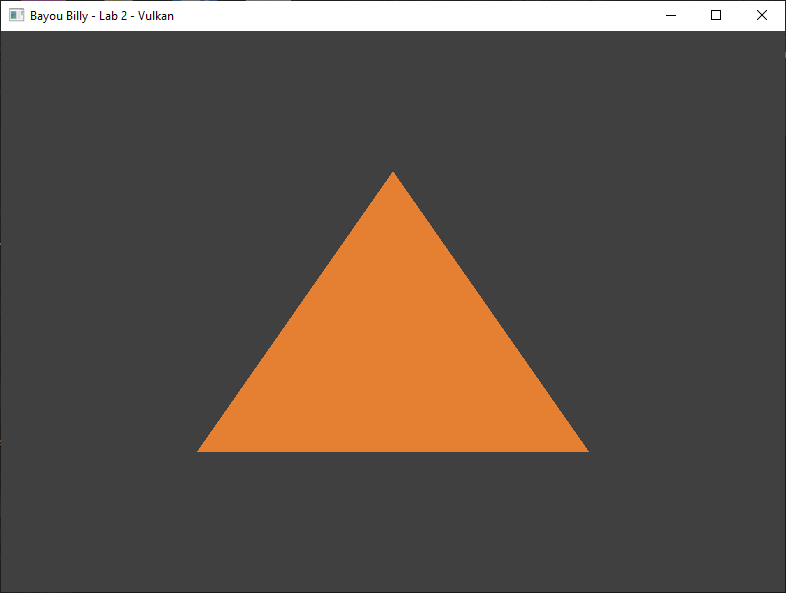
# Mastery 2

## Part 1 | 25%

### Part 1a

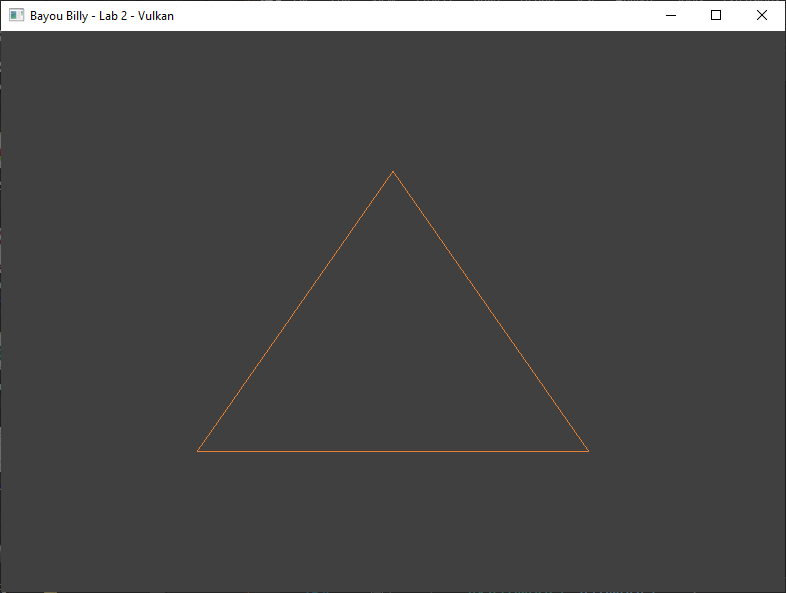
From here on out choose any colors you like (optional). Study the code and familiarize yourself where things are.

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



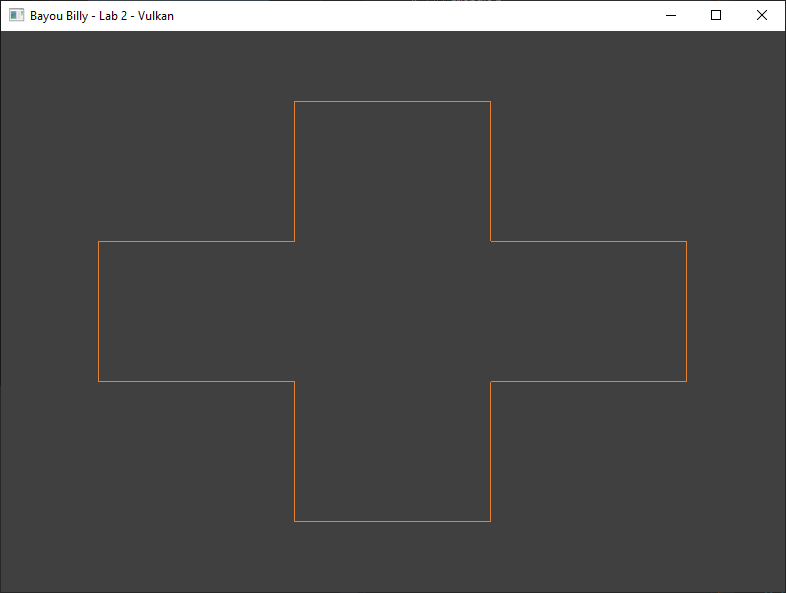
### Part 1b

Next, we are going to switch the TOPOLOGY to be able to draw a STRIP 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

Now that we can successfully draw lines where we want, we are going to draw a “+” symbol using our lines which will serve as our ever so slightly more complex shape for this practice assignment.



## Part 2 | 50%

### Part 2a

Our next goal is to get this shape to start spinning around. 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 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)

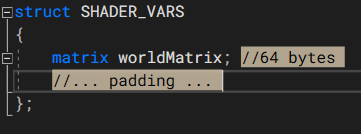
In the **Render()** function use the rotation operations to create a matrix that spins around the **Z axis**. We recommend using **std::chrono** to make the movement time based rather than frame based.

### 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 rotating 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 rotating 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)

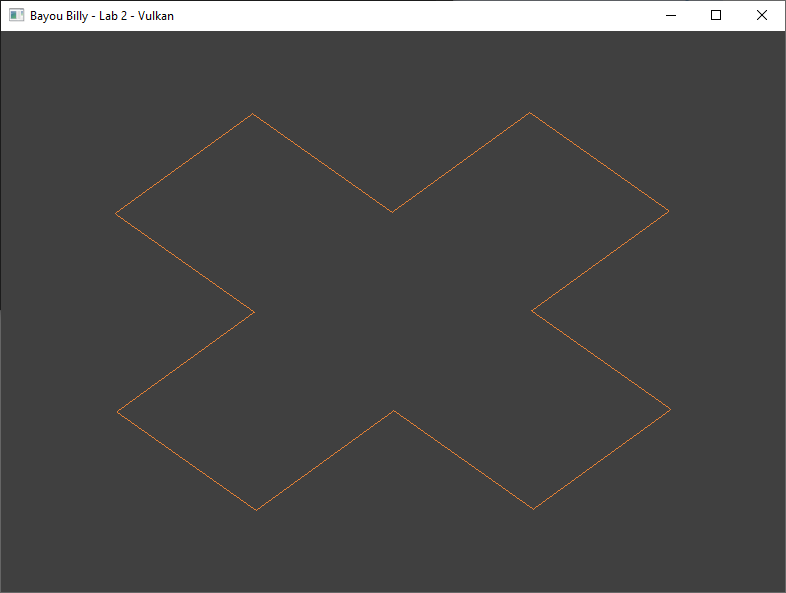
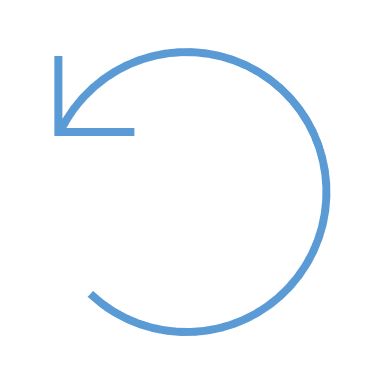
*Note: Push Constants are very convenient but very limited (about 128bytes). A uniform buffer, descriptor layout, descriptor pool and descriptor set are required for significant amounts of data that need to be used by a shader.*

### Part 2d

In **SetUpPipeline()** 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 Plus Sign in motion:



## Part 3 | 75%

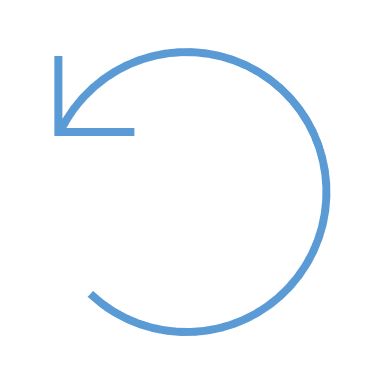
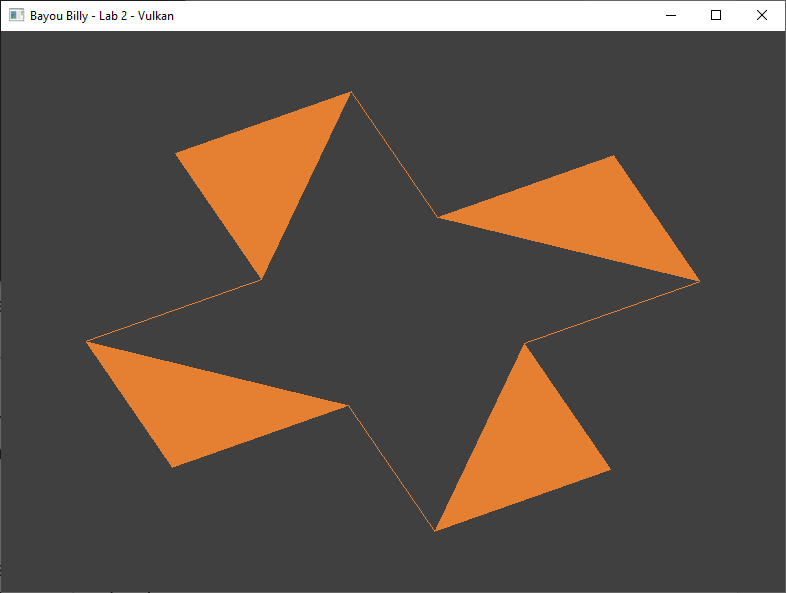
### Part 3a

Ok, time to add some substance to this shape by filling in the interior with some triangles. We will start by telling the GPU we want to draw the shape again, but this time with triangles as our chosen **primitive topology**.

In the class declare another **VkPipeline** object for our second drawing and be sure to free it in the **CleanUp()**. After the first VkPipeline is initialized make this one using the same arguments BUT change the *primitive topology* to a **list of triangles** before doing so. (The way vertices & indices are combined is referred to as *Input Assembly*)

### Part 3b

With our second pipeline ready to go, we will now draw the shape again. This time we **bind** it to the **command buffer** before **drawing** the second time. If you look inside **Render()** you can see this already happening for the first draw. Just switch the pipeline to the new one and do it again, you should end up with something like this:

The good news is we already have the all the vertices we need. But as you can see, the bad news is that the GPU cannot simply predict the way in which we want theses triangles drawn. (thus, no filled interior)

### Part 3c

Alright, the 12 unique vertices we currently have could be duplicated and re-ordered multiple times to form the appropriate triangles but that just seems a bit clunky. Not only would it be awkward, it would break the existing set of lines that form our perimeter strip; there must be a better way.

Thankfully, there is: **Indexed Geometry** is a way to re-use a vertex multiple times when drawing a set of primitives. Not only can it be more convenient to work with, it also uses less memory and therefore less GPU bandwidth!

The idea is quite simple: make an array of indices(integers) that are offsets into the original vertex buffer. You then load these into a **VkBuffer** and tell the **VkCommandBuffer** during drawing to use these numbers as a substitute for raw vertex data.(example: just a 4 vertex square could form 8 unique triangles, 4 clockwise, 4 counterclockwise)

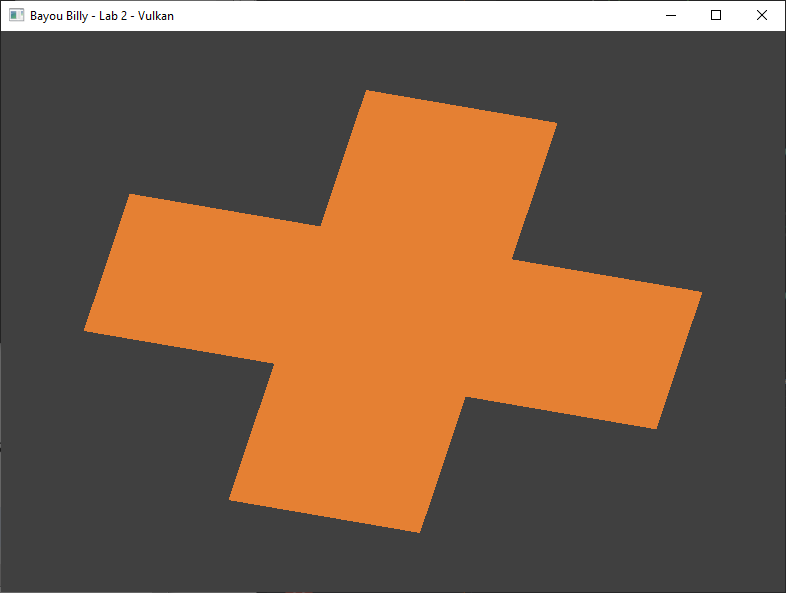
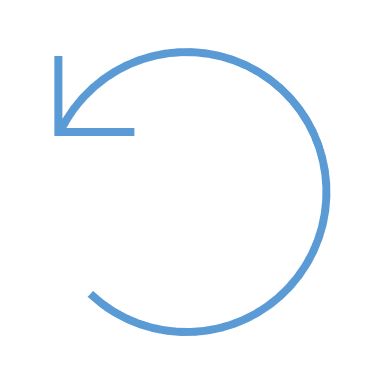
Enough chatter, time to dig in. Make another **VkBuffer & VkDeviceMemory** in your class and be sure to free them during **CleanUp()**. Now go to where the vertex buffer is being created and make an “index” array where every three numbers form a **clockwise** triangle. Make enough triangles to cover the entire **Plus Sign** without overlapping any. This process can be hard to visualize in your head, so it is HIGHLY recommended you draw this out with labels & numbers.

Once you are done, use the same method as the Vertex Buffer to upload your index data to the new **VkDeviceMemory** and link it to a **VkBuffer** indexhandle.

*Note: Scalable memory management in Vulkan is very important but beyond the scope of these assignments.*

### Part 3d

Assuming you setup your index buffer correctly, this last section should be a breeze. Just go to where you are currently calling the second **draw** with the **new pipeline**. Use **vkCmdBindIndexBuffer** to connect your new index buffer to the **command buffer** and swap the draw call itself to use **vkCmdDrawIndexed**.



*Tip: If you do not see all your triangles, some of them may be wound the wrong way. This will cause the API to think that they are facing away from the camera and they will be back-face-culled.*

Even if the shape draws you could have an issue with some of the arguments when creating the buffer. Be sure the check the **console window** for any reported API errors and address them immediately.

## Part 4 | 100%

### Part 4a

It is about time to upgrade our vertices so they can support random colors.(and eventually other stuff)

*Note: Starting this part may break your drawing code for a time.*

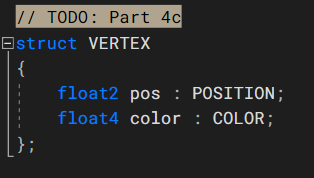
Create a new vertex type that also contains 4 floats for color RGBA. Transfer your **Plus Sign** to this new vertex format and randomize each color by using rand()/static\_cast<float>(RAND\_MAX). This will replace the original set of vertex data. (we will need to adjust quite a few things to make this compatible)

### Part 4b

Create a second vertex shader and pixel shader to work with the new type of vertex. Make sure not to place them in the build folder, and be sure to add them to the CMakeLists.txt. Start by copying the current shaders and their compilation code. Don’t forget to clean up their memory as well!

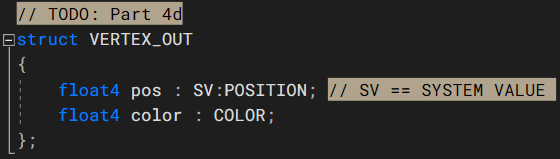
### Part 4c

In the new vertex shader create a struct of the same name as your new vertex type but use HLSL types for the members such as float2 & float4. Append **semantics** after each variable by using the “**:**” operator followed by a descriptive name such as “POSITION” and “COLOR”.



### Part 4d

Adjust the “main” argument of the vertex shader so it takes your structure instead of a float2. Create a “VERTEX\_OUT” struct that is the same as the input struct expect the position has 4 floats instead of just 2 and the semantic is SV\_POSITION. (SV\_POSITION is a special type of HLSL **semantic** that indicates an xyzw coordinate is to be used for drawing an NDC shape during rasterization) Change the return type of main to this new output struct.



Finally **adjust the code** inside the function to initialize one of your new output structures using the data from the input structure so it can be sent to the rasterizer hardware. (be sure to also include the existing matrix multiply)

### Part 4e

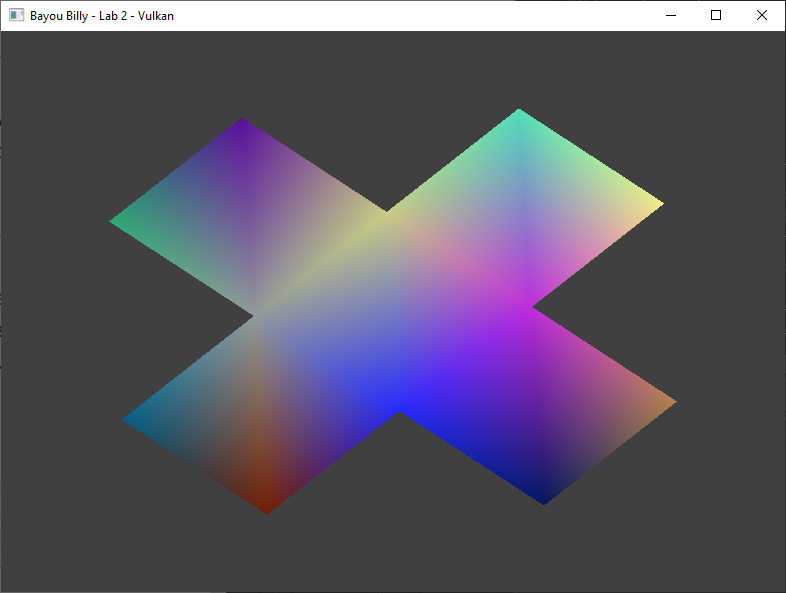
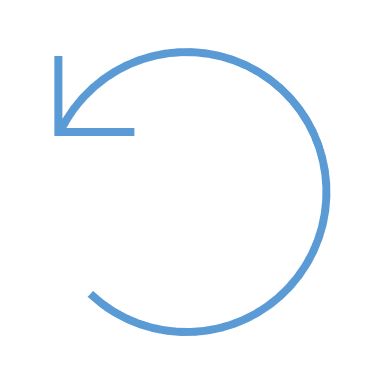
Adjust the Pixel shader so it also has the new vertex output structure and accepts it as the input argument to the main function. Return the color of the input variable instead of hardcoding it.

### Part 4f

*(Hint: If you have not already done so, now is a great time to crack open the Vulkan API documentation.)*

Adjust the existing **vertex input binding description** so that it exactly describes the memory layout of your new vertex type. (*note:* there are now **two** attributes, adjust the code appropriately) You will also need to switch the Vulkan pipeline to also **use the new shaders** you just authored instead of the original ones. Once you do this, your program should compile again and be ready to draw again.

Carefully examine the debug output of your program. Any compiler errors in your new shaders or run-time errors in the API will be printed to the console. Correct these and you should get output that looks like this:

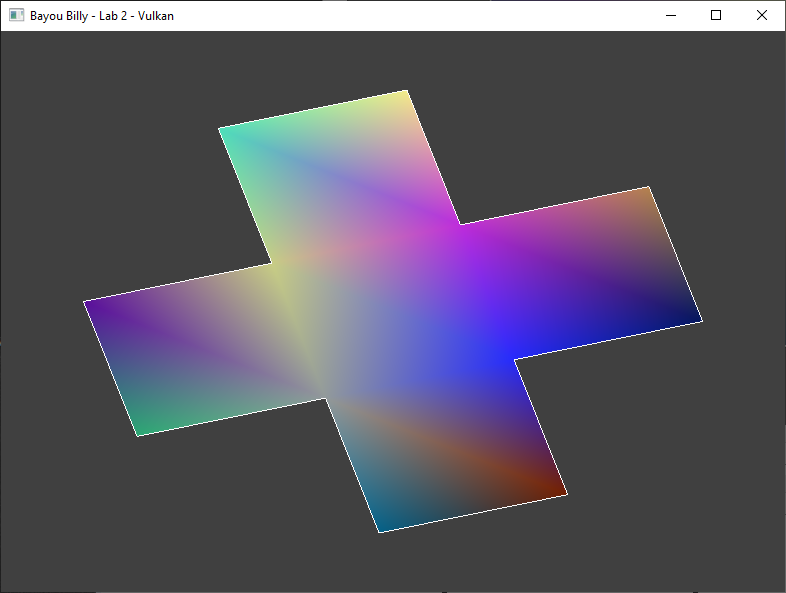
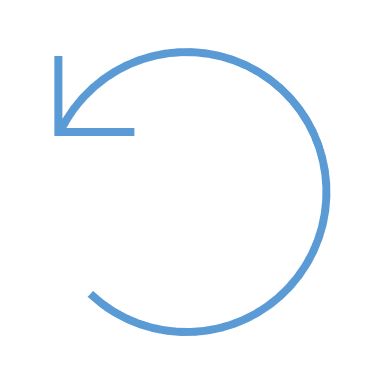


### Part 4G

Excellent! the final issue we need to address is that our **outline is missing** (or is it?). Because the outline also uses the same pixel shader as the triangles, the lines seem to disappear as they interpolate along the edges of the shape. The easiest way to solve this issue it to just tell the pixel shader for the outline to use the older version.

Adjust the outline’s **VkPipeline** to select the more basic **pixel shader**(I also set the color to white). Note that you will need to ensure we still fill the interior with the barycentrically interpolated colors when drawing triangles.

Since there is a **ZBuffer** in the template by default, the triangles will **“z-fight”** with some pixels from the lines and create a noisy appearance. **Switch the drawing order** and **disable depth testing** you should end up with this:



***Tip:*** *The order in which draws & bindings are submitted to a specific command buffer does matter.*

# Summary

Great job! You are now lightly introduced to one the most powerful rendering APIs on the planet. We also covered rudimentary submission of shader variables to the GPU, and how to modify the GPU pipeline to accept a variety of different vertex data & attributes.

Once you have completed assignment 2 you will have experienced all the fundamental features available to modern graphics APIs. The Level Renderer requires a strong understanding of all these features, so make sure to review your code carefully and remember where everything is and what it does. (Comments help!)

# 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

* 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.
* 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 am struggling to complete Part 4. Any additional places I can look to help figure out what may be wrong?
  + Part 4 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 should find it in today’s handout.
* 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 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.