Lab 1 Vulkan

# Introduction

The first lab in this course is designed to introduce you to working with basic types of geometry and the shaders needed to draw them. Additionally, you will be introduced to the CMake build system used to compile the base code provided to you in this course’s various labs.

# 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>

## Use CMake to build your assigned API template

1. Download & install the CMake build tool [cmake.org](file:///C:\Users\lnorr_000\AppData\Roaming\Microsoft\Word\cmake.org) (be sure to check “install for all users”)
2. Reboot your computer. (or type **taskkill /f /im explorer.exe && explorer.exe** into a command prompt)
3. Open the directory containing this document in windows explorer and select the path bar at the top.
4. Type **cmd** into the bar and a command prompt should open. Type: **cmake -S ./ -B ./build** enter.
5. This should generate a solution inside a new folder. Open it and set it as your startup project.

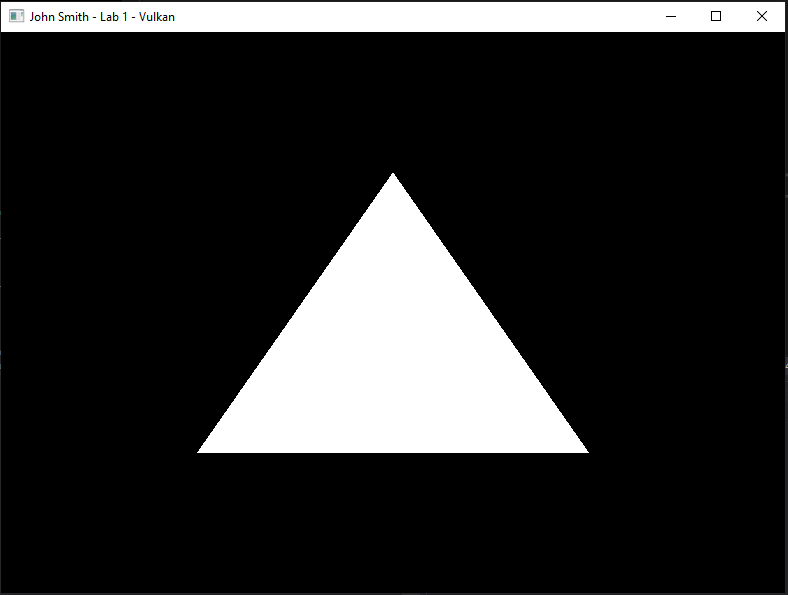
# Lab 1

## Part 1 | 25%

### Part 1a

Clear background to black, set Triangle to white. Study the code and familiarize yourself with where things are.

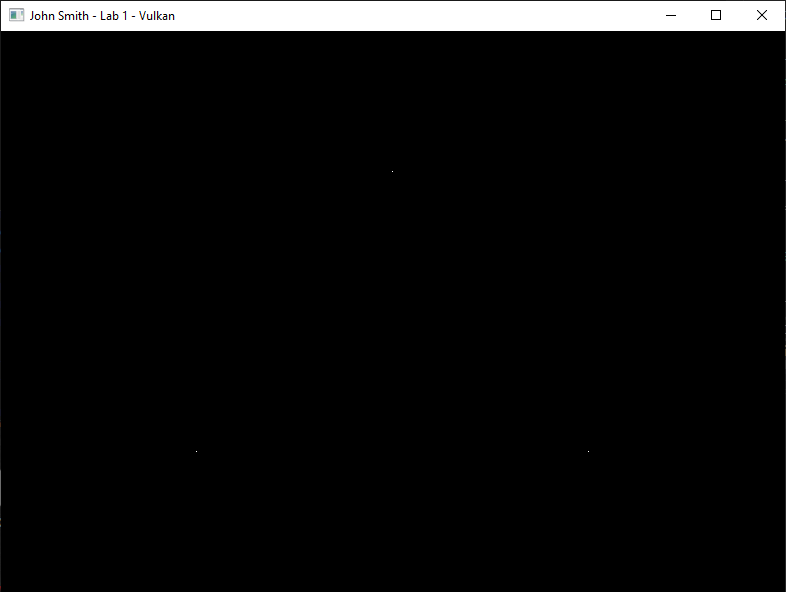
Use the “SetWindowName” function from GWindow to place your name and lab variant at the top.



## Part 2 | 50%

### Part 2a

Switch the polygon mode to POINT, you will find this setting as part of the creation of the Vulkan pipeline.



### Part 2b

Now modify the vertex buffer to draw 10,000 random “stars”. Keep in mind just like in CGS, NDC is king. I would recommend placing a constant or #define somewhere for the number of stars and the math to generate them.



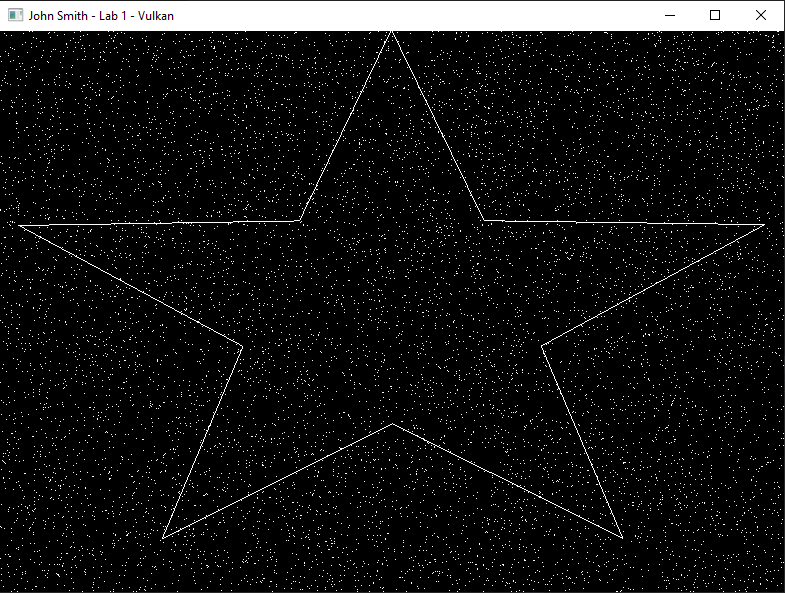
## Part 3 | 75%

### Part 3a

Create a second vertex buffer that contains a 2D star. Do not forget to Free the memory in CleanUp().

### Part 3b

Create another Vulkan vkPipeline(using the same values from before), adjust the TOPOLOGY for drawing connected lines and set the vertex buffer to the star for this new pipeline only. Draw the star using the second pipeline.



## Part 4 | 100%

### Part 4a

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

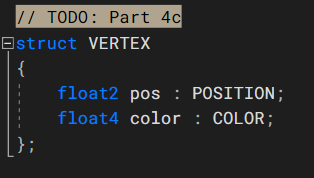
Create a new vertex type that also contains 4 floats for color RGBA. Transfer your star to this new vertex format and randomize each color by using rand()/static\_cast<float>(RAND\_MAX).

### Part 4b

Create a second vertex shader and pixel/fragment 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!

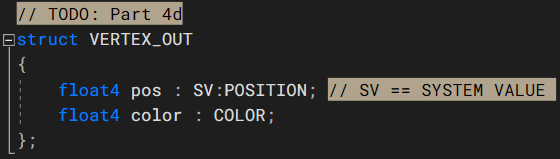
### Part 4C

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 except 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.

### Part 4e

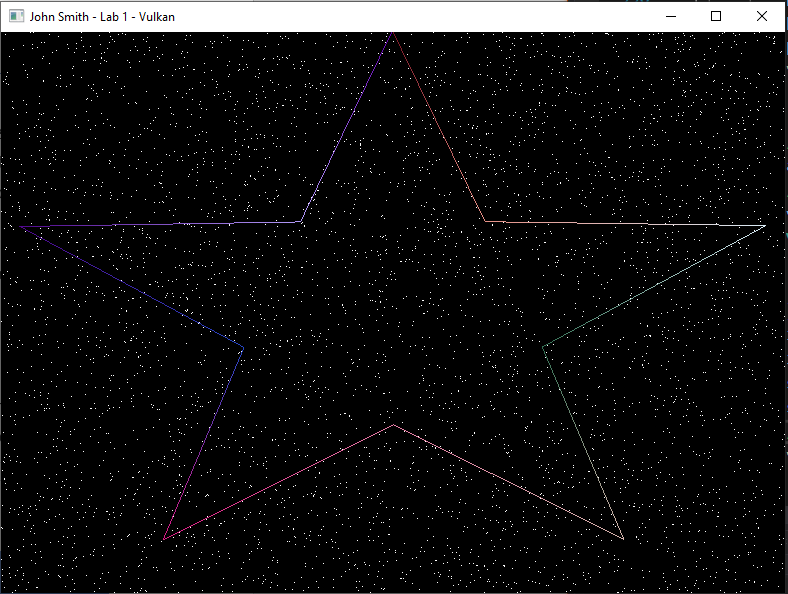
Adjust the Pixel/Fragment shader so it also has the new vertex output structure and accepts it as the input argument to the main function. Return the color of the input variable instead of hardcoding it.

### Part 4f

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

For the second pipeline object you made, create a new **vertex input binding description** that 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 secondary “star” 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:



# Summary

Congrats! Nice work! This was a crash course on the Vulkan API. You’ve learned how to render points & lines utilizing Vertex and Pixel/Fragment shaders. You have also learned about the Vulkan **Pipeline** object and how it directly impacts the geometry that you submit for rasterization.

In the next lab you will learn about using indexed geometry, spend more time debugging the rule of three, and learn how to upload global C++ variable data to your HLSL shaders using Vulkan **Push Constants**.

# 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)](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/](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/)

## 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. Recently Vulkan can also make use of the language. You should just study the syntax of the language when using it with Vulkan as other things like compiling are done differently.*

## Gateware

We will be using this API occasionally throughout these labs 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 **shader** 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.
  + 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 labs without Gateware? I prefer to do things from the ground up.
  + Technically yes, practically no. Someone (Derrick Ramirez) originally had to write the Vulkan interface to Gateware. However, just setting up a modern Graphics API like Vulkan or Direct3D12 from scratch would easily eat an entire week. Unfortunately, it is not something we have time for in a one-month course.
  + 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.