**Lab/Homework 2**

**Lab Part:**

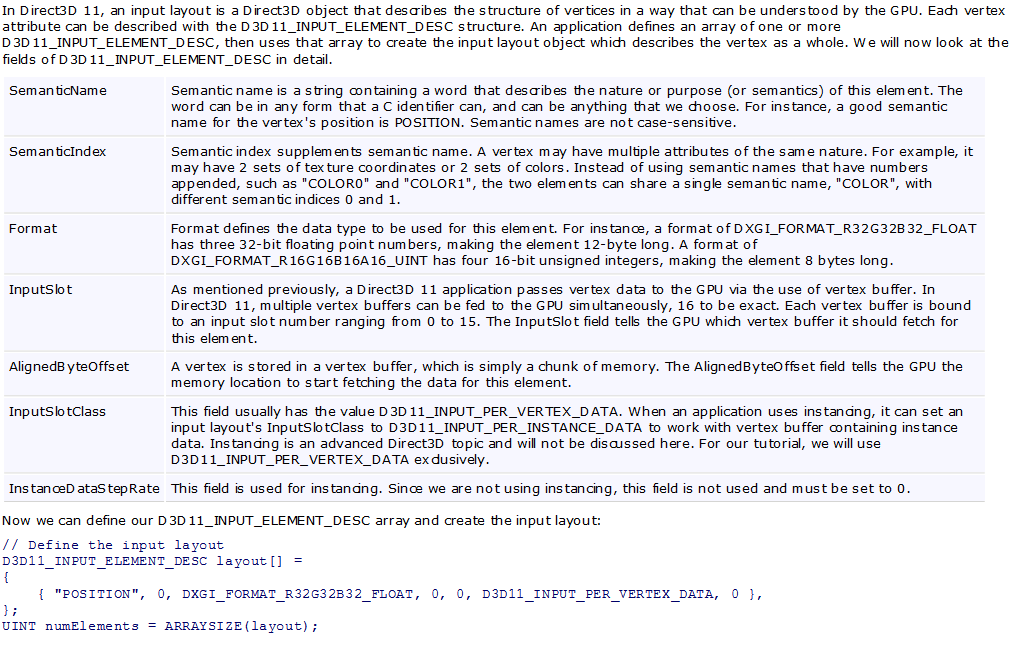
**Go to DirectX Sample Browser. Install the code for Tutorial 2: Rendering a Triangle. Also open the docs. The following lab and homework will be using the code downloaded.**

Questions:

1. In the documentation, go to the section about Input Layout. How is the size of a vertex obtained?

The size of the vertex is conveniently obtained from the size of the structure.

1. Why does the GPU need to know about the vertex layout?



1. What is the vertex shader responsible for?

Sets input layout

1. What is the pixel shader responsible for?

The pixel shader is responsible for calculating the final output color for each pixel of the triangle.

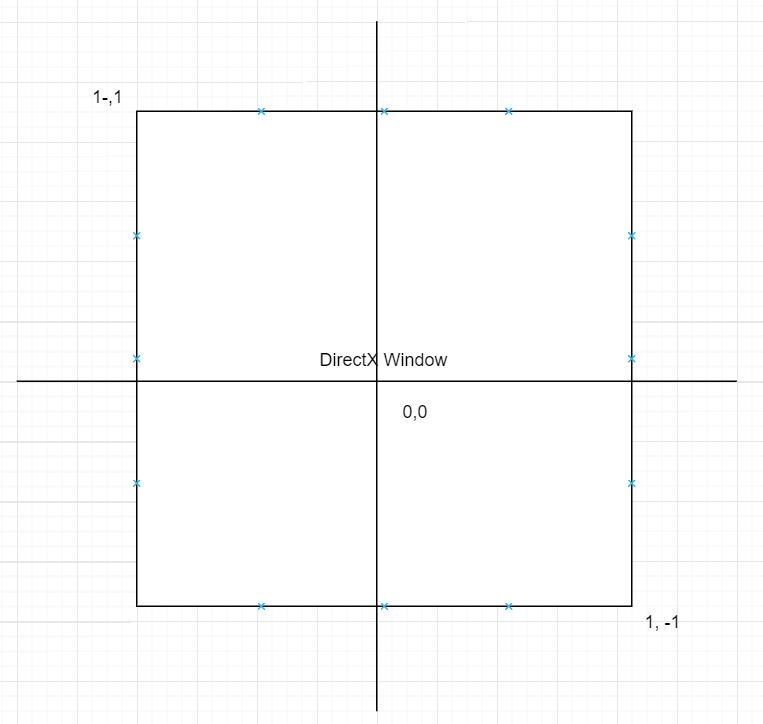
1. In the code downloaded, what is the primitive topology used?

Triangle list

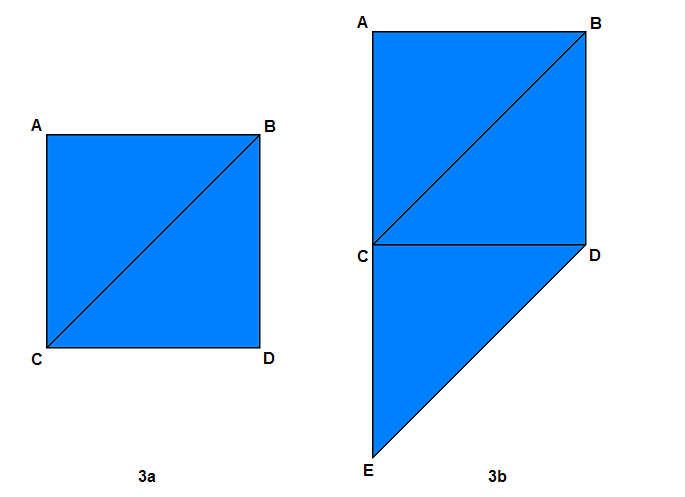
1. Go to where the input layout is defined in the code ( D3D11\_INPUT\_ELEMENT\_DESC layout[] )

Code Walk Through:

1. Go to where the input layout is defined in the code ( D3D11\_INPUT\_ELEMENT\_DESC layout[] ). Input layout allows us to define what information the vertices should have. In our case, we only care about position. In the future, this may change.
2. Given the coordinate system of DirectX, where the center of the window is (0,0), look at the current triangle coordinates in the vertex buffer inside of initDevice. There are currently 3 vertices. Change the triangle so that it is upside down.



1. Add 3 more vertices to the vertices to create a diamond. Run it and see if the diamond shows up. You will notice that you still only see the original triangle.
   1. In order to draw more than just the triangle, we set up vertices a SimpleVertex structure, but that is not enough. We also must find out the size of what we are setting up so we can store it in the buffer (our vertices structure is on the CPU and we need to store this information on the GPU for rendering). Look at the line that has bd.ByteWidth = sizeof( SimpleVertex ) \* 3;. In the triangle example, the 3 here signifies how many vertices you have. Define a variable named “numVertices” or something similar, set it to 6 and change the above line to be bd.ByteWidth = sizeof( SimpleVertex ) \* numVertices;. We need the correct byte size when initializing the vertex buffer and updating this line will do that.
   2. Now when the buffer gets created, the ByteWidth will be correct and you will be able to see your diamond. In hr = g\_pd3dDevice->CreateBuffer( &bd, &InitData, &g\_pVertexBuffer );, when the buffer is created, there is no need to pass anything new into the create buffer device here because we have already taken care of that. So now we have set up the buffer to be able to handle our information, but we haven’t actually drawn anything yet.
   3. Go to the Render function. When Rendering, we need to clear the render view, set up what we want to render, and draw it. We need to specify the Vertex Shader we want to use, the Pixel Shader we want to use, and how many vertices will be drawn in our model (we only have one pixel shader and vertex shader so don’t worry about that part for now). First, change the ClearColor to something different. Inside the Render function, Draw() needs to know the vertex count. Initially we had 3 vertices, but now, you will need to change Draw to draw the number of vertices that now exists in your model (the diamond). You should now be able to see a diamond.
2. Draw either of the following shapes



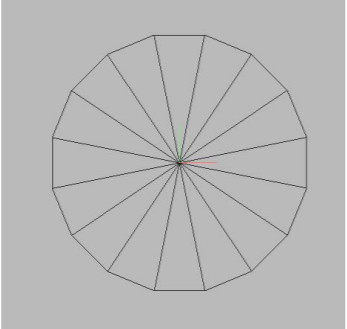
1. Now let’s take a look at the shaders. Go to the .fx file in your project. The vertex shader will get the position of every vertex in the model and pass it to the rasterizer. The position of the model can be changed in the vertex shader (for animation etc). Change the vertex shader. Add a float4 named position and set it equal to Pos. Add the following line after that → position += float4(0.2, 0.0, 0.0, 0.0); then, instead of returning Pos, change it to return position. Run the program and see what has happened. Your drawing should have shifted to the right.
2. The pixel shader is also in the .fx file. Pixel shader will give the color for every pixel in the model. Right now it returns yellow. You can change the pixel shader to draw pixel colors based off of information you have. Create a float4 inside the PS function like this: float4 color = float4(0.0f, 0.0f, 0.0f, 1.0f); Add the following line after it: color += Pos/500.0; Now change the function to return the color rather than the hardcoded value it was sending before. You should see some sort of gradient when the image is drawn.

Read the following documentation

* <https://docs.microsoft.com/en-us/windows/desktop/direct3d9/vertex-buffers>
* <https://docs.microsoft.com/en-us/windows/desktop/direct3d11/overviews-direct3d-11-resources-buffers-vertex-how-to>
* <https://docs.microsoft.com/en-us/windows/desktop/direct3d11/d3d10-graphics-programming-guide-input-assembler-stage>
* <https://docs.microsoft.com/en-us/windows/desktop/direct3d11/vertex-shader-stage>
* <https://docs.microsoft.com/en-us/windows/desktop/direct3d11/pixel-shader-stage>

**Homework Part (50 points):**

1. Draw the following shape. 16 lines



2. Make the background green

3. Make the shape a new color (or a gradient of colors)