BEGINNING METAL

HANDS-ON CHALLENGES

Beginning Metal

Caroline Begbie

Notice of Rights

Notice of Liability

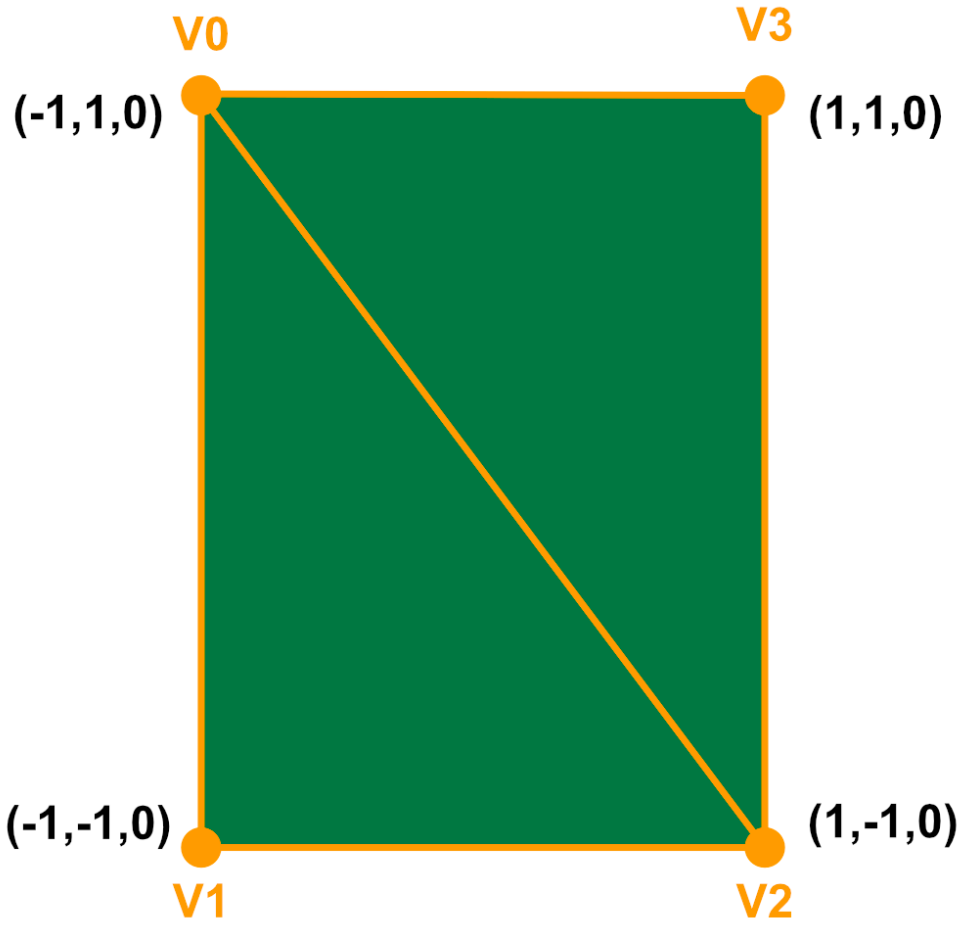
Trademarks

3 Challenge #3: Quads

By Caroline Begbie

In this challenge you're going to create a second triangle to make a quad.

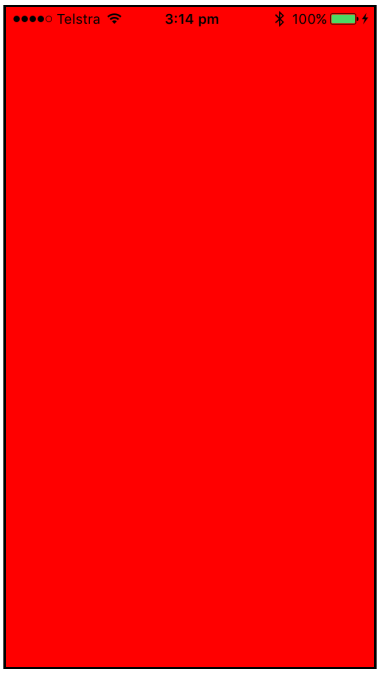
The triangles you create should look like this using these vertices:



In Renderer, change the vertices array to:

```
var vertices: [Float] = [
  -1,  1,  0,  // V0
  -1, -1,  0,  // V1
   1, -1,  0,  // V2
   1, -1,  0,  // V2
   1,  1,  0,  // V3
  -1,  1,  0]  // V0
```

That's it! Build and run. and your screen will now be colored red.

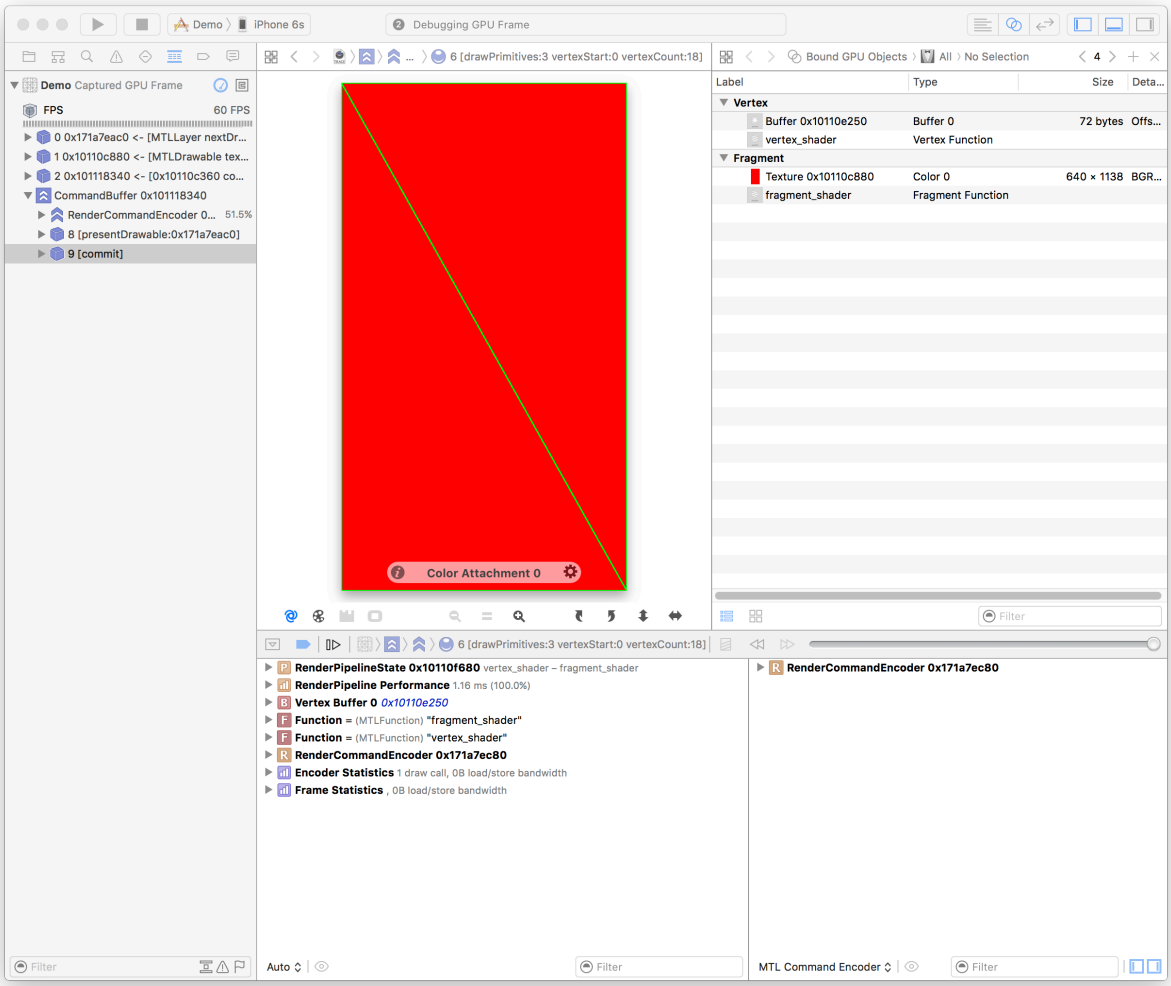


This works because in `buildModel()` you're setting up the size of the Metal vertex buffer dependent upon the number of vertices in the array. So as long as you're creating three vertices array entries for every triangle, the GPU will draw all the triangles you specify.

However, how do you know that you're outputting triangles?

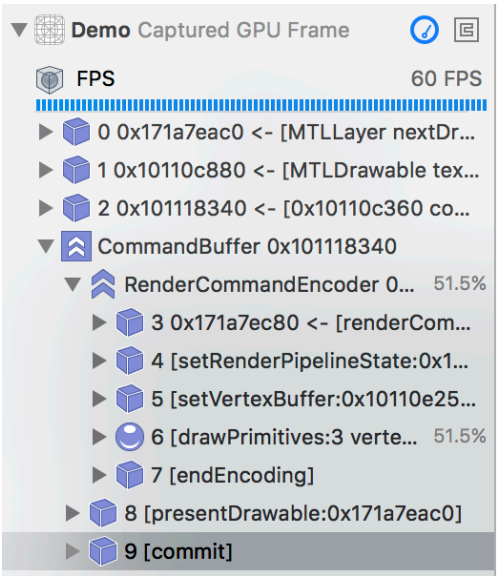
The GPU debugger in Xcode is really excellent. Between your code entry panel and the debug console, there's a debugging strip of icons:

Click on the camera icon and Xcode will take a snapshot of what's happening on the screen and GPU.



You can see your two triangles outlined in green. Earlier in the video, I showed you a triangulated head shot. Using this GPU debugging facility is how I captured that image.

Take a look at the other information you're given. On the left hand pane, click the triangle next to `RenderCommandEncoder` to open it. All the commands in the command buffer are listed. Remember we added `setRenderPipelineState`, `setVertexBuffer` and `drawPrimitive` commands in the `draw(_:)` method in the demo.



At the top right are all the objects that are "bound" to the GPU.

Label	Type	Size	Deta...
Vertex			
Buffer 0x10110e250	Buffer 0	72 bytes	Offs...
vertex_shader	Vertex Function		
Fragment			
Texture 0x10110c880	Color 0	640 × 1138	BGR...
fragment_shader	Fragment Function		

There's the vertex buffer containing the vertices array. Double click on that buffer and see the vertices array listed there. Later on when you're passing values to the GPU, this is a great way of finding out exactly what those values are.

Row	Offset	vertices		
0	0x0	-1.000	1.000	0.000
1	0xC	-1.000	-1.000	0.000
2	0x18	1.000	-1.000	0.000
3	0x24	1.000	-1.000	0.000
4	0x30	1.000	1.000	0.000
5	0x3C	-1.000	1.000	0.000

Going back to the bound objects - next is listed the vertex function. From here you can tell that the GPU is actually using the vertex function that you thought it was using. This can be very useful when you have several functions to choose from.

The fragment texture is the output texture, and the fragment function is also listed.

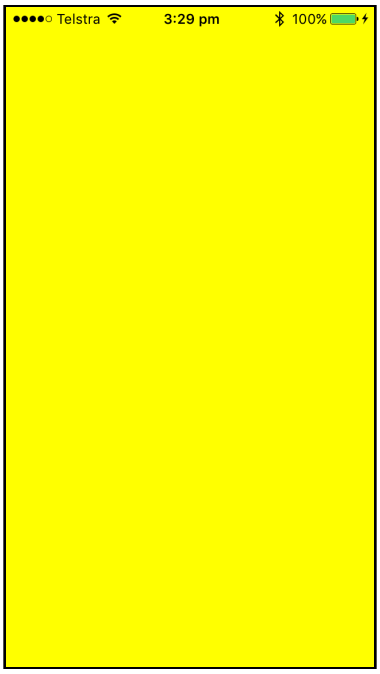
The next part of your challenge is to change the red square yellow. I know that we haven't covered fragment functions in depth, but shader functions are a fundamental part of the rendering process. Remember that vertex and fragment functions run on the GPU, whereas your other code was running on the CPU.

All you have to do here is locate the fragment function in **Shader.metal** and change the red color to yellow.

Your fragment function should look like this when you've finished:

```
fragment half4 fragment_shader() {
  return half4(1, 1, 0, 1);
}
```

Build and run and you should get your two yellow triangles rendering.



Congratulations! You're now rendering triangles with Metal!