BEGINNING METAL

HANDS-ON CHALLENGES

Beginning Metal

Caroline Begbie

Notice of Rights

Notice of Liability

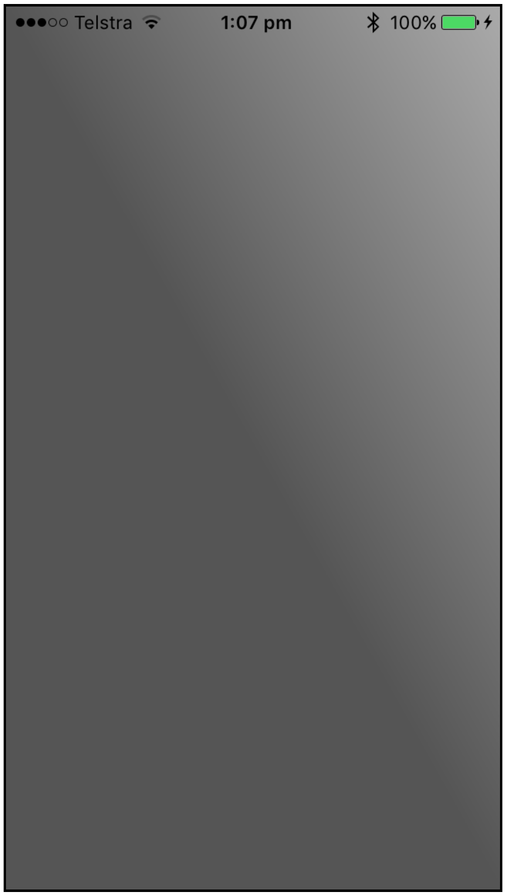
This challenge and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express of implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

Challenge #5: Shaders and Refactoring

By Caroline Begbie

You have two challenges now. Your first challenge is to make the quad grayscale.



This is a very simple challenge where you just convert the interpolated color fragments to gray.

There are better ways of converting to grayscale, but we'll just keep it simple here and average the color.

In **Shader.metal**, in the fragment function, take the RGB values from the fragment's vertex in color and divide by three:

```
float grayColor = (vertexIn.color.r +
                   vertexIn.color.g +
                   vertexIn.color.b) / 3;
```

And return the color value using this averaged gray color:

```
return half4(grayColor, grayColor, grayColor, 1);
```

Build and run, and all the colors on your quad should be converted to grayscale as in the image on the previous page.

This demonstrates how simple it is to manipulate the color of the fragment. Later on when we're covering lighting, we'll be taking the fragment color and working out how bright the fragment should be by multiplying the color by lighting variables.

Your second challenge is to refactor the pipeline state so that it has a closer relationship with the `Plane` object.

First create a new Swift file called **Renderable.swift**. All renderable objects, such as the `Plane`, and later other model types, will conform to `Renderable`.

Replace the contents of the file with:

```
import MetalKit

protocol Renderable {
}
```

`Renderable` objects will need a pipeline state property, a vertex function name, a fragment function name and a vertex descriptor. Add these properties to the protocol:

```
var pipelineState: MTLRenderPipelineState! { get set }
var vertexFunctionName: String { get }
var fragmentFunctionName: String { get }
var vertexDescriptor: MTLVertexDescriptor { get }
```

Add an extension to `Plane` to conform to `Renderable`:

```
extension Plane: Renderable {
}
```

Add the `Renderable` properties to the class:

```
// Renderable
var pipelineState: MTLRenderPipelineState!
var fragmentFunctionName: String = "fragment_shader"
var vertexFunctionName: String = "vertex_shader"
```

These are the same as the properties in `Renderer`, and we'll remove those from `Renderer` soon.

Copy the vertex descriptor code from `buildPipelineState()` in `Renderer` and create the vertex descriptor property using this code:

```
var vertexDescriptor: MTLVertexDescriptor {
  let vertexDescriptor = MTLVertexDescriptor()

  vertexDescriptor.attributes[0].format = .float3
  vertexDescriptor.attributes[0].offset = 0
  vertexDescriptor.attributes[0].bufferIndex = 0

  vertexDescriptor.attributes[1].format = .float4
  vertexDescriptor.attributes[1].offset = MemoryLayout<float3>.stride
  vertexDescriptor.attributes[1].bufferIndex = 0

  vertexDescriptor.layouts[0].stride = MemoryLayout<Vertex>.stride
  return vertexDescriptor
}
```

You now have a vertex descriptor for `Plane`. Other model types later may have a different vertex descriptor.

In `Renderable`, create an extension for the protocol:

```
extension Renderable {
}
```

Copy `buildPipelineState()` from `Renderer` to this `Renderable` protocol extension and change it so it looks like the following code:

```
func buildPipelineState(device: MTLDevice) -> MTLRenderPipelineState {
  let library = device.newDefaultLibrary()
  let vertexFunction =
            library?.makeFunction(name: vertexFunctionName)
  let fragmentFunction =
             library?.makeFunction(name: fragmentFunctionName)

  let pipelineDescriptor = MTLRenderPipelineDescriptor()
  pipelineDescriptor.vertexFunction = vertexFunction
  pipelineDescriptor.fragmentFunction = fragmentFunction
  pipelineDescriptor.colorAttachments[0].pixelFormat = .bgra8Unorm
  pipelineDescriptor.vertexDescriptor = vertexDescriptor

  let pipelineState: MTLRenderPipelineState

  do {
    pipelineState = try
      device.makeRenderPipelineState(descriptor: pipelineDescriptor)
  } catch let error as NSError {
    fatalError("error: \(error.localizedDescription)")
  }
  return pipelineState
}
```

Here you're creating a default method that will take the vertex and fragment functions and vertex descriptor from the `Renderable` object. Instead of being in `Renderer`, which renders all objects, each `Renderable` object will be able to have different vertex and fragment functions and different vertex descriptors.

Now to fix up `Renderer`. In **Renderer.swift**, delete `buildPipelineState()` and remove the call from the initializer.

In **Plane.swift**, at the end of `init(device:)` call `Renderable`'s default method `buildPipelineState(device:)` to set up `Plane`'s pipeline state:

```
pipelineState = buildPipelineState(device: device)
```

In **Renderer.swift**, in `draw(in:)`, cut the line:

```
commandEncoder.setRenderPipelineState(pipelineState)
```

and paste it into `Plane`'s `render(commandEncoder:deltaTime:)` before setting the command encoder's vertex buffer.

```
commandEncoder.setRenderPipelineState(pipelineState)
```

Back in `Renderer`, remove all references to `pipelineState`. Remove the `pipelineState` property and the `guard` statement in `draw(in:)`.

Build and run, and your app should still render the grayscale quad.

●●●○○ Telstra
1:07 pm
100%

However, now you've abstracted away the pipeline from the renderer. Currently you only have a `Plane` to render, but later on, you'll have different objects that will require different rendering pipelines. These objects will use different vertex and fragment functions, and the vertices will have different layouts too.