Metal® Programming Guide

Tutorial and Reference via Swift

Janie Clayton

Metal®
Programming Guide

```
                              lightIntensity: intensity,
                              projectionMatrix: projectionMatrix,
                              modelViewMatrix: modelViewMatrix)

let uniforms = [uniform]
uniformBuffer = device.makeBuffer(
                              length: MemoryLayout<Uniforms>.size,
                              options: [])
memcpy(uniformBuffer.contents(), uniforms,
             MemoryLayout<Uniforms>.size)
```

Over in the shader program, as you did for the position data, you need to create a struct that correlates to the data types in the Swift struct:

```
struct Uniforms
{
   float4 lightPosition;
   float4 color;
   packed_float3 reflectivity;
   packed_float3 intensity;
   float4x4 modelViewMatrix;
   float4x4 projectionMatrix;
};
```

Finally, you add this buffer as an argument to the fragment shader:

```
vertex VertexOut lightingVertex(
                 VertexIn vertexIn [[stage_in]],
                 constant Uniforms &uniforms [[buffer(1)]])
{
}
```

All the components you need to create your shader are in place. It's time to set up and program the GPU.

Vertex Shader

Earlier, you saw the equation for the intensity of the light source for each vertex. For convenience, it's repeated here:

$$L = K_d L_d s \cdot n$$

It's straightforward enough, but you need to do a bit of work to determine the direction of the surface point to the light source (the *s* value). You need to determine the eye coordinates using the projection matrix and the current vertex normal being computed. The result of this operation is then subtracted from the passed-in light position.

Here is your finished vertex shader:

```
vertex VertexOut lightingVertex(
                 VertexIn vertexIn [[stage_in]],
                 constant Uniforms &uniforms [[buffer(1)]])
{
   VertexOut outVertex;

   // Lighting code
   float3 tnorm = normalize(uniforms.projectionMatrix *
                      vertexIn.normal);
   float4 eyeCoords =
       uniforms.modelViewMatrix * float4(vertexIn.position, 1.0);
   float3 s = normalize(float3(uniforms.lightPosition -
                                  eyeCoords));
   outVertex.lightIntensity =
       uniforms.intensity * uniforms.reflectivity *
                                max( dot(s, tnorm),
       0.0);
   outVertex.position =
       uniforms.modelViewMatrix * float4(vertexIn.position, 1.0);

    return outVertex;
};
```

Fragment Shader

The only responsibility of the fragment shader is to determine the color at each pixel. This is determined by multiplying the color of the teapot by the light intensity at each pixel.

```
fragment half4 lightingFragment(
                      VertexOut inFrag [[stage_in]],
                      constant Uniforms &uniforms [[buffer(1)]])
{
    return half4(inFrag.lightIntensity * uniforms.color);
};
```

Generally, you want to make your fragment shader as small as possible. It gets called much more frequently than the vertex shader, so the more work you have in the fragment shader, the longer your render time will be.

In some cases, such as photorealistic lighting, you want to move work to the fragment shader, because it provides a much better representation of how light behaves in the real world. This is covered in Chapter 9.

Summary

Shaders are simple programs written in the MSL. MSL is based on C++14 and contains specialized methods and data types commonly used in graphics mathematics, such as vectors and dot products.

Shader programs are like icebergs. The code you see in the shaders is just the tip, obscuring a lot of the scaffolding and support built into the program through the buffers.

All graphics shaders must include both a vertex and a fragment shader. The output of the vertex shader is sent to the rasterizer. The output of the rasterizer is sent to the fragment shader.

All data used by the shaders must be fed into it in the form of buffers. Buffers are encoded with uniform and position data. They are added to the argument table, which is the way the shader and the main program can pass data back and forth. The shader also requires data structures that correlate to the data in the buffers so the shaders can decode which bytes correlate to which arguments.

6 Metal Resources and Memory Management

Time is the scarcest resource and unless it is managed nothing else can be managed.
—Peter Drucker

The purpose of Metal, in a nutshell, is to prepare data to be processed by the GPU. It doesn't have to be vertex position data—it could be image data, fluid dynamic information, and so on. In Chapter 5, "Introduction to Shaders," you explored how to program the GPU to process this data. In this chapter, you learn the ways to prepare blocks of data to be processed by the GPU.

Introduction to Resources in Metal

Resources in Metal are represented by instances of `MTLResource`. The `MTLResource` protocol defines the interface for any resource object that represents an allocation of memory. Resources fall into one of two categories:

- **`MTLBuffer`**: An allocation of unformatted memory that can contain any type of data
- **`MTLTexture`**: An allocation of formatted image data with a specified texture type and pixel format

Instances of `MTLResource` are responsible for managing access to and permission to change data that is queried by the GPU. These tasks are more specialized depending on whether you are working with buffers or textures. References to these blocks of data are added to an argument table so that it can be managed and synchronized between the CPU and GPU to ensure data is not modified as it is being read.

This chapter clarifies a few concepts that you have worked with already but might not have a full grasp on. Understanding how to format data to send to the GPU is the essence of Metal. Having a firm grasp of this process is essential and will help you on your journey.

The Argument Table: Mapping between Shader Parameters and Resources

One of the most important structures in Metal programming that you need to understand is the *argument table*. The argument table is the liaison between the encoder and the shaders. It is the shared space in memory where the CPU and the GPU coordinate which data is referred to and modified by the shaders.

Everything in the argument table is an instance of `MTLArgument`. It contains the argument's data type, access restrictions, and associated resource type. You do not create these arguments directly. They are created when you create and bind the arguments to the `MTLRenderPipelineState` or `MTLComputePipelineState`. Three types of resources can be assigned to the argument table:

- Buffers
- Textures
- Sampler states

Sampler states are discussed in Chapter 12, "Texturing and Sampling."
This chapter focuses on the first two resource types. At the time of writing, you can have, at most, 31 different buffers and 31 different textures on iOS devices. They are subject to change based on the GPU, so be sure to check the