Metal: An Introduction

William Woody
Glenview Software
woody@alumni.caltech.edu

Introduction

Metal is Apple's API for creating computer graphics on the Macintosh, iOS and tvOS platforms.

Metal provides support for GPU-accelerated 3D graphics as well as GPU computation workloads. Metal is a *low level API*, meaning Metal provides a low overhead API for creating graphics and computational workloads but requires significantly greater effort to use. Low level APIs are very flexible but notorious for being extremely hard to use and harder to learn.

The purpose of this document is to both introduce Metal, and dive into different aspects of Metal so you can create very complex graphics and compute-bound applications using Metal.

> *Note that because different API entry points can have a tremendous amount of complexity and different options when using them, this book is not necessarily intended to be read in order. Links are provided for more information, and for greater depth in discussing those API endpoints. Because of that, this book should be read as a PDF file on a computer with an Internet connection.*
>
> *Note also that I'm trying to be intentionally terse in my descriptions, in order to get to the heart of the matter as quickly as I can.*
>
> *This document is not an introduction to Computer Graphics, and assumes a degree of knowledge about computer graphics and some familiarity with OpenGL or Direct X.*

Table of Contents

Translation

Scale

Rotate

Perspective

The Metal Execution Model

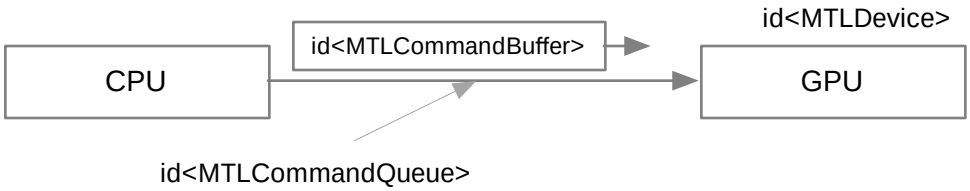
Computers which have hardware-supported 3D rendering generally provide this functionality using a Graphics Processing Unit. A graphics processing unit is a separate microcomputer which provides specialized functionality for performing massively parallel calculations, such as those used when accelerating 3D computer graphics.

One key thing about the GPU is that you can write custom software that runs on the GPU separate from the CPU. These "shader functions" permit you to write programs for performing complex calculations on the GPU for various visual effects, and for computational tasks. Shaders are discussed in greater detail in a later chapter.

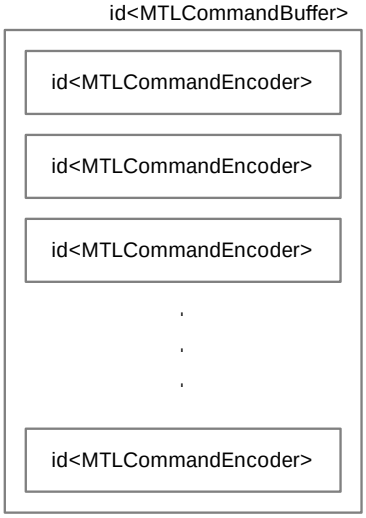
The Metal Framework provides a method for communicating requests to a separate GPU, such as the one built into the iPhone or the graphics card built into the Macintosh.

This involves APIs for obtaining a representation of a GPU, for copying data to a GPU (such as texture map data or geometry data), and for sending commands to use that data for performing computations and for rendering graphics.

The execution model used by Metal is that of a device which a buffer full of commands is sent.



Each buffer of commands consists of one or more encoded collections of commands.



Commands are then executed asynchronously on the GPU. The Metal API contains a number of methods for receiving callbacks when a command finishes, as well as methods for copying data to the GPU. With

this, you could construct a model of a planet and space ship--and during rendering simply update the relative position of the two bodies in order to reduce the amount of data that has to be sent back and forth between the CPU and the GPU.

Because the types of commands sent in a single command buffer can be mixed, you can even construct a simulation that runs entirely on the GPU using a compute kernel, and display the results of the simulation graphically using a 3D rendering shader--all without involving the CPU in the calculations.

The Use of Protocols

The Apple Metal API makes extensive use of Objective C Protocols in order to hide the implementation of the underlying Metal objects. This implies that obtaining and using certain objects within Metal follows a well-defined pattern of using a protocol (generally MTLDevice, though not always) to obtain new instances of objects.

Further, some objects are expensive to obtain or initialize, so they are expected to be set up on the initialization of your application, and reused whenever possible. This implies the initialization code of your application can become extremely complex as you set up the GPU for rendering or for performing computational tasks.

The Device

A GPU is represented by the MTLDevice protocol. This protocol provides interfaces for copying data to the device, for compiling or obtaining shader functions on a GPU, and for submitting requests.

Obtaining a Device

The easiest way to obtain a device is by using MTLCreateSystemDefaultDevice. On the Macintosh (where multiple GPUs may be installed), there are techniques for device selection which are more efficient that are not covered in this document.

The Command Queue

The MTLCommandQueue protocol represents the communications queue for sending commands to the GPU. This is obtained from the MTLDevice protocol, and should be obtained once and reused as required.

The Command Buffer

The MTLCommandBuffer protocol represents a collection of commands to be sent to the GPU and executed as a single unit. This is obtained from the MTLCommandQueue protocol.

Generally when building a 3D application, you would create one MTLCommandBuffer to represent the commands sent to the GPU to render a single frame. Thus, you would create a command buffer each time you render a frame--updating the command parameters necessary to show movement on the screen.

Once all of the commands have been encoded in this buffer, you can optionally add handlers that are called back when the buffer is scheduled for execution or when it completes, to specify the drawable in which the graphics will be presented (if the commands represent a graphics operation), and finally to commit the commands to the GPU for execution.

The Command Encoder

The MTLCommandEncoder protocol represents the parent protocol of a collection of protocols used to encode commands in a command buffer. You would obtain an encoder from the MTLCommandBuffer, using the appropriate method depending on the types of commands to be encoded.

A command buffer may contain one or more sets of commands encoded with different encoders, and the commands from each encoder is executed in the order they are created in a command buffer.

Encoders are created one at a time. Thus, if you have multiple encoders (for example, a compute command which runs one step of a simulation, followed by a render command which renders the results on a display), the compute command encoder in our example is guaranteed to run first.

Multiple encoders can be used for creating complex rendering effects, such as performing shadow mapping by encoding the two separate rendering passes required as two separate rendering commands. Multiple rendering passes can also be used for reflection mapping and deferred shading.

Rendering Command Encoder

The most commonly used command sent to a GPU is rendering graphics, and this is encoded using the MTLRenderCommandEncoder. The rendering encoder is obtained from the MTLCommandBuffer by first constructing an MTLRenderPassDescriptor class and setting the parameters that will be used by that rendering pass--such as the destination screen or texture that the graphical results will be rendered to.

You would then set the MTLRenderPipelineState representing the rendering pipeline that will be used to render graphics on the display. Rendering pipelines are expensive objects to create and are generally built in advance during initialization. Multiple pipelines can be used in a single command encoder to specify different rendering effects for different objects. Objects drawn with the drawing commands in a render command encoder are drawn with the last set pipeline state.

You would set up the different texture references, buffer references and other parameters used for rendering objects.

Then you would issue the appropriate drawing commands to draw the primitives (whose geometry was set with the buffer references above).

And finally you would finish encoding by using the endEncoding method.

Compute Command Encoder

The compute command encoder is used to send a compute request to the GPU, and this is encoded using the MTLComputeCommandEncoder. This follows a similar pattern as the rendering command encoder

above: a compute encoder is obtained from the MTLCommandBuffer, and a MTLComputePipelineState representing the compute pipeline (with a reference to the kernel function on the GPU to execute) is associated with it.

A compute pipeline has a number of options associated with it involving the number of GPU threads that are associated with the task, as well as the resources associated with it.

Like the render command encoder, when you are finished encoding the commands (including specifying the buffers passed to the command buffer for use during execution), you would finish encoding with the endEncoding method.

Memory Management Tasks

Memory management tasks (such as copying data to the GPU) can be encoded using the MTLBlitCommandEncoder. Generally data such as textures or models or transformation matrices used to calculate the position of 3D data is handled by manipulating the data buffers directly, but this command also provides you the ability to handle certain tasks, such as blurring of images, where other techniques may not work as efficiently.

Parallel Rendering

The MTLParallelRenderCommandEncoder allows for encoding multiple multiple rendering passes in parallel, and can be used to create MTLRenderCommandEncoders for rendering passes which operate simultaneously. This can be used to gain greater throughput for complex rendering tasks.

This document does not discuss either the MTLBlitCommandEncoder or the MTLParallelRenderCommandEncoder protocols.

Building a Basic MacOS Metal Application

This section discusses the steps for putting together a basic MacOS metal application. The first section will describe how to put together just enough of an application to get a blank screen. The second section will discuss adding geometry to render a triangle. The third will discuss drawing a more complex shape, and the fourth will discuss texture maps.

Each of these sections are intended to provide a very simple introduction to some of the basic features of Metal, and should help you understand the role of the different components in doing basic graphics operations.

All source code for the examples below can be downloaded from GitHub. You are also free to use the sources on GitHub in your own projects.

Starting a New Metal Application

The goal of this section is to introduce the basic metal view class and delegate, obtaining a device, initializing a command queue and buffer, and enqueuing the buffer to the GPU to render a blank screen.

This may not sound like a lot, but it does illustrate the basic moving parts of a Metal application.

Creating the View

Currently the proper approach for building MacOS, iOS and tvOS applications which use metal is to use the MTKView class as the destination view. The MTKView class uses the MTKViewDelegate class to actually handle rendering.

> *Note: As with all applications which use a delegate pattern, you have three choices in developing your application.*
>
> *You can create a separate delegate class object (inheriting from NSObject) which contains the rendering methods for your application. You would then create an instance of your class in the NSViewController/UIViewController of your class, and add it as a delegate to the MTKView class.*
>
> *You can make your NSViewController/UIViewController conform to the MTKViewDelegate protocol, and set the view controller as the delegate to the MTKView class.*
>
> *Or you can inherit from the MTKView class and make it conform to the MTKViewDelegate protocol, and have the view class set itself as its own delegate.*
>
> *In our sample code we use the third option. There is nothing special about any of these options and they have their own advantages and disadvantages.*
>
> *Apple's "Hello Triangle" example gives a good example of the first option of a separate rendering class.*

In our sample code we set the root view of our view controller inside our *Main.storyboard* file to be an instance of our MXView class, which inherits from MTKView and which conforms to the MTKViewProtocol. Thus, when our application starts up, the view controller and view are assembled for us.

You can use other techniques for constructing the rendering view in a UIView or NSView object by using the CAMetalLayer class.

Initializing our MTKView.

Before we can use our MTKView to render a blank screen, we need to initialize a few things first. The code for initializing our view is contained in the *internalInit* method, which is invoked by both the *initWithFrame:* and *initWithCoder:* methods.

Obtaining the GPU Device

The MTKView by default does not have a device associated with it, and during initialization you must obtain the GPU device to use for rendering.

Within our source code the device is obtained with the line

```
self.device = MTLCreateSystemDefaultDevice();
```

Setting Other MTKView Parameters

There are a number of other parameters that can be set with the MTKView class, some of which are required. For our application we set both the pixel format of the texture that represents our display, and the default color to clear the display to, with the calls:

```
self.colorPixelFormat = MTLPixelFormatBGRA8Unorm;
self.clearColor = MTLClearColorMake(0.1, 0.1, 0.2, 1.0);
```

There are a number of other optional parameters that may be set to cause the MTKView class to behave differently, including settings which control frame rate, and if the display should be redrawn continuously on a timer loop or only when the display has been invalidated.

Obtaining the Command Queue

The command queue is directly obtained from the device and stored in its own class property for reuse.

```
self.commandQueue = [self.device newCommandQueue];
```

Implementing the MTKViewDelegate Methods

There are two methods that must be implemented in the MTKView delegate.

- (void)mtkView:(MTKView *)view drawableSizeWillChange:(CGSize)size

This method is invoked by the MTKView when its size changes. You would use this method to handle tasks such as setting the transformation matrices of your rendering engine to correct for the changed aspect ratio, and potentially for reloading or resetting textures or other resources to handle the changed resolution of the screen.

For our first sample application, this method does nothing.

- (void)drawInMTKView:(MTKView *)view

This method is called each time you view needs to be redrawn. The parameter passed is the view that needs to be redrawn. (Since our sample code inherits from the MTKView class, the *view* parameter will be the same as *self*.)

For our drawing method we will need to carry out the following steps in order to request that our screen be filled with a background color.

Obtaining a Command Buffer.

First, we obtain a command buffer for constructing the rendering commands to render the screen. Because the command buffer renders the screen once, if we were creating an animation we would effectively create a new buffer for every frame of our animation.

We do this with the call

```
id<MTLCommandBuffer> buffer = [self.commandQueue commandBuffer];
```

Creating a Command Encoder

We next create a rendering command encoder for performing a single rendering pass to render our blank screen.

Note that if we were rendering a complex scene that requires multiple rendering passes, we could encode each rendering pass in its own separate MTLRenderCommandEncoder. But since we are doing a single rendering pass, we only need one MTLRenderCommandEncoder.

Remember to create a command encoder we must first create a MTKRenderPassDescriptor with the parameters for our encoder.[1] Because we are rendering to our display, we can use the convenience method in the MTKView class to obtain a default descriptor for rendering to our display.

```
MTLRenderPassDescriptor *descriptor = [view currentRenderPassDescriptor];
```

This is effectively a shortcut for the following:

---

[1] This pattern: of creating a descriptor then using the descriptor to generate an encoder or state, is a common design pattern in Metal.

```
id<CAMetalDrawable> drawable = self.currentDrawable;

MTLRenderPassDescriptor *descriptor = [MTLRenderPassDescriptor
renderPassDescriptor];
descriptor.colorAttachments[0].texture = drawable.texture;
descriptor.colorAttachments[0].loadAction = MTLLoadActionClear;
descriptor.colorAttachments[0].storeAction = MTLStoreActionStore;
descriptor.colorAttachments[0].clearColor = self.clearColor;
```

In the above snippet of code, we first obtain the drawable (which is vended[2] by the method), then we use it as the destination texture during rendering in our rendering pass. We also specify that the results of the rendering command be stored in our screen 'texture', that it should be cleared before drawing into it, and it should be cleared to the provided color.

> *Note: For a system that requires multiple rendering passes, such as with reflection, generally we would create a separate writeable texture map to render into in the first pass, then use the results in that texture map in the second pass for our reflection special effect.*

We then use the descriptor to generate our rendering encoder:

```
id<MTLRenderCommandEncoder> encoder = [buffer
renderCommandEncoderWithDescriptor:
    descriptor];
```

Finishing Up.

Since we are not drawing anything (but just allowing the encoder to clear the screen to the background color) we can finish up our encoder by calling:

```
[encoder endEncoding];
```

If we were doing multiple rendering passes (say, for reflection), we would start constructing the next rendering pass here. But since we are not we can finish up our command buffer and submit it to the GPU for rendering.

To finish up we first indicate to the buffer that once our screen has been redrawn, it can be "presented" (that is, displayed on the screen) by calling:

```
[buffer presentDrawable:self.currentDrawable];
```

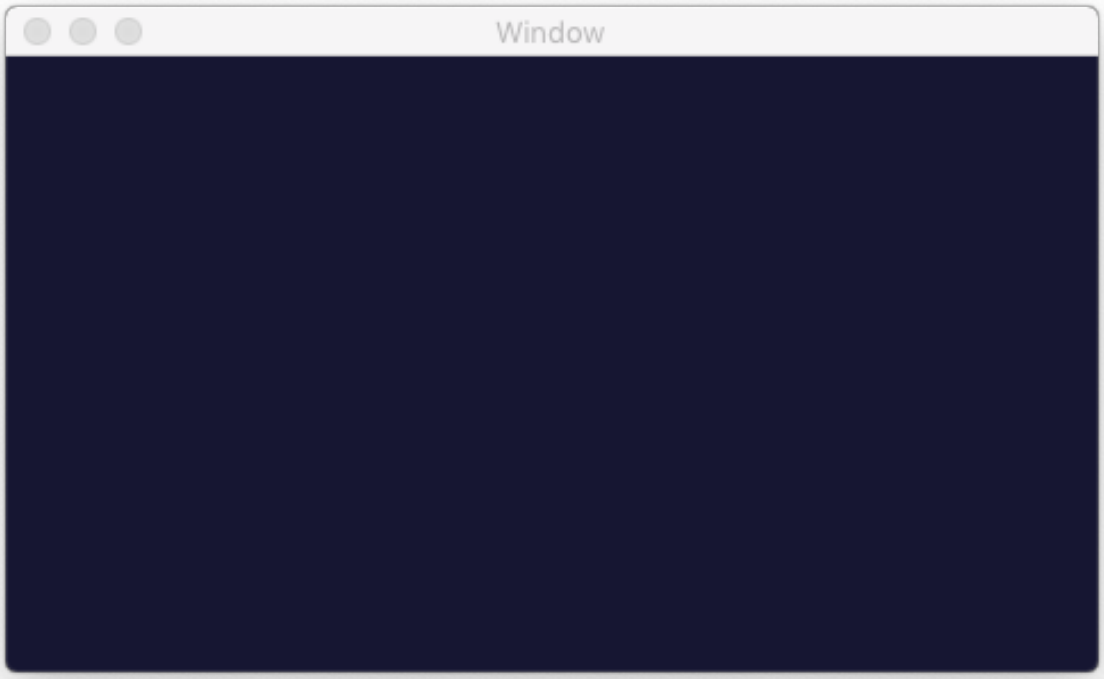
Once we've done this, we can submit the buffer of commands to the GPU for rendering by calling:

```
[buffer commit];
```

---

[2] That is, each time the method is called, a new object is created from a pool of objects. Thus, when you call this method, you should store the result in a local variable and reuse the same object (rather than calling *currentDrawable* repeatedly) for the duration of your method call.

Once all this is done, when we run our finished application (which can be downloaded from GitHub) we should see... well...



It's not much, granted. But it does show our code works.

Adding Support To Draw A Triangle.

The next obvious step is to draw something. In order to draw something we will need to add support for a rendering shader (that is, a small program that actually runs on the GPU), for setting up a rendering pipeline (so we know which code snippets on the GPU to execute), for creating a buffer (so we can send geometry data to the GPU), and for adding rendering commands to our render command encoder.

A Word About Shaders

Shaders are discussed in greater detail later in this document. Shaders, however, are basically small routines written in a modified version of C++, which are compiled by Xcode and which are automatically loaded into the GPU for you when your application starts up. Most of the interesting stuff that can make for visually impressive displays for video games comes from a deep understanding of how shaders work.

In the next few examples we will make very basic use of shaders, and try to introduce a few very basic concepts along the way.

A Word About Our Initialization Routine

Our prior example put all of our initialization code inside a single *internalInit* method. However, this can be poor practice in a real-world example, simply because the amount of code necessary to initialize our drawing routines can grow to be extraordinarily large. In fact, because of the nature of GPU rendering (where for performance we want to put as much stuff in the GPU as we reasonably can), for many programs our initialization routine may be the largest block of routines in our entire program.

Thus, we will get into the practice of refactoring our internal initialization into smaller chunks of more manageable code.

Creating Our Geometry In A Buffer

As noted above, before we can render any geometry we must somehow get the geometry to our GPU. For our triangle we create an MTLBuffer object with the raw data that makes up the 3 corners of our triangle. Because geometry is potentially very expensive we create our triangle as part of our *internalInit* method call.

Because we are both laying out the memory of our geometry and using it in the GPU, we must make sure the format of the data stored in memory is known to the GPU. The way we handle this is by declaring a new class *MXVertex* which represents the data in our triangle. We then initialize our triangle manually:

```
static const MXVertex triangle[] =
{
    // Homogeneous pts , RGBA colors
    { {  1,  -1, 0, 1 }, { 1, 0, 0, 1 } },
    { { -1,  -1, 0, 1 }, { 0, 1, 0, 1 } },
    { {  0,   1, 0, 1 }, { 0, 0, 1, 1 } },
};
```

The location of our points are given in homogeneous coordinates, which we use for handling complex transformations, such as rotation and scaling and perspective.

Once we've constructed the block of memory with our vertices, we create the MTLBuffer with the following call:

```
self.triangle = [self.device newBufferWithBytes:triangle
                                         length:sizeof(triangle)
                  options:MTLResourceOptionCPUCacheModeDefault];
```

This will take the block of memory and make it available to the GPU.

Because creating geometry is expensive we do this at initialization time. If you were writing an application that modified geometry (such as a 3D CAD program) you would cache the data in an MTLBuffer rather than trying to generate the MTLBuffer on the fly each time you render the display.

Creating Our Pipeline

Next we create our pipeline. This can be far more complicated in that we need to create our shader functions used by our pipeline to control how we will display our triangle on the GPU, obtain our shaders, and configure our pipeline to pass our geometry to our shaders for rendering.

This is also generally done at initialization time as all of the operations are quite expensive.

Creating the Shader Functions

First we need to create our shader functions.

We have two shader functions for a rendering pass. The first is the vertex shader function; the vertex shader handles the transformation of the location of each vertex in our geometry so it is displayed in the correct position on the screen.

The second is the fragment shader function; it handles computing the color of each pixel on the screen based on information passed to it from the vertex shader.

For this we need to create a .metal file; the file is compiled by Xcode and stored as a resource that is then loaded into our GPU when we start up our Metal application.

For our vertex and fragment shaders we need to set up two structures: one declaring the input vertex, and one which declares the vertex output:

```
struct VertexIn {
    float4 position  [[attribute(MXAttributeIndexPosition)]];
    float4 color     [[attribute(MXAttributeIndexColor)]];
};

struct VertexOut
{
   float4 position [[position]];
   float4 color;
};
```

Notice they are more or less the same thing. However, the first declaration indicates the attribute indexes (in the [[attribute(N)]] declarations) that will also be used when setting up our vertex attributes for our pipeline, and the second establishes (in the [[position]] declaration) which stores the coordinates for extrapolation of vertices into fragments for rendering.

In practice the two would be very different, depending on the type of rendering we're performing.

Our vertex shader is declared like a C function call, but with extra attributes associated with the parameters indicating where they come from:

```
vertex VertexOut vertex_main(VertexIn v [[stage_in]])
{
   VertexOut out;
```

```
    out.position = v.position;
    out.color = v.color;

    return out;
}
```

Our vertex shader basically copies the geometry to the output without modification. When we start doing 3D rendering, we will perform other math operations that update the output position based on a transformation matrix--but that's for later.

Our fragment shader is even easier. The fragment shader is called on each visible pixel found in the triangle whose vertices were transformed by our vertex shader--with values in VertexOut extrapolated across the surface of the triangle. So our fragment shader is pretty simple:

```
fragment float4 fragment_main(VertexOut v [[stage_in]])
{
    return v.color;
}
```

Loading the Library

Once we've created the .metal file, we can access information about our GPU's compiled graphic or compute functions by using the MTLLibrary protocol. We do this during initialization of our pipeline, storing a reference to our library away for later use.

We obtain our library from our device by writing:

```
self.library = [self.device newDefaultLibrary];
```

Obtaining References to Our Shader Functions

We now can obtain references to our two shader functions by name from our library, storing them in two MTLFunction objects.

```
self.vertexFunction = [self.library newFunctionWithName:@"vertex_main"];
self.fragmentFunction = [self.library newFunctionWithName:
                                                    @"fragment_main"];
```

Constructing the Vertex Descriptor

The vertex descriptor object MTLVertexDescriptor is used to describe the layout of the memory we allocated when we constructed our geometry above. This is used to help tie the memory format and layout of our triangle's vertices to the attribute positions in our vertex shader function.

In each attribute of our MTLVertexDescriptor we indicate the type of the attribute (such as a vertex made of four floating-point numbers), the offset of the attribute in memory, and the size of the attribute in memory.

We also indicate the 'stride'; that is, the size of each vertex in memory.

```
MTLVertexDescriptor *d = [[MTLVertexDescriptor alloc] init];
d.attributes[MXAttributeIndexPosition].format = MTLVertexFormatFloat4;
d.attributes[MXAttributeIndexPosition].offset = 0;
d.attributes[MXAttributeIndexPosition].bufferIndex = 0;
d.attributes[MXAttributeIndexColor].format = MTLVertexFormatFloat4;
d.attributes[MXAttributeIndexColor].offset = sizeof(vector_float4);
d.attributes[MXAttributeIndexColor].bufferIndex = 0;
d.layouts[0].stride = sizeof(MXVertex);
```

Constructing the Pipeline Descriptor

Now that we have all of the pieces used by our pipeline descriptor, we can now build our pipeline descriptor MTLRenderPipelineDescriptor.

```
MTLRenderPipelineDescriptor *pipelineDescriptor =
[MTLRenderPipelineDescriptor new];
pipelineDescriptor.vertexFunction = self.vertexFunction;
pipelineDescriptor.fragmentFunction = self.fragmentFunction;
pipelineDescriptor.colorAttachments[0].pixelFormat =
self.colorPixelFormat;
pipelineDescriptor.vertexDescriptor = d;
```

Building the Pipeline State Object

Once we have the descriptor we build the MTLRenderPipelineState object for use in our system. We store this as part of our class for reuse when we actually want to render some graphics.

```
self.pipeline = [self.device newRenderPipelineStateWithDescriptor:
                                                        pipelineDescriptor
                                                            error:nil];
```

Rendering Our Triangle

Now that we've constructed a pipeline state, we can use it with our MTLRenderCommandEncoder to actually render our triangle.

Setting our Render Pipeline State

After we have constructed our MTLRenderCommandEncoder we can set the pipeline by calling:

```
[encoder setRenderPipelineState:self.pipeline];
```

Note that the pipeline is then used on the drawing commands that follow. If we then were to call setRenderPipelineState: with a new pipeline, that new pipeline would apply to subsequent drawing calls. In this way the same rendering pass can use multiple pipelines to construct the image.

Setting the Vertex Buffer With Our Triangle Geometry

We now pass to the encoder the vertex geometry we will render.

```
[encoder setVertexBuffer:self.triangle
                  offset:0
                 atIndex:MXVertexIndexVertices];
```

> *Note: Because our vertex shader uses the [[stage_in]] attribute to specify our vertices, the vertex buffer containing our geometry* ***must be at index 0.*** *The [[stage_in]] parameter implies our geometry will come from index 0.*
>
> *(TODO: Verify this is correct, and that the buffer 0 index is not implicit by the position in the parameter list. Verify we can't mix stage_in and buffer:N attributes.)*

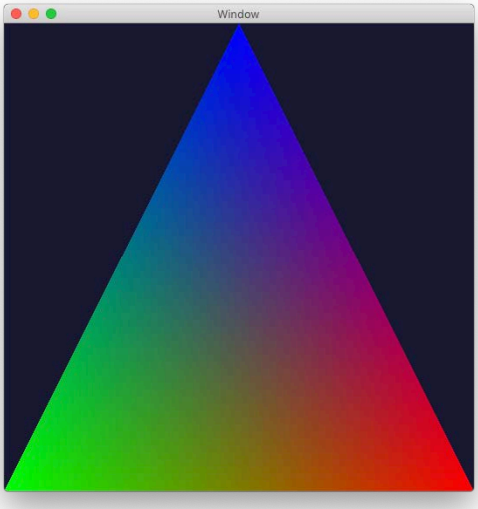
Drawing Our Triangle

We can now draw our triangle by calling the appropriate drawing routine with our encoder:

```
[encoder drawPrimitives:MTLPrimitiveTypeTriangle
            vertexStart:0
            vertexCount:3];
```

Once all these changes have been made, when we compile and run our application (which can be downloaded from GitHub), we should see:

Okay, so it's a little more than what we had the last time. But now we've gotten shaders working, we can send geometry to our GPU, we can send geometry to our GPU for rendering.



Drawing Our Triangle in 3D

Next we'd like to draw our triangle in 3D.

The problem with a low level API like Metal is that it does not provide a number of tools for 3D graphics, but expects you to provide them yourself, either by writing the code yourself or by incorporating a third party library.

So in this section we will add support for 3D by writing our own 3D transformation system, and incorporate 3D transformations into our vertex shader by sending updated transformation matrices to the vertex shader to animate a rotating triangle.

3D Transformations

Homogeneous coordinate systems and transformations used in computer graphics is an entire topic that isn't covered here. There is a brief introduction to the subject later in this document.

Creating the Transformation Class

The MXTransformationStack class provides a complete implementation of transformation matrices commonly used in computer graphics (for translation, scaling, rotation and perspective), and also provides a mechanism for pushing and popping transformation matrices on a stack so that elements can be re-used when rendering a model.

> *Note: The ability to push and pop transformations is useful when animating an articulated figure. For example, the overall body transformation may be constructed first, then pushed on a stack. The additional transformations for positioning and moving an arm is constructed and used, then the body transformation is popped so the next arm's transformation can be constructed, and so forth.*
>
> *By doing this a hierarchy of transformations, corresponding to the hierarchy of relationships between components of a model, can be constructed and managed relatively easily.*
>
> *Our transformation stack operates with the idea that the transformations are constructed back to front: that is, the last transformation (such as positioning the body in the world scene) is added first, then the next transformation (such as positioning the arm relative to the body) is added next, and so forth.*

Adding Transformations to Our MXView Class

Each MXTransformationStack object represents a single transformation matrix which accumulates multiple translations, rotations and scaling operations into a single 4x4 matrix of numbers. We use two such matrices to represent the full translation; one for the perspective transformation and one for the model. We do this for later, when we will use the model matrix for lighting calculations.

We also add constructors in a new *setupTransformation* method which is called from our initializer.

Updating -mtkView:drawableSizeWillChange:

The method in the MTKViewDelegate protocol is called when the size of the screen changes. We can use this to set the proper aspect ratio of our display by resetting and initializing the view perspective. When the delegate method is called, we reset the view matrix and set the perspective matrix with the correct aspect ratio:

```
[self.view clear];
[self.view perspective:M_PI/3
                aspect:size.width/size.height
                  near:0.1
                   far:1000];
```

Updating the model matrix to rotate as we redraw the display.

We update the model matrix each time we pass through the draw method, with the current elapsed time since the starting of our application. We then translate our triangle away from the eye position, and slowly rotate our triangle by the amount of elapsed time.

```
double elapsed = CACurrentMediaTime() - self.startTime;
[self.model clear];
[self.model translateByX:0 y:0 z:-2];
[self.model rotateAroundAxis:(vector_float3){ 0, 1, 0 } byAngle:elapsed];
[self.model scaleBy:2];
```

Updating the Vertex Shader

Now that we have the transformation matrices which represent how we alter the 3D geometry on our display, we must pass the transformation matrices to the GPU so our vertex shader can update the location of our vertices.

To do this, we define the structure that we wish to pass to the GPU, then set the structure and copy it to the GPU via a MTLBuffer. On the GPU side, we obtain a reference to the buffer and use the data passed to us to update the vertices.

Declare the Uniforms Structure

We declare the structure format in the shader types header. That way the same declaration is used on both sides: the Metal side and the Objective-C side. We can do this because we do not need to use any attribute parameters for the Metal declarations that the Objective-C compiler does not understand.

```
typedef struct MXUniforms
{
   matrix_float4x4 model;
   matrix_float4x4 view;
} MXUniforms;
```

Update the Vertex Shader Function

We need to update our shader function to use the uniforms data that will be passed in. We're passing in the parameters using an MTLBuffer, passed in using the *setVertexBytes:length:atIndex:* method call, which we then match up to the parameter in our vertex buffer using the [[buffer(n)]] attribute.

We then update our vertex location to the output vertex location through matrix/vector multiplication.

```
vertex VertexOut vertex_main(VertexIn v [[stage_in]],
                 constant MXUniforms &u [[buffer(MXVertexIndexUniforms)]])
{
   VertexOut out;

   float4 worldPosition = u.model * v.position;
   out.position = u.view * worldPosition;
   out.color = v.color;

   return out;
}
```

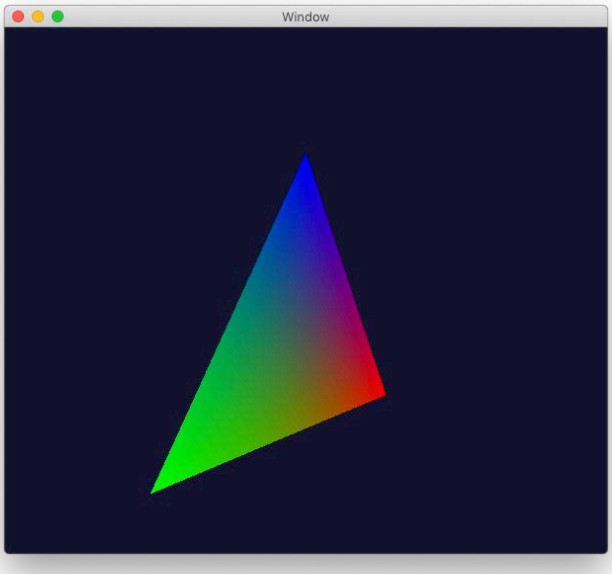
Passing in the Uniforms

In our drawing method we need to pass the uniform data into the buffer.

Right after recalculating the model CTM for our drawing, we pass the data into the vertex buffer through initializing a structure. We then copy the bytes to the GPU:

```
MXUniforms u;
u.view = self.view.ctm;
u.model = self.model.ctm;
[encoder setVertexBytes:&u
                 length:sizeof(MXUniforms)
                atIndex:MXVertexIndexUniforms];
```

Putting all of these together and running our application (which can be downloaded from GitHub), we now have a rotating triangle:



Loading More Complex Objects From a Resource

Now of course the triangle is interesting, and it's a useful model for if we wish to draw more complex objects constructed in memory, it's often useful to be able to load predefined objects from a resource. You may wish to do this when building a game, for example.

Resources are loaded using an MDLAsset, and we use the MDLVertexDescriptor to describe the vertex offsets in order to load our asset into memory into a known format.

The asset we will be loading is a variation of the Utah Teapot.

Because we have a complex object we're rendering, we also need to enable z-buffering so that the back side of the teapot does not draw in front of the front side.

Updating the Vertices for Our Model

First, we must rewrite the vertex descriptor by rewriting the code that generates our vertex descriptor. We also need to update the vertex structure on both the Metal and Objective-C sides to match.

The teapot.obj file contains a 3D location for each vertex, along with a 3D normal (an arrow pointing out from the center of the teapot at a right angle to the surface), and a texture coordinate so we can wrap our teapot with a texture.

Rewriting Our Structures

We first need to replace our MXVertex structure in Objective-C:

```
typedef struct MXVertex
{
   vector_float3 position;
   vector_float3 normal;
   vector_float2 texture;
} MXVertex;
```

And in Metal:

```
struct VertexIn {
    float3 position  [[attribute(MXAttributeIndexPosition)]];
    float3 normal    [[attribute(MXAttributeIndexNormal)]];
    float3 texture   [[attribute(MXAttributeIndexTexture)]];
};
```

Rewriting the Vertex Descriptor

Next we rewrite the vertex descriptor to match our vertex structure. This is done using the MDLVertexDescriptor class so we can also use it to load our object from a resource using the Model I/O API.

```
MDLVertexDescriptor *d = [[MDLVertexDescriptor alloc] init];

d.attributes[0] = [[MDLVertexAttribute alloc]
    initWithName:MDLVertexAttributePosition format:MDLVertexFormatFloat3
    offset:0 bufferIndex:0];

d.attributes[1] = [[MDLVertexAttribute alloc]
    initWithName:MDLVertexAttributeNormal format:MDLVertexFormatFloat3
    offset:sizeof(vector_float3) bufferIndex:0];

d.attributes[2] = [[MDLVertexAttribute alloc]
    initWithName:MDLVertexAttributeTextureCoordinate
    format:MDLVertexFormatFloat2 offset:sizeof(vector_float3) * 2
    bufferIndex:0];

d.layouts[0] = [[MDLVertexBufferLayout alloc]
    initWithStride:sizeof(MXVertex)];
```

Because we're using the Model I/O API, we'll also need to translate the vertex descriptor to a format useable by Metal.

```
pipelineDescriptor.vertexDescriptor =
    MTKMetalVertexDescriptorFromModelIO(d);
```

Loading the Model

We can now use Model I/O to load our teapot from a resource in our application.

First, we need the URL to the resource in our application's bundle:

```
NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"teapot"
        withExtension:@"obj"];
```

Next, we create a mesh buffer allocator that will be used to load our asset:

```
MTKMeshBufferAllocator *allocator = [[MTKMeshBufferAllocator alloc]
        initWithDevice:self.device];
```

And then we load our MDLAsset from our resource:

```
MDLAsset *asset = [[MDLAsset alloc] initWithURL:modelURL
        vertexDescriptor:d bufferAllocator:allocator];
```

And finally we convert the asset into an array of meshes which will be used to render the teapot. The resulting array of MTKMesh contains buffer data sent to the GPU for rapid rendering of our object.

```
self.teapot = [MTKMesh newMeshesFromAsset:asset device:self.device
        sourceMeshes:nil error:nil];
```

Updating the Vertex Shader Function

Because we've changed the vertices when loading our model we need to update our vertex shader function. For this example we're going to use the normal vector to populate our color to help us "see" our teapot. Later examples will update our shader to emulate lighting effects.

```
vertex VertexOut vertex_main(VertexIn v [[stage_in]],
                                        constant MXUniforms &u
[[buffer(MXVertexIndexUniforms)]])
{
   VertexOut out;

   float4 worldPosition = u.model * float4(v.position,1.0);
   out.position = u.view * worldPosition;
   out.color = float4(v.normal,1.0);

   return out;
}
```

Rendering the Model

Now that we have the model loaded in an array of meshes, we can render the array of meshes:

```
for (MTKMesh *mesh in self.teapot) {
    MTKMeshBuffer *vertexBuffer = [[mesh vertexBuffers] firstObject];
    [encoder setVertexBuffer:vertexBuffer.buffer
                      offset:vertexBuffer.offset
                     atIndex:0];

    for (MTKSubmesh *submesh in mesh.submeshes) {
        MTKMeshBuffer *indexBuffer = submesh.indexBuffer;
        [encoder drawIndexedPrimitives:submesh.primitiveType
```

```
                                    indexCount:submesh.indexCount
                                     indexType:submesh.indexType
                                   indexBuffer:indexBuffer.buffer
                             indexBufferOffset:indexBuffer.offset];
        }
    }
```

Hiding Stuff That Should Not Be Drawn.

Now if we were to simply stop here we would get all sorts of nasty visual effects. That's because we don't hide the back-side of the rendered image and elements that may be visually occluded by the front side of the teapot.

We need to set a couple of flags in order to prevent parts of the model which should not be visible from being drawn.

There are two things we can do to hide the back side of the model: culling back-facing polygons and enabling depth testing.

Back-Face Culling

Back face culling works by finding triangles that are "back-facing." The idea is that if we have all our triangles with a clock-wise winding, if after the transformations are performed the transformed matrices are counter-clockwise, then the polygon is facing away from us--and can be hidden before it is turned into an array of pixels.

```
[encoder setCullMode:MTLCullModeFront];
```

> *Note: The triangles in our object are stored in the reverse order expected by Metal, so we use the flag to cull front-facing triangles.*

Z-Buffer

Even with back-face culling when the teapot rotates so the handle or spout are behind the body of the teapot they are still visible. We can resolve this problem by using z-buffering, which keeps track of the depth associated with each pixel, overwriting pixels only if they are closer to the camera.

We first enable the depth stencil format in our view by calling setDepthStencilPixelFormat in the MTKView. This causes the MTKView class to generate the appropriate depth texture map.

```
self.depthStencilPixelFormat = MTLPixelFormatDepth32Float;
```

We also need to specify the depth format with the pipeline descriptor, so our graphics pipeline knows to use depth testing. This indicates to the pipeline the format of the depth buffer initialized by the MTKView:

```
pipelineDescriptor.depthAttachmentPixelFormat =
        self.depthStencilPixelFormat;
```

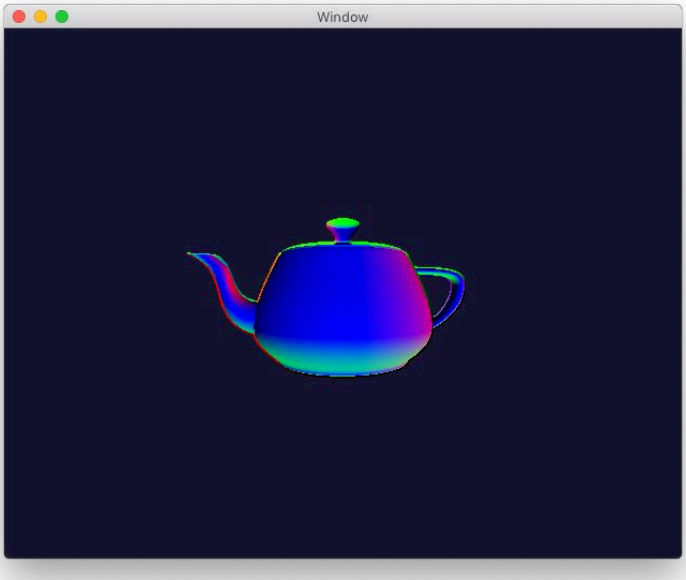
We now create a MTLDepthStencilState. This is used to indicate to the rendering pass encoder the way the depth buffer should be used: if the depth buffer is write enabled and the compare function used to determine if a pixel is closer.

```
MTLDepthStencilDescriptor *depthDescriptor = [[MTLDepthStencilDescriptor
        alloc] init];
depthDescriptor.depthCompareFunction = MTLCompareFunctionLess;
[depthDescriptor setDepthWriteEnabled:YES];

self.depth = [self.device
        newDepthStencilStateWithDescriptor:depthDescriptor];
```

And finally we set the depth stencil state to our encoder so the rendering pass uses depth testing.

```
[encoder setDepthStencilState:self.depth];
```

Once we make all of the changes above (all uploaded at GitHub) we should see a rotating teapot:



Basic Lighting Effects

Of course while this shows all the basics necessary to draw in 3D, it looks terrible. We need to update our fragment shader function in order to simulate the lighting effects you would see with a light source reflecting off of a surface.

Note that the subject of lighting effects is an entire topic of its own. This section only adds the constants necessary for minimal lighting effects: ambient, diffuse and specular lighting.

Updating the Shader Vertex Values

Before we can carry out any of our lighting effects we need to update our internal structures for the vertex data passed to our fragment shader function. We rewrite our VertexOut structure:

```
struct VertexOut
{
   float4 position [[position]];
   float3 normal;
   float2 texture;
};
```

We also rewrite our vertex shader function to pass the normal and texture values:

```
out.normal = v.normal;
out.texture = v.texture;
```

Ambient Lighting

Ambient lighting is the lighting of an object represented by an omni-directional fixed-intensity light source from the background. The ambient lighting color for a polygon is given by:

$$color = M_{color} \; L_{color} \; ambient$$

Where M is the surface material's color, L is the color of the ambient light, and *ambient* is the brightness of the ambient light. All values are from 0 to 1, and for each color channel red, green and blue, the calculation is repeated for each color channel.

Updating the Shader With Ambient Colors

We set constants defining the color of our light, the color of the teapot, and the ambient intensity, and return that for our fragment color:

```
constant float3 teapotColor(1.0,0.5,0.75);
constant float3 lightColor(1,1,1);
constant float ambientIntensity = 0.1;

fragment float4 fragment_main(VertexOut v [[stage_in]])
{
   return float4(teapotColor * lightColor * ambientIntensity,1.0);
}
```

Diffuse Lighting

Diffuse lighting effects alters the color of our surface depending on the relative angle of the normal of our polygon and the light source illuminating our scene.

This requires we know the angle of the normal vector to our light source. Fortunately we can calculate the relative position of our normal vector and our lighting vector (which is relative to our viewpoint) by finding the inverse of our model matrix and passing that in to our vertex shader.

Updating Our Uniforms

We first add a new field to our Uniforms structure:

```
typedef struct MXUniforms
{
   matrix_float4x4 model;
   matrix_float4x4 view;
   matrix_float4x4 inverse;      // inverse of model
} MXUniforms;
```

Next, we populate the inverse when we populate the contents of our uniforms while drawing:

```
u.inverse = self.model.inverseCtm;
```

Diffuse Lighting Calculations

The diffuse lighting value is given by calculating the intensity by calculating the dot product of the normal vector with the direction to the light. (The dot product is 1 if the normal is pointed in the same direction as the light vector, and 0 if they are perpendicular.)

$$color = M_{color}\ L_{color}\ (N \cdot P)$$

We modify our vertex matrix to calculate the orientation of the normal vector by translating it according to the inverse of the model matrix:

```
float4 nvect = float4(v.normal,0) * u.inverse;
out.normal = normalize(nvect.xyz);
```

We then update our list of constants to give the light location:

```
constant float3 lightDirection(1,0,1);
```

and we update our shader:

```
fragment float4 fragment_main(VertexOut v [[stage_in]])
{
    // Ambient lighting
    float4 ambient = float4(teapotColor * lightColor * ambientIntensity,
            1.0);

    // Diffuse lighting
    float diffuseIntensity = dot(v.normal,lightDirection);
    diffuseIntensity = clamp(diffuseIntensity,0,1);
```

```
    float4 diffuse = float4(teapotColor * lightColor * diffuseIntensity,
            1.0);

    // Specular lighting

    return ambient + diffuse;
}
```

Specular Lighting

The specular lighting effect simulates the reflection of a light on the surface of our object. The specular lighting effect involves us calculating the direction the light reflects off our surface.

$$L_{reflect}=2(N\cdot L)N-L$$

We then calculate the specular intensity by calculating the dot product raised to a power. (Note that the result of the dot product is a value from 0 to 1, so raising it to a power causes the value to be close to zero for most dot product, going to 1 when our dot product is close to 1. Thus higher tightness values cause the specular reflection to be smaller.)

$$s = (L_{reflect} \cdot E)^{tightness}$$

The color is the calculated as a function of the light color, but not the surface color.

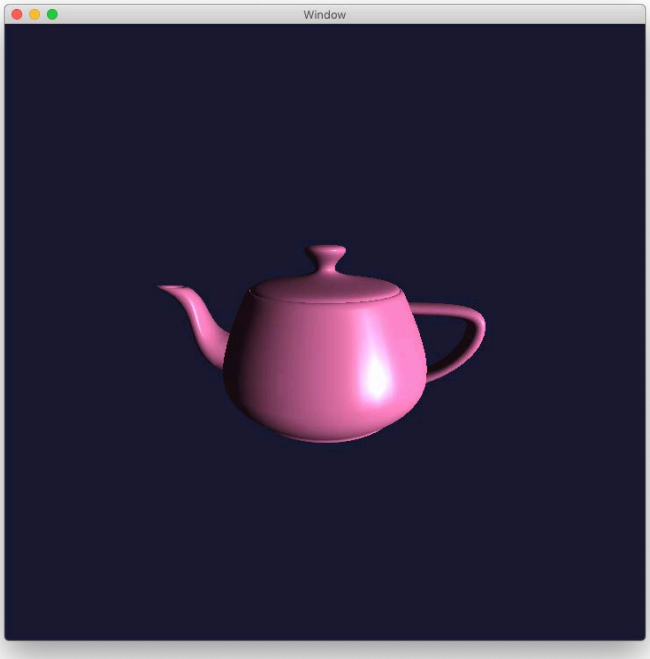
> *Note: One problem with color effects is getting them right. Often things can go haywire because of the wrong vector normal direction, or because of a sign error. If you don't get the light effect you expect it is worth experimenting with the results until you see what you want.*

The math for calculating our specular lighting effect is given by rewriting our fragment shader function:

```
constant float3 teapotColor(1.0,0.5,0.75);

constant float3 lightColor(1,1,1);
constant float ambientIntensity = 0.1;
constant float3 lightDirection(1,0,1);
constant float3 eyeDirection(0,0,1);
constant float specularTightness = 25;
constant float specularIntensity = 0.75;

fragment float4 fragment_main(VertexOut v [[stage_in]])
{
    const float3 normalLight = normalize(lightDirection);
    const float3 normalEye = normalize(eyeDirection);

    // Ambient lighting
    float4 ambient = float4(teapotColor * lightColor * ambientIntensity,
            1.0);
```

```
        // Diffuse lighting
        float dotprod = dot(v.normal,normalLight);
        float diffuseIntensity = clamp(dotprod,0,1);
        float4 diffuse = float4(teapotColor * lightColor * diffuseIntensity,
                1.0);

        // Specular lighting
        float3 refl = (2 * dotprod) * v.normal - normalLight;
        float specIntensity = dot(refl,normalEye);
        specIntensity = clamp(specIntensity,0,1);
        specIntensity = powr(specIntensity,specularTightness);
        float4 specular = float4(lightColor * specIntensity *
                specularIntensity,1.0);

        return ambient + diffuse + specular;
    }
```

With all of these changes (available on GitHub) we should see a final shaded teapot:



Using Textures

Of course pink teapots can get boring fast. You can add more visual complexity to a scene by using texture maps. This shows how to add a texture map, by loading a texture into the GPU, passing it to the shader, and using the texture to specify the color of the triangle.

Note that each vertex in our teapot has a texture coordinate *(u,v)* associated with it, so we know how a texture will wrap around the vertices that make up our teapot.

Loading A Texture

First we need to load a texture into the GPU to render onto our teapot.

Loading the Image Into Memory

The texture is loaded from a resource using the MTKTextureLoader loader, which loads the texture into an MTLTexture.

```
MTKTextureLoader *textureLoader = [[MTKTextureLoader alloc]
        initWithDevice:self.device];

self.texture = [textureLoader newTextureWithName:@"texture"
                                     scaleFactor:1.0
                                          bundle:nil
                                         options:@{}
                                           error:nil];
```

This loads our texture from the asset catalog containing our images.

Pass the Texture to Our Shader

The texture is passed to our GPU through the MTLRenderPassDescriptor.

```
[encoder setFragmentTexture:self.texture atIndex:MXTextureIndex0];
```

Update The Fragment Shader

We now need to update our fragment shader to use the appropriate pixel in our texture.

Obtaining the Texture Parameter

In order to use our texture we need to declare the parameter in the list of parameters to our fragment shader function:

```
fragment float4 fragment_main(VertexOut v [[stage_in]],
                              texture2d<float, access::sample> texture
                                      [[texture(MXTextureIndex0)]])
```

In order to sample the contents of our texture we need to create a sampler, which we can do within the shader function itself:

```
constexpr sampler linearSampler(mip_filter::linear,
                                mag_filter::linear,
                                min_filter::linear);
```

This can also be done by using a MTLSamplerDescriptor to create an MTLSamplerState, and passing it to the command encoder by calling the setFragmentSamplerState:atIndex: method. The sampler is then passed in as a parameter to our fragment shader function via a [[sampler(index)]] parameter.

Getting the Color of the Teapot

We now have the texture location for our teapot and we have the texture. We can now get the color of our teapot at the pixel location by:

```
float3 teapotColor = texture.sample(linearSampler,v.texture).rgb;
```

Once we do these steps (at GitHub) we now have a textured teapot.

Using Stencils

Stencils provide an additional channel of data (beyond depth) which allows control over which pixels are rendered on the screen. Originally used to allow overlaying of graphics, they can be used to achieve a number of effects including reflection. This demo will show the teapot reflected in a mirror.

In order to draw our reflected teapot we'll need to do several things.

First, we'll need to generate two more MTLRenderPipelineState objects; one which will be used to draw the stencil shape, another which will be used to draw a semi-transparent surface which will be our "mirror."

Second, we'll need to create a second fragment function in order to render the semi-transparent surface.

And third we'll need to draw four objects in total in our rendering pass: the stencil (which will be used to clip the reflected teapot so it only draws in the mirror), the reflected teapot, the mirror surface, and the unreflected teapot.

Initialization

There are a number of things we need to initialize so we can render our reflection.

Updating the Depth/Stencil Pixel Format

For our drawing we will use an 8-bit stencil. We will update our view's initialization to specify an 8-bit stencil and how to clear the stencil on the start of drawing.

```
self.depthStencilPixelFormat = MTLPixelFormatDepth32Float_Stencil8;
self.clearStencil = 0;
```

Creating our depth stencil states

Previously we only created a single depth stencil for z-buffer rendering in the *setupDepthStencilState* method. We need to add two more depth stencil states. The first will be used for drawing into the stencil; pixels which pass the depth and compare functions will update the stencil value, but it won't update the depth buffer.

```
depthDescriptor = [[MTLDepthStencilDescriptor alloc] init];
depthDescriptor.depthCompareFunction = MTLCompareFunctionLess;

MTLStencilDescriptor *stencilDescriptor = [[MTLStencilDescriptor alloc]
        init];
stencilDescriptor.stencilCompareFunction = MTLCompareFunctionAlways;
stencilDescriptor.depthStencilPassOperation = MTLStencilOperationReplace;
depthDescriptor.backFaceStencil = stencilDescriptor;
depthDescriptor.frontFaceStencil = stencilDescriptor;
depthDescriptor.depthWriteEnabled = NO;

self.drawStencil = [self.device
        newDepthStencilStateWithDescriptor:depthDescriptor];
```

The second will be used to draw only if the stencil value at a pixel location has been set; this is how we will make sure we only draw our reflected image inside of our mirror:

```
depthDescriptor = [[MTLDepthStencilDescriptor alloc] init];
depthDescriptor.depthCompareFunction = MTLCompareFunctionLess;
```

```
stencilDescriptor = [[MTLStencilDescriptor alloc] init];
stencilDescriptor.stencilCompareFunction = MTLCompareFunctionEqual;
depthDescriptor.backFaceStencil = stencilDescriptor;
depthDescriptor.frontFaceStencil = stencilDescriptor;
depthDescriptor.depthWriteEnabled = YES;

self.maskStencil = [self.device
newDepthStencilStateWithDescriptor:depthDescriptor];
```

Creating Our Pipelines

For drawing our mirror and for drawing our stencil we also need to create two separate pipeline states. Neither need to perform the complicated fragment calculations used to texture map our teapot, but instead can use a very simplified fragment:

```
fragment float4 fragment_mirror(VertexOut v [[stage_in]])
{
   return float4(1,1,1,0.2);
}
```

We load a reference to our fragment function in our *setupPipeline* method:

```
self.fragmentFunction2 = [self.library
        newFunctionWithName:@"fragment_mirror"];
```

We extend our first render pipeline to indicate the format of our stencil attachment:

```
pipelineDescriptor.stencilAttachmentPixelFormat =
        self.depthStencilPixelFormat;
```

And we build two more pipelines; the first with alpha blending enabled. (The red elements show the elements which enable alpha blending.)

```
MTLRenderPipelineDescriptor *pipelineDescriptor2 =
[MTLRenderPipelineDescriptor new];
pipelineDescriptor2.vertexFunction = self.vertexFunction;
pipelineDescriptor2.fragmentFunction = self.fragmentFunction2;
pipelineDescriptor2.vertexDescriptor =
        MTKMetalVertexDescriptorFromModelIO(d);
pipelineDescriptor2.depthAttachmentPixelFormat =
        self.depthStencilPixelFormat;
pipelineDescriptor2.stencilAttachmentPixelFormat =
        self.depthStencilPixelFormat;

pipelineDescriptor2.colorAttachments[0].pixelFormat =
        self.colorPixelFormat;
pipelineDescriptor2.colorAttachments[0].blendingEnabled = YES;
pipelineDescriptor2.colorAttachments[0].rgbBlendOperation =
        MTLBlendOperationAdd;
pipelineDescriptor2.colorAttachments[0].alphaBlendOperation =
        MTLBlendOperationAdd;
```

```
pipelineDescriptor2.colorAttachments[0].sourceRGBBlendFactor =
        MTLBlendFactorSourceAlpha;
pipelineDescriptor2.colorAttachments[0].sourceAlphaBlendFactor =
        MTLBlendFactorSourceAlpha;
pipelineDescriptor2.colorAttachments[0].destinationRGBBlendFactor =
        MTLBlendFactorOneMinusSourceAlpha;
pipelineDescriptor2.colorAttachments[0].destinationAlphaBlendFactor =
        MTLBlendFactorOneMinusSourceAlpha;

self.pipeline2 = [self.device
        newRenderPipelineStateWithDescriptor:pipelineDescriptor2
        error:nil];
```

The second has drawing disabled (highlighted segment in red):

```
MTLRenderPipelineDescriptor *pipelineDescriptor3 =
        [MTLRenderPipelineDescriptor new];
pipelineDescriptor3.vertexFunction = self.vertexFunction;
pipelineDescriptor3.fragmentFunction = self.fragmentFunction2;
pipelineDescriptor3.vertexDescriptor =
        MTKMetalVertexDescriptorFromModelIO(d);
pipelineDescriptor3.depthAttachmentPixelFormat =
        self.depthStencilPixelFormat;
pipelineDescriptor3.stencilAttachmentPixelFormat =
        self.depthStencilPixelFormat;

pipelineDescriptor3.colorAttachments[0].pixelFormat =
        self.colorPixelFormat;
pipelineDescriptor3.colorAttachments[0].writeMask = MTLColorWriteMaskNone;

self.pipelineNoDraw = [self.device
        newRenderPipelineStateWithDescriptor:pipelineDescriptor3
        error:nil];
```

Create Our Mirror Square

Our mirror is simply a square, and we create a square with two triangles and store it in an MTLBuffer, using the same MXVertex structure used to store our teapot:

```
static const MXVertex square[] = {
    { { -0.5, 0, -0.5 }, { 0, 0, 1 }, { -1, -1 } },
    { { -0.5, 0,  0.5 }, { 0, 0, 1 }, { -1,  1 } },
    { {  0.5, 0, -0.5 }, { 0, 0, 1 }, {  1, -1 } },
    { { -0.5, 0,  0.5 }, { 0, 0, 1 }, { -1,  1 } },
    { {  0.5, 0, -0.5 }, { 0, 0, 1 }, {  1, -1 } },
    { {  0.5, 0,  0.5 }, { 0, 0, 1 }, {  1,  1 } },
};
self.mirror = [self.device newBufferWithBytes:square
                                       length:sizeof(square)
                                      options:MTLResourceOptionCPUCacheModeDefault];
```

Refactor Our Drawing

Because we're drawing the mesh that represents our teapot multiple times, we extract our drawing code into a common method:

```
- (void)renderMesh:(NSArray<MTKMesh *> *)meshArray
         inEncoder:(id<MTLRenderCommandEncoder>)encoder
{
    for (MTKMesh *mesh in meshArray) {
        MTKMeshBuffer *vertexBuffer = [[mesh vertexBuffers] firstObject];
        [encoder setVertexBuffer:vertexBuffer.buffer
                          offset:vertexBuffer.offset
                         atIndex:0];

        for (MTKSubmesh *submesh in mesh.submeshes) {
            MTKMeshBuffer *indexBuffer = submesh.indexBuffer;
            [encoder drawIndexedPrimitives:submesh.primitiveType
                                indexCount:submesh.indexCount
                                 indexType:submesh.indexType
                               indexBuffer:indexBuffer.buffer
                         indexBufferOffset:indexBuffer.offset];
        }
    }
}
```

This method is sufficiently common enough it's worth copying this into its own method to simplify drawing.

Drawing Our Model.

With all the pieces (the three separate pipelines, the three separate depth descriptors, our model and our refactored drawing, we can now render the elements of our scene in our rendering pass.

Note that because we're dealing with a semi-transparent polygon, we must render the objects in the correct order: the upside down teapot must be rendered *before* the semi-transparent mirror so that the colors of our teapot are attenuated properly.

Render the Stencil

The first step is to draw the shape of our mirrored surface in the stencil buffer. We use this with the non-drawing pipeline (so we don't draw into the color buffer or the depth buffer), and render our mirror surface to set *only* the stencil buffer.

```
[self.view push];
[self.view translateByX:0 y:-0.55 z:0];

u.view = self.view.ctm;
[encoder setVertexBytes:&u
                 length:sizeof(MXUniforms)
                atIndex:MXVertexIndexUniforms];
```

```
[encoder setStencilReferenceValue:1];
[encoder setRenderPipelineState:self.pipelineNoDraw];
[encoder setDepthStencilState:self.drawStencil];
[encoder setCullMode:MTLCullModeNone];
[encoder setVertexBuffer:self.mirror
                  offset:0
                 atIndex:MXVertexIndexVertices];
[encoder drawPrimitives:MTLPrimitiveTypeTriangle vertexStart:0
vertexCount:6];

[self.view pop];
```

This will set the view matrix (pushing the old CTM state in order to preserve it), and perform the translation to move our mirror surface down below the teapot. We then render our mirror surface, setting the pipeline, stencil state and culling modes as required so we only draw in the stencil.

Render the Teapot

Now we render our right-side up teapot:

```
u.view = self.view.ctm;
[encoder setVertexBytes:&u
                 length:sizeof(MXUniforms)
                atIndex:MXVertexIndexUniforms];

[encoder setRenderPipelineState:self.pipeline];
[encoder setCullMode:MTLCullModeFront];
[encoder setDepthStencilState:self.depth];
[encoder setFragmentTexture:self.texture atIndex:MXTextureIndex0];

[self renderMesh:self.teapot inEncoder:encoder];
```

Render the Reflected Teapot

Now we render the upside down teapot. We render the teapot using the depth stencil state, so we only render in the pixels that were originally drawn above. This causes the reflected teapot to only be visible in the mirror.

```
[self.view push];
[self.view translateByX:0 y:-1.1 z:0];
[self.view scaleByX:1 y:-1 z:1];

u.view = self.view.ctm;
[encoder setVertexBytes:&u
                 length:sizeof(MXUniforms)
                atIndex:MXVertexIndexUniforms];

// scale by -1 flips the winding order, so flip the culling mode.
[encoder setRenderPipelineState:self.pipeline];
[encoder setCullMode:MTLCullModeBack];
```

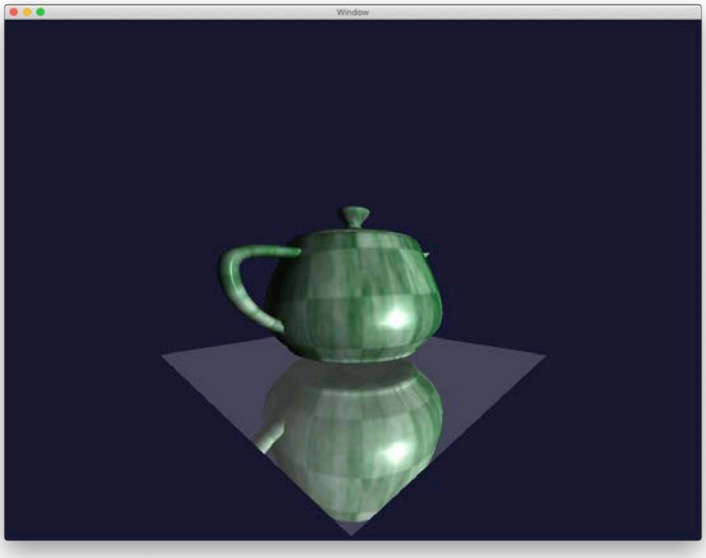
```
[encoder setDepthStencilState:self.maskStencil];
[encoder setFragmentTexture:self.texture atIndex:MXTextureIndex0];

[self renderMesh:self.teapot inEncoder:encoder];
[self.view pop];
```

Render the Mirror Surface

And finally we render the semi-transparent mirror surface. This will be rendered in front of our reflected teapot, giving it the mirror surface appearance.

```
[self.view push];
[self.view translateByX:0 y:-0.55 z:0];

u.view = self.view.ctm;
[encoder setVertexBytes:&u length:sizeof(MXUniforms)
atIndex:MXVertexIndexUniforms];

[encoder setRenderPipelineState:self.pipeline2];
[encoder setCullMode:MTLCullModeNone];
[encoder setDepthStencilState:self.depth];
[encoder setVertexBuffer:self.mirror
                  offset:0
                 atIndex:MXVertexIndexVertices];
[encoder drawPrimitives:MTLPrimitiveTypeTriangle vertexStart:0
vertexCount:6];

[self.view pop];
```

Once these changes have been made (at GitHub) you should see the following:



Complex Rendering Techniques

This section covers more complex rendering techniques, including techniques which execute multiple rendering passes or which combine compute kernels and rendering techniques for rendering the results of a simulation.

It is worth going through these examples after going through the examples above, and they are provided in as terse a way possible so you can understand how they work and which elements of Metal are used to execute them.

Shadow Mapping

Shadow mapping is its own complex topic, and there are several tutorials on the topic. This provides a simple implementation which adds shadows to our rotating teapot. We start with the unreflected teapot from the prior example.

This example will require two pass rendering: the first pass renders a shadow map from the point of view of the light, rendering the results into an off-screen texture in the GPU. The second pass uses the off-screen texture to apply shadows to the surface of our teapot.

This demo illustrates the steps necessary in Metal, as an illustration of how Metal works. The result is passable--but requires refinement if we want something that is visually pleasing.

The Shadow Rendering Pass

We must first construct all of the infrastructure for rendering our teapot from the point of view of our light. The resulting depth map will become our shadow mask, used to test if a point on our final rendered image is in the shadows.

Adding the Shadow Transformation Matrix to Our Uniforms

Shadow mapping works by rendering the scene's depth map from the point of view of our light. This requires we supply a matrix which can be used to render the scene from our light's point of view.

```
typedef struct MXUniforms
{
    matrix_float4x4 model;
    matrix_float4x4 view;
    matrix_float4x4 inverse;  // inverse of model
    matrix_float4x4 shadow;   // matrix for light position/shadow mapping
} MXUniforms;
```

Defining our Shadow Shader Functions

We now need to create two new shader functions used by our shadow rendering pipeline. Since we are rendering to the depth map but not rendering colors, our vertex and fragment shaders are quite simple.

The vertex shader only needs to transform vertices to the destination output using the new shadow matrix. The fragment shader does nothing.

```
vertex VertexOut vertex_shadow(VertexIn v [[stage_in]],
           constant MXUniforms &u [[buffer(MXVertexIndexUniforms)]])
{
    VertexOut out;
    out.position = u.shadow * float4(v.position,1.0);
    return out;
}

fragment void fragment_shadow(VertexOut v [[stage_in]])
{
}
```

Setting Up Our Shadow Render Pipeline

The steps are similar to setting up our original pipeline earlier. The key differences is that we are only rendering to our depth map; we are not rendering to a color texture map.

First, we load our shadow shader functions:

```
self.shadowVertexFunction = [self.library
        newFunctionWithName:@"vertex_shadow"];
self.shadowFragmentFunction = [self.library
        newFunctionWithName:@"fragment_shadow"];
```

Next, we create our shadow pipeline, disabling rendering to the color output channel:

```
MTLRenderPipelineDescriptor *shadowPipelineDescriptor =
        [MTLRenderPipelineDescriptor new];
shadowPipelineDescriptor.vertexFunction = self.shadowVertexFunction;
shadowPipelineDescriptor.fragmentFunction = self.shadowFragmentFunction;
shadowPipelineDescriptor.vertexDescriptor =
        pipelineDescriptor.vertexDescriptor;
shadowPipelineDescriptor.depthAttachmentPixelFormat =
        MTLPixelFormatDepth32Float;
shadowPipelineDescriptor.colorAttachments[0].writeMask =
        MTLColorWriteMaskNone;

self.shadowPipeline = [self.device
        newRenderPipelineStateWithDescriptor:shadowPipelineDescriptor
        error:nil];
```

The Shadow Map

Our shadow map is simply the depth map from the first rendering pass generated by our shadow pipeline. We render this into a texture map with format MTLPixelFormatDepth32Float.

First, we create a MTLTextureDescriptor which describes the format of our texture map:

```
MTLTextureDescriptor *shadowDescriptor = [MTLTextureDescriptor
        texture2DDescriptorWithPixelFormat:MTLPixelFormatDepth32Float
                                     width:SHADOW_WIDTH
                                    height:SHADOW_HEIGHT
                                 mipmapped:NO];
```

The dimensions SHADOW_WIDTH and SHADOW_HEIGHT are defined at the top of our MXView.m file.

Note that our texture is not "mipmapped".

Before we create our texture we need to set some parameters. Because our depth map is only used by the GPU, we need to set the storage mode as private:

```
shadowDescriptor.storageMode = MTLStorageModePrivate;
```

We also need to set the usage of our texture. Because we're rendering to our texture we need to set the texture as a render target. We also need to set our texture usage for reading so we can read from it in the second rendering pass.

```
shadowDescriptor.usage = MTLTextureUsageRenderTarget |
        MTLTextureUsageShaderRead;
```

Once we've set up our descriptor we can create the MTLTexture which represents our shadow map:

```
self.shadowMap = [self.device newTextureWithDescriptor:shadowDescriptor];
```

Rendering the Shadow Map

Our *drawInMTKView* method is reorganized to make it easier to see the two rendering passes, with the transformation matrices moved to the top of the function. First, we generate our transformations and populate our uniforms structure as before, and we also set our shadow matrix.

> *Note: The shadow map population code in this example is kludged. In a more sophisticated application you would generate the shadow map according to the location of the light in your scene.*

For our first rendering pass where we render our shadow map, we create a render pass descriptor and populate it ourselves. Note that the store action is to store; we wish to preserve the depth map generated for our second pass. We also set the depth attachment texture to our shadow map, so the results of the first pass is rendered into our shadow map.

```
descriptor = [MTLRenderPassDescriptor renderPassDescriptor];
descriptor.depthAttachment.texture = self.shadowMap;
descriptor.depthAttachment.loadAction = MTLLoadActionClear;
descriptor.depthAttachment.storeAction = MTLStoreActionStore;
descriptor.depthAttachment.clearDepth = 1.0;
encoder = [buffer renderCommandEncoderWithDescriptor:descriptor];
```

Next, we set up the encoder to render using our shadow pipeline. We reuse the depth stencil state, since our behavior for handling depth mapping is the same as for rendering our scene: our pixels are updated if a pixel is closer to our camera (or, in this case, our light). (But since we're only rendering the depth map and not color pixels, this means in the end our depth map will contain the depth of the closest pixel at each pixel location.)

```
[encoder setRenderPipelineState:self.shadowPipeline];
[encoder setVertexBytes:&u length:sizeof(MXUniforms)
atIndex:MXVertexIndexUniforms];
[encoder setDepthStencilState:self.depth];
[self renderMesh:self.teapot inEncoder:encoder];
[encoder endEncoding];
```

At the end of the first pass, we will have a depth map from the point of view of our camera.

In the image above, white is a depth of 1.0, and darker shades are pixels closer to the light. (This was captured by using the debug functionality in Xcode.)

Using the Shadow Map

Now that we have our shadow map we need to use it when rendering our scene. We do this by updating our shaders from a prior example to add the shadow coordinates to our vertices.

The idea is this: instead of simply tracking the position of each vertex where the vertex will eventually be rendered on our display, we track *two positions:* the position of the vertex on our display *and the position of the vertex in our shadow map*. Then in the fragment shader we can determine if our pixel is in the shadow, and adjust our colors accordingly.

Updating the Shader's VertexOut Structure

We need to add the position of our vertex in our shadow map to our *VertexOut* structure in our shader.

```
struct VertexOut
{
   float4 position [[position]];
   float4 shadow;
   float3 normal;
```

```
    float2 texture;
};
```

Populating the Shadow Position

Our vertex shader function needs to be updated to populate the shadow field. We reuse our shadow transformation matrix to do this:

```
vertex VertexOut vertex_main(VertexIn v [[stage_in]],
        constant MXUniforms &u [[buffer(MXVertexIndexUniforms)]])
{
    VertexOut out;

    float4 worldPosition = u.model * float4(v.position,1.0);
    out.position = u.view * worldPosition;

    out.shadow = u.shadow * float4(v.position,1.0);

    float4 nvect = float4(v.normal,0) * u.inverse;
    out.normal = normalize(nvect.xyz);
    out.texture = v.texture;

    return out;
}
```

The Second Rendering Pass

Remember: this is a two-pass rendering operation. We've now rendered the shadow map, we next must render our teapot. The steps are the same as above, but we also pass in our shadow map as a fragment texture, so we can use the shadow map during rendering. (The red highlighted segment is the additional step necessary to pass in our shadow map.)

```
    descriptor = [view currentRenderPassDescriptor];
    encoder = [buffer renderCommandEncoderWithDescriptor:descriptor];
    [encoder setRenderPipelineState:self.pipeline];
    [encoder setVertexBytes:&u
                     length:sizeof(MXUniforms)
                    atIndex:MXVertexIndexUniforms];
    [encoder setDepthStencilState:self.depth];

    [encoder setFragmentTexture:self.texture
                        atIndex:MXTextureIndex0];
    [encoder setFragmentTexture:self.shadowMap
                        atIndex:MXTextureIndexShadow];
    [self renderMesh:self.teapot inEncoder:encoder];
    [encoder endEncoding];
```

Updating Our Fragment Shader Function

We also need to modify our fragment shader function to obtain the depth from our shadow map, and determine if the depth of the pixel we wish to render is in front of the depth from our shadow map (and thus, visible to the light), or if it is behind the depth from our shadow map (and thus, is in the shadows).

First, we need to update our fragment shader function to add the shadow map as a parameter:

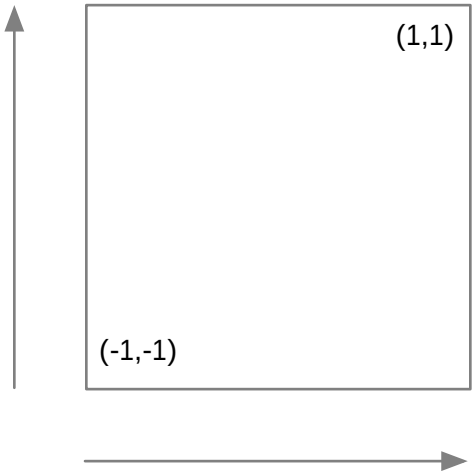
```
fragment float4 fragment_main(VertexOut v [[stage_in]],
        texture2d<float, access::sample> texture
                [[texture(MXTextureIndex0)]],
        depth2d<float, access::sample> shadowMap
                [[texture(MXTextureIndexShadow)]])
```

Next we need to get the depth from our shadow map. We need to do a little math to get the correct pixel, however. First, because our *shadow* field is in homogeneous coordinates, we need to calculate:

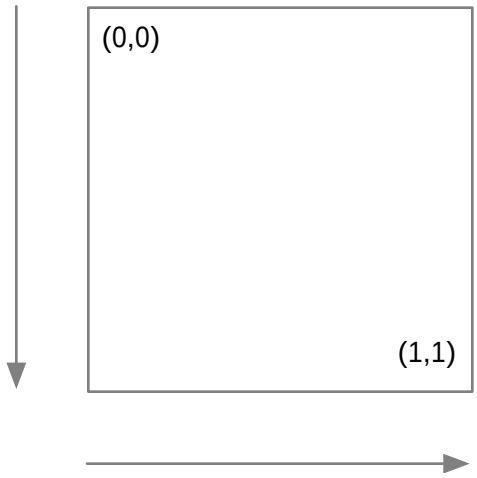
$$(x,y,z) = \left(\frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'}\right)$$

> *Note: If we were using an orthographic projection for our light source--that is, if our light source was at infinity, we can skip the division by w. That's because in an orthographic projection, w = 1.*
>
> *If this was the case, many of these steps to remap our shadow coordinates into texture coordinates could be put into the vertex shader, gaining a bit of a performance gain since we would not need to do three divisions by w, as well as addition and subtraction and division by 2.*
>
> *This is left as an exercise for the reader.*

Next we need to take into account that while points in our geometry coordinate system run from -1 to 1 from bottom left to top right:



Pixels in our texture map run from 0 to 1 from top left to bottom right:



We handle this in our fragment shader.

```
float x = (1 + v.shadow.x / v.shadow.w) / 2;
float y = (1 - v.shadow.y / v.shadow.w) / 2;
float depth = shadowMap.sample(linearSampler,float2(x,y));
float zd = v.shadow.z / v.shadow.w - 0.001;
```

At this point, zd is the depth of our current point in our shadow coordinate system, and depth is the depth of the pixel closest to the light. (We subtract a small fudge factor in order to prevent "shadow acne.")

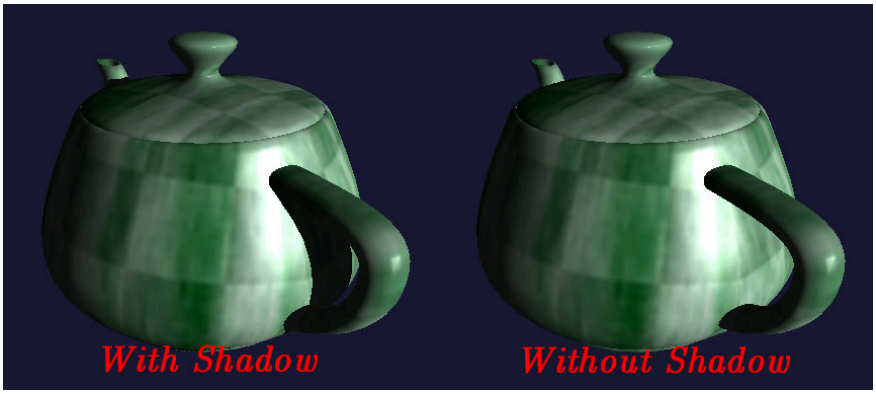
This means if zd is larger than depth, our point is farther from the light, and is in the shadow.

We use this to skip calculating the diffuse and specular lighting in our fragment shader by adding the following line to skip those calculations:

```
if (zd >= depth) return ambient;
```

One all these changes are made (reflected in GitHub), we should see:

The effect can be subtle, and is more apparent when running the application and watching the animation. However, we can compare the image with a version without shadow mapping:



If you look at where the handle meets the body, you can see shadowing making the teapot looking more "realistic."

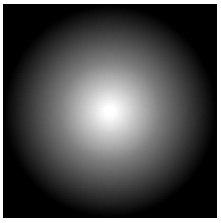
Fairy Lights

Fairy lights is one term for small glowing dots of lights that can dance around a scene, and this next demo will show adding fairy lights by demonstrating two aspects of Metal: rendering with the drawPrimitives:vertexStart:vertexCount:instanceCount: method (and modifying the vertex shader function to deal with multiple instances of the same object), and with a vertex shader that is a little more interesting than just transforming points in 3D space.

Each fairy light is rendered as a square with a texture associated with it; the texture is semi-transparent (to render the glowing effect).

Loading the Fairy Texture

The fairy texture itself is stored as a texture in our assets.



Our fragment shader will color the texture with the appropriate color. It's loaded along with the texture for our teapot:

```
    self.fairyTexture = [textureLoader newTextureWithName:@"fairy"
                                              scaleFactor:1.0
                                                   bundle:nil
                                                  options:@{}
```

```
                                                    error:nil];
```

Setting Up the Vertices

The vertices for our fairy lights are represented as two dimensional vertices:

```
struct VertexIn {
    float2 position       [[attribute(MXAttributeIndexPosition)]];
};
```

(And the corresponding MXFairyVertex declaration in MXGeometry.h.)

We then define the square as two triangles:

```
static const MXFairyVertex square[] = {
    { { -1, -1 } },
    { { -1,  1 } },
    { {  1, -1 } },
    { { -1,  1 } },
    { {  1, -1 } },
    { {  1,  1 } }
};
self.fairySquare = [self.device newBufferWithBytes:square
                                            length:sizeof(square)
                        options:MTLResourceOptionCPUCacheModeDefault];
```

Setting Up the Fairy Locations

Each fairy light instance will be drawn with its corresponding fairy light record. The list of fairy lights will be passed in with our fairy vectors, which will have the end result of duplicating our squares, once for each fairy light record.

The Fairy Light Instance Record

Our fairy light record is:

```
struct FairyLightIn {
   float3 position;
   float3 color;
   float size;

   float2 angle;
   float speed;
   float radius;
};
```

(And the corresponding MXFairyLocation declaration in MXGeometry.h)

Initializing the Fairy Lights

The fairy lights are stored as a fixed array in our MXView record. The lights are initialized with random locations and rotational speeds (so they are scattered around the waist of the teapot) in the *setupFairyLights* method.

Adding Our Fairy Shader Functions

We now need to create our vertex and fragment shader functions.

The Vertex Shader Function

Our vertex shader function will be invoked with a drawing method which calls the fairy vertex function for each vertex in our box, for each instance of our fairy lights. This will allow us to specify the location of our vertices on our screen.

So in addition to the vertex location of each corner of our square and the uniforms giving the transformations we will use to find the position of our fairy lights, we also receive the array of fairy light positions and an instance index that allows us to get the values for the specific instance being rendered:

```
vertex VertexOut vertex_fairy(VertexIn v [[stage_in]],
      constant MXUniforms &u [[buffer(MXVertexIndexUniforms)]],
      constant FairyLightIn *positions [[buffer(MXVertexIndexLocations)]],
      uint iid [[instance_id]])
```

The array of fairy light positions are specified as passed in with the vertex buffers, and the specific instance being rendered is specified with the *[[instance_id]]* attribute.

> *Note: the attributes [[instance_id]] is used to iterate through all the instances when any version of the MTLRenderCommandEncoder draw methods that specify an instance count is called. The array of positions happens to have the same length as the range of instance counts.*
>
> *The Metal Shader specification also specifies a [[vertex_id]] if you wish to iterate through the vertices in a similar fashion. Both of these can be used with the [[base_vertex]] and [[base_instance]] attributes.*

Now the position of the center of each fairy light should be specified by the 3D transformed into screen space:

```
float4 screenPos = u.view * u.model * float4(positions[iid].position,1.0);
```

This gives us the center of our fairy light on the screen. We then adjust the corners appropriately by using the vertex parameter, adjusting the size by dividing by the w component of our screen position. (The w component correlates to how far from the camera our light is.)

```
screenPos.xy += v.position * 0.03 * positions[iid].size / screenPos.w;
```

We then pass the screen position to our vertex shader, as well as the UV location (for the texture map) and the color (from the fairy lights array) to our shader:

```
out.position = screenPos;
out.uv = (v.position + 1)/2;
out.color = positions[iid].color;
```

The Fragment Shader Function

The fragment shader function adjusts the color of our texture and sets the transparency for our color.

```
fragment float4 fragment_fairy(VertexOut v [[stage_in]],
        texture2d<float> fairyTexture [[texture(MXTextureIndex0)]])
{
    constexpr sampler linearSampler(mip_filter::linear,
                                    mag_filter::linear,
                                    min_filter::linear);

    float4 c = fairyTexture.sample(linearSampler,v.uv);
    float3 color = v.color * c.x;
    return float4(color,c.x);
}
```

Setting Up the Fairy Rendering Pipeline

Because our fairy lights are rendered as semi-transparent lights we need to create a new rendering pipeline to handle our fairy lights.

The Fairy Attribute Descriptors

Because our fairy light vertices have a different layout than our teapot, we need to create a new MTLVertexDescriptor describing our layout of our MXFairyVertex record:

```
MTLVertexDescriptor *fairyDescriptors = [[MTLVertexDescriptor alloc]
        init];
fairyDescriptors.attributes[MXAttributeIndexPosition].format =
        MTLVertexFormatFloat2;
fairyDescriptors.attributes[MXAttributeIndexPosition].offset = 0;
fairyDescriptors.attributes[MXAttributeIndexPosition].bufferIndex = 0;
fairyDescriptors.layouts[0].stride = sizeof(MXFairyVertex);
```

The Fairy Attribute Pipeline State

We can now set up our pipeline. We need to set the blending options to our pipeline descriptor to handle alpha blending:

```
MTLRenderPipelineDescriptor *fairyPipelineDescriptor =
[MTLRenderPipelineDescriptor new];
```

```
fairyPipelineDescriptor.vertexFunction = self.fairyVertexFunction;
fairyPipelineDescriptor.fragmentFunction = self.fairyFragmentFunction;
fairyPipelineDescriptor.vertexDescriptor = fairyDescriptors;
fairyPipelineDescriptor.depthAttachmentPixelFormat =
self.depthStencilPixelFormat;
fairyPipelineDescriptor.colorAttachments[0].pixelFormat =
self.colorPixelFormat;

fairyPipelineDescriptor.colorAttachments[0].blendingEnabled = YES;
fairyPipelineDescriptor.colorAttachments[0].rgbBlendOperation =
MTLBlendOperationAdd;
fairyPipelineDescriptor.colorAttachments[0].alphaBlendOperation =
MTLBlendOperationAdd;
fairyPipelineDescriptor.colorAttachments[0].sourceRGBBlendFactor =
MTLBlendFactorSourceAlpha;
fairyPipelineDescriptor.colorAttachments[0].sourceAlphaBlendFactor =
MTLBlendFactorSourceAlpha;
fairyPipelineDescriptor.colorAttachments[0].destinationRGBBlendFactor =
MTLBlendFactorOneMinusSourceAlpha;
fairyPipelineDescriptor.colorAttachments[0].destinationAlphaBlendFactor =
MTLBlendFactorOneMinusSourceAlpha;

self.fairyPipeline = [self.device
newRenderPipelineStateWithDescriptor:fairyPipelineDescriptor error:nil];
```

The Fairy Depth Stencil

Fairy lights are drawn as semi-transparent objects. This implies two things: we must render them after we've rendered the other objects in our scene. And since fairy lights do not block each other (in z-order), fairy lights should not update the depth buffer (and cause fairy lights not drawn in order to block each other).

We can achieve the latter part by creating a new MTLDepthStencilState:

```
MTLDepthStencilDescriptor *fairyDepthDescriptor =
        [[MTLDepthStencilDescriptor alloc] init];
fairyDepthDescriptor.depthCompareFunction = MTLCompareFunctionLess;
[fairyDepthDescriptor setDepthWriteEnabled:NO];

self.fairyDepth = [self.device
        newDepthStencilStateWithDescriptor:fairyDepthDescriptor];
```

Rendering Our Fairy Lights

Now that we've set up our rendering pipeline and geometry, we can now render our fairy lights.

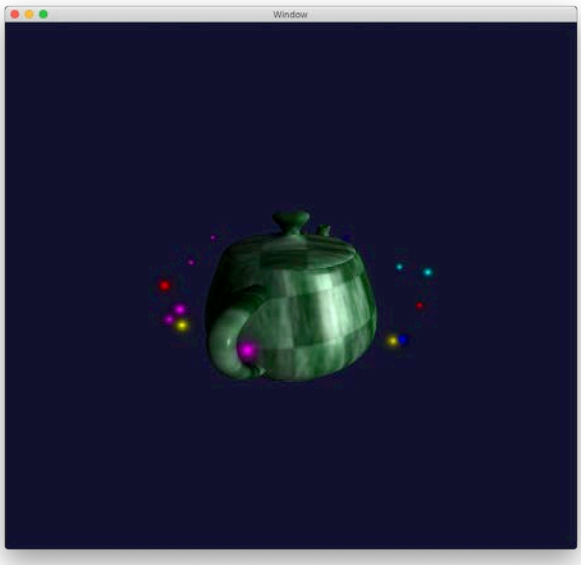
Before committing our final rendering pass to render our teapot, we add the code to update our fairy light positions, then pass in the information to draw our fairy lights:

```
for (int i = 0; i < MAX_FAIRYLIGHTS; ++i) {
```

```
    ... Update our fairy light locations.
}
[encoder setRenderPipelineState:self.fairyPipeline];
[encoder setDepthStencilState:self.fairyDepth];
[encoder setVertexBuffer:self.fairySquare
                  offset:0
                 atIndex:MXVertexIndexVertices];
[encoder setVertexBytes:fairyLights
                 length:sizeof(fairyLights)
                atIndex:MXVertexIndexLocations];
[encoder setFragmentTexture:self.fairyTexture atIndex:MXTextureIndex0];
[encoder drawPrimitives:MTLPrimitiveTypeTriangle
            vertexStart:0
            vertexCount:6
          instanceCount:MAX_FAIRYLIGHTS];

[encoder endEncoding];
```

Note the line highlighted in red is how we request we render multiple instances of the same vertex array.

When all of this is put together (as noted in GitHub), we should get fairy lights:



Deferred Shading

The problem with our fairy lights is that we don't see their illumination reflected on the surface of our teapot. We can add this effect by using deferred shading.

In essence, instead of rendering our teapot to the on-screen display, we render our teapot to an off-screen buffer. We also render other information we need to properly calculate our lighting effects (such as the normal vector for each visible pixel) into separate buffers--collectively known as a Geometry Buffer or G-Buffer. With the information in our G-Buffer we can then calculate the lighting effects for each of our

lights, and limit the lighting effect so we only add the light from each fairy light that is close to the pixels to be updated.

Because the purpose of this demo is to illustrate the parts of Metal required to execute Deferred Shading, some of this presentation won't be as optimal as possible. Further, as these demos target the Macintosh, we won't take advantage of Raster Order Groups, as used in Apple's Deferred Lighting example.

Generating the G-Buffer

The first phase of our changes will split our rendering pipeline into two phases: one which generates the G-Buffer and one which uses the G-Buffer to perform our lighting calculations.

Our lighting calculations (from above) require the color of each pixel in the teapot and the normal vector; from that, we calculate the shaded colors of our teapot.

> *Note: Different shading effects may require different values to be calculated in the first phase, to be carried over to the second phase in our G-Buffer. For example, Apple's deferred lighting example calculates albedo and specular information rather than color information, then obtains the texture map color information at a later rendering pass.*

Create the GBuffer

The first rendering pass for our G-Buffer system generates our G-Buffer. While the *vertex_gbuffer* vertex shader function remains the same as the prior *vertex_main* vertex buffer from above, the second half which renders our fragment is modified to render our G-Buffer instead.

We declare our G-Buffer:

```
struct GBufferOut
{
    float4 color     [[color(MXColorIndexColor)]];
    float4 normal    [[color(MXColorIndexNormal)]];
};
```

The [[color(N)]] attribute is associated with the colorAttachment array of the MTLRenderPassDescriptor. It allows us to write multiple color channels at the same time.

In this case we're writing to two texture maps. The first will contain the color of our object as (r,g,b), and the alpha channel contains the shadow (with 0 meaning the pixel is in a shadow and 1 meaning it is not). The second will contain our transformed normal vector in (r,g,b), and the alpha channel is ignored.

Our fragment shader which renders into our G-Buffer should seem familiar as it is the first half of our prior *fragment_main* fragment shader function, but instead of calculating our lighting effects we put our intermediate values into our output buffer.

```
fragment GBufferOut fragment_gbuffer(VertexOut v [[stage_in]],
        texture2d<float, access::sample> texture
                [[texture(MXTextureIndex0)]],
        depth2d<float, access::sample> shadowMap
```

```
                [[texture(MXTextureIndexShadow)]])
{
    constexpr sampler linearSampler (mip_filter::linear,
                                     mag_filter::linear,
                                     min_filter::linear);

   float x = (1 + v.shadow.x / v.shadow.w) / 2;
   float y = (1 - v.shadow.y / v.shadow.w) / 2;
   float depth = shadowMap.sample(linearSampler,float2(x,y));
   float zd = v.shadow.z / v.shadow.w - 0.001;

   /*
    *    Generate the GBuffer
    */

   GBufferOut out;

   // Texture color
   float3 teapotColor = texture.sample(linearSampler,v.texture).rgb;
   out.color = float4(teapotColor, (zd >= depth) ? 0 : 1);

   // Normal vector
   out.normal = float4(v.normal,0);

   return out;
}
```

Creating our GBuffer rendering pipeline

We now load our new shader functions:

```
self.gVertexFunction = [self.library
        newFunctionWithName:@"vertex_gbuffer"];
self.gFragmentFunction = [self.library
        newFunctionWithName:@"fragment_gbuffer"];
```

The rendering pipeline will then make use of our new vertex functions, and must also specify the format of our color attachments:

```
MTLRenderPipelineDescriptor *gBufferPipelineDescriptor =
[MTLRenderPipelineDescriptor new];
gBufferPipelineDescriptor.vertexFunction = self.gVertexFunction;
gBufferPipelineDescriptor.fragmentFunction = self.gFragmentFunction;
gBufferPipelineDescriptor.colorAttachments[0].pixelFormat =
        MTLPixelFormatRGBA16Float;
gBufferPipelineDescriptor.colorAttachments[1].pixelFormat =
        MTLPixelFormatRGBA32Float;
gBufferPipelineDescriptor.vertexDescriptor =
        MTKMetalVertexDescriptorFromModelIO(d);
gBufferPipelineDescriptor.depthAttachmentPixelFormat =
        MTLPixelFormatDepth32Float_Stencil8;
gBufferPipelineDescriptor.stencilAttachmentPixelFormat =
```

```
        MTLPixelFormatDepth32Float_Stencil8;

self.gPipeline = [self.device
        newRenderPipelineStateWithDescriptor:gBufferPipelineDescriptor
        error:nil];
```

Creating our Stencil

We also use a stencil in our G-Buffer, to note the pixels that are being rendered to. This way we can skip the majority of the screen that does not contain a teapot. (This is important so we can bypass doing lighting calculations where they are not necessary.)

Our drawing stencil sets the stencil value to a non-zero value as we render.

```
MTLDepthStencilDescriptor *drawStencilDescriptor =
[[MTLDepthStencilDescriptor alloc] init];
drawStencilDescriptor.depthCompareFunction = MTLCompareFunctionLess;
[drawStencilDescriptor setDepthWriteEnabled:YES];

MTLStencilDescriptor *drawStencil = [[MTLStencilDescriptor alloc] init];
drawStencil.stencilCompareFunction = MTLCompareFunctionAlways;
drawStencil.depthStencilPassOperation = MTLStencilOperationReplace;
drawStencilDescriptor.backFaceStencil = drawStencil;
drawStencilDescriptor.frontFaceStencil = drawStencil;

self.drawStencil = [self.device
newDepthStencilStateWithDescriptor:drawStencilDescriptor];
```

Allocating the G-Buffer Textures

Our G-Buffer textures are the same dimensions as our screen. We use calls to *mtkView:drawableSizeWillChange:* in order to allocate and resize our textures as the screen is created and as it is resized. The textures are then updated in the *setupGBufferTextureWithSize:* method call.

```
MTLTextureDescriptor *gbufferDescriptor;

// Color map
gbufferDescriptor = [MTLTextureDescriptor
texture2DDescriptorWithPixelFormat:MTLPixelFormatRGBA16Float
width:size.width height:size.height mipmapped:NO];
gbufferDescriptor.storageMode = MTLStorageModePrivate;
gbufferDescriptor.usage = MTLTextureUsageRenderTarget |
MTLTextureUsageShaderRead;
self.colorMap = [self.device newTextureWithDescriptor:gbufferDescriptor];

// normal vectors
gbufferDescriptor = [MTLTextureDescriptor
texture2DDescriptorWithPixelFormat:MTLPixelFormatRGBA32Float
width:size.width height:size.height mipmapped:NO];
gbufferDescriptor.storageMode = MTLStorageModePrivate;
```

```
gbufferDescriptor.usage = MTLTextureUsageRenderTarget |
MTLTextureUsageShaderRead;
self.normalMap = [self.device newTextureWithDescriptor:gbufferDescriptor];
```

Note that we use a different format for our colors than for our normals, so we can preserve the maximum resolution possible for our normals. Of course in practice you would fiddle with these parameters to balance saving memory and graphical resolution.

Also note that sometimes on MacOS, the drawInMTKView method may be called prior to our resize method being called. If this happens we need to skip drawing, as the GBuffer textures have not been initialized yet:

```
if (self.colorMap == nil) return;
```

Rendering our G-Buffers

We now can render our G-Buffer. Note that the depth and stencils we use for rendering is the same as the depth and stencil rendering textures used by our view.

First, we set up our descriptor:

```
descriptor = [MTLRenderPassDescriptor renderPassDescriptor];
descriptor.depthAttachment.texture = self.depthStencilTexture;
descriptor.depthAttachment.loadAction = MTLLoadActionClear;
descriptor.depthAttachment.storeAction = MTLStoreActionStore;
descriptor.depthAttachment.clearDepth = 1.0;
descriptor.stencilAttachment.texture = self.depthStencilTexture;
descriptor.stencilAttachment.loadAction = MTLLoadActionClear;
descriptor.stencilAttachment.storeAction = MTLStoreActionStore;
descriptor.stencilAttachment.clearStencil = 0;

descriptor.colorAttachments[MXColorIndexColor].texture = self.colorMap;
descriptor.colorAttachments[MXColorIndexColor].clearColor =
        MTLClearColorMake(0.1, 0.1, 0.2, 1.0);
descriptor.colorAttachments[MXColorIndexColor].loadAction =
        MTLLoadActionClear;
descriptor.colorAttachments[MXColorIndexColor].storeAction =
        MTLStoreActionStore;
descriptor.colorAttachments[MXColorIndexNormal].texture = self.normalMap;
descriptor.colorAttachments[MXColorIndexNormal].clearColor =
        MTLClearColorMake(1, 0, 0, 0);
descriptor.colorAttachments[MXColorIndexNormal].loadAction =
        MTLLoadActionClear;
descriptor.colorAttachments[MXColorIndexNormal].storeAction =
        MTLStoreActionStore;
encoder = [buffer renderCommandEncoderWithDescriptor:descriptor];
```

Next we set the graphics pipeline, the matrices (for transformation), the drawing stencil and other attributes to render our teapot into the G-Buffer:

```
[encoder setRenderPipelineState:self.gPipeline];
```

```
[encoder setVertexBytes:&u length:sizeof(MXUniforms)
atIndex:MXVertexIndexUniforms];
[encoder setDepthStencilState:self.drawStencil];
[encoder setStencilReferenceValue:1];
[encoder setFragmentTexture:self.texture atIndex:MXTextureIndex0];
[encoder setFragmentTexture:self.shadowMap atIndex:MXTextureIndexShadow];
[self renderMesh:self.teapot inEncoder:encoder];
[encoder endEncoding];
```

Rendering the Image from the G-Buffer

At this stage we have the color, normal and shadow information stored in two off-screen textures. We need to perform a second pass in order to render our scene, which performs the lighting calculations on a per-pixel basis.

This is performed as a second rendering pass.

Declare Our Rendering Shaders

In order to render the pixels we will ultimately render a square that covers the entire screen. This will trigger our fragment shader function to render all of the pixels on our screen (that are part of our stencil).

Our vertex shader is very simple: it takes a two-dimensional vertex representing the corners of our screen, and outputs a vertex output structure which contains both the u/v coordinates of our color and normal textures, and the x/y location on our screen.

```
struct VertexIn {
    float2 position     [[attribute(MXAttributeIndexPosition)]];
};

struct VertexOut {
    float4 position     [[position]];
    float2 uv;
    float2 xy;
};

vertex VertexOut vertex_grender(VertexIn v [[stage_in]])
{
    VertexOut out;

    out.position = float4(v.position,0,1);
    out.xy = v.position;
    out.uv = float2((1 + v.position.x)/2,(1 - v.position.y)/2);

    return out;
}
```

> *Note: We pass in the (x,y) parameter because when the position is passed to our fragment shader in the [[position]] parameter, it is scaled to pixel coordinates. So if we wish to preserve the pre-pixel coordinates, we need to pass the parameters in separately.*

We also perform the (u,v) coordinate transformation here in order to accelerate the conversion process to the texture coordinates used to access our color and normal vectors. That's required because while we are passed in the pixel coordinates, we access our textures using coordinates in the range (0,1).

The fragment shader contains all the code we previously used to calculate our diffuse and specular lighting effects:

```
constant float3 lightColor(1,1,1);
constant float ambientIntensity = 0.1;
constant float3 lightDirection(1,0,1);
constant float3 eyeDirection(0,0,1);
constant float specularTightness = 25;
constant float specularIntensity = 0.75;


fragment float4 fragment_grender(VertexOut v [[stage_in]],
        texture2d<float, access::sample> color
                [[texture(MXTextureIndexColor)]],
        texture2d<float, access::sample> normal
                [[texture(MXTextureIndexNormal)]],
        depth2d<float, access::sample> depth
                [[texture(MXTextureIndexDepth)]])
{
    constexpr sampler linearSampler (mip_filter::linear,
                                     mag_filter::linear,
                                     min_filter::linear);


   /*
    *    Get the color and run our color calculations for our basic
    *    lighting effects
    */

   const float3 normalLight = normalize(lightDirection);
   const float3 normalEye = normalize(eyeDirection);

   /*
    *    Get our values from the gbuffer for lighting calculations
    */

   float3 teapotColor = color.sample(linearSampler,v.uv).rgb;
   float shadow = color.sample(linearSampler,v.uv).a;
   float3 teapotNormal = normalize(normal.sample(linearSampler,v.uv).xyz);


   /*
    *    Calculate lighting effects for primary lighting
    */

   // Ambient lighting
   float4 ambient = float4(teapotColor * lightColor * ambientIntensity,
        1.0);
```

```
    // Diffuse lighting
    float dotprod = dot(teapotNormal,normalLight);
    float diffuseIntensity = clamp(dotprod,0,1) * shadow;
    float4 diffuse = float4(teapotColor * lightColor * diffuseIntensity,
         1.0);

    // Specular lighting
    float3 refl = (2 * dotprod) * teapotNormal - normalLight;
    float specIntensity = dot(refl,normalEye);
    specIntensity = clamp(specIntensity,0,1);
    specIntensity = powr(specIntensity,specularTightness) * shadow;
    float4 specular = float4(lightColor * specIntensity *
         specularIntensity,1.0);

    return ambient + diffuse + specular;
}
```

Creating our Stencil Descriptor

We use the stencil in the prior pass to skip processing pixels that do not need to be processed:

```
MTLDepthStencilDescriptor *maskStencilDescriptor =
[[MTLDepthStencilDescriptor alloc] init];
maskStencilDescriptor.depthCompareFunction = MTLCompareFunctionLess;
[maskStencilDescriptor setDepthWriteEnabled:NO];

MTLStencilDescriptor *maskStencil = [[MTLStencilDescriptor alloc] init];
maskStencil.stencilCompareFunction = MTLCompareFunctionEqual;
maskStencilDescriptor.backFaceStencil = maskStencil;
maskStencilDescriptor.frontFaceStencil = maskStencil;

self.maskStencil = [self.device
newDepthStencilStateWithDescriptor:maskStencilDescriptor];
```

Creating our Graphics Pipeline

We also need to create a new graphics pipeline with our shaders:

```
self.grVertexFunction = [self.library
        newFunctionWithName:@"vertex_grender"];
self.grFragmentFunction = [self.library
        newFunctionWithName:@"fragment_grender"];

MTLRenderPipelineDescriptor *grBufferPipelineDescriptor =
        [MTLRenderPipelineDescriptor new];
grBufferPipelineDescriptor.vertexFunction = self.grVertexFunction;
grBufferPipelineDescriptor.fragmentFunction = self.grFragmentFunction;
grBufferPipelineDescriptor.colorAttachments[0].pixelFormat =
        self.colorPixelFormat;
grBufferPipelineDescriptor.vertexDescriptor = fairyDescriptors;
```

```
grBufferPipelineDescriptor.depthAttachmentPixelFormat =
        MTLPixelFormatDepth32Float_Stencil8;
grBufferPipelineDescriptor.stencilAttachmentPixelFormat =
        MTLPixelFormatDepth32Float_Stencil8;

self.grPipeline = [self.device
        newRenderPipelineStateWithDescriptor:grBufferPipelineDescriptor
        error:nil];
```

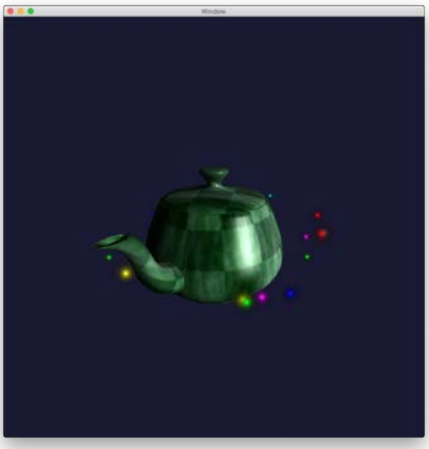
Rendering Our Scene

We can now render our scene using our shader into our screen.

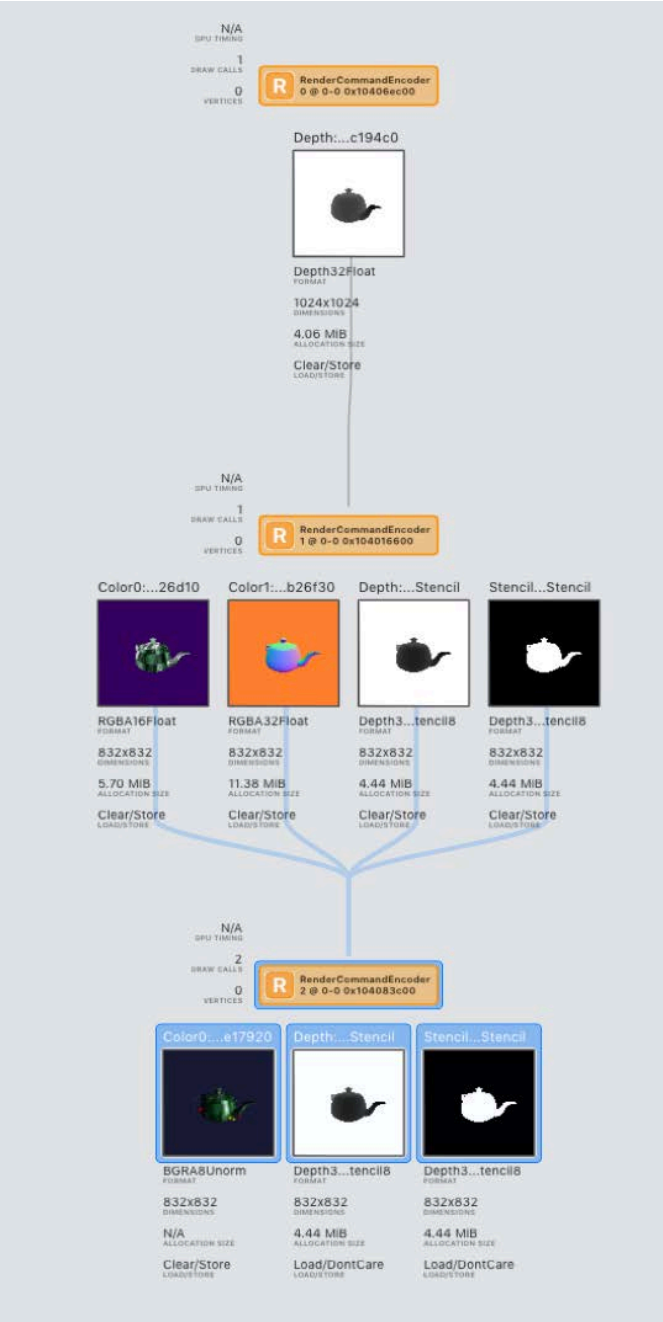
```
descriptor = [view currentRenderPassDescriptor];
descriptor.depthAttachment.loadAction = MTLLoadActionLoad;
descriptor.stencilAttachment.loadAction = MTLLoadActionLoad;
encoder = [buffer renderCommandEncoderWithDescriptor:descriptor];

[encoder setRenderPipelineState:self.grPipeline];
[encoder setStencilReferenceValue:1];
[encoder setVertexBuffer:self.square offset:0
atIndex:MXVertexIndexVertices];
[encoder setDepthStencilState:self.maskStencil];
[encoder setFragmentTexture:self.colorMap atIndex:MXTextureIndexColor];
[encoder setFragmentTexture:self.normalMap atIndex:MXTextureIndexNormal];
[encoder setFragmentTexture:self.depthStencilTexture
atIndex:MXTextureIndexDepth];
[encoder drawPrimitives:MTLPrimitiveTypeTriangle vertexStart:0
vertexCount:6];
```

At this stage, if you were to run our code (available on GitHub) you would see--well pretty much the same thing as you've seen in the prior example above.

That's because we've done a lot of work to separate geometry from shading--but then we've done nothing with our geometry.



This, however, is a good checkpoint to make sure that we are in fact populating and using our G-Buffer correctly. If we use Xcode and look at the debugger we can see the steps our system is using to compose our scene:



This can be a very powerful tool to help visualize the steps used in rendering a scene.

At this point you may be thinking "sure, we've done all this work--but so what?"

The thing is, now that we have the geometry of the scene we can use it to help efficiently illuminate our fairy lights as they dance around our teapot.

For each fairy light we render a square that is (approximately) the range of pixels that would be illuminated by our fairy light. We then calculate the (x,y,z) location of our teapot by using an inverse transformation, and use that information to calculate how brightly the fairy light will illuminate the surface.

Adding the Glowing Lights

For us to properly calculate the intensity of the glow on the teapot added by a fairy light, we need to obtain the (x,y,z) position of a pixel on the screen. There are two ways we can handle this. The first is by storing the (x,y,z) location of each pixel in our G Buffer. The advantage is that this is extremely fast--but it has the disadvantage of requiring significantly more memory.

The other way is to use an inverse transformation matrix to translate from screen space back to model space. This has the disadvantage of requiring a full matrix multiply per pixel. We will use the second option.

Adding the View Inverse

We add the inverse of the view matrix to our uniforms.

```
typedef struct MXUniforms
{
    matrix_float4x4 model;
    matrix_float4x4 view;
    matrix_float4x4 inverse;    // inverse of model
    matrix_float4x4 vinverse;   // inverse of view
    matrix_float4x4 shadow;     // matrix for shadow mapping
} MXUniforms;
```

Adding the Shader Functions

The shader functions for our fairy light illumination shaders calculates the amount each fairy light illuminates our teapot. The vertex shader provides the position of each light in model space, the screen position and coordinates on our G-Buffer texture map, and the color of each fairy light. It's similar to our fairy light vertex shader, but it passes more information so we can calculate our illumination.

Note that each fairy light illumination area is constrained by an area about 5 times bigger than the fairy light itself. The assumption is that the visible part of the illumination from our lights will only contribute to a finite volume around our fairly light--so we do not need to perform a shading calculation for the entire teapot--just a small area around the light where the teapot is close to the light.

```
vertex VertexOut vertex_illumination(VertexIn v [[stage_in]],
        constant MXUniforms &u [[buffer(MXVertexIndexUniforms)]],
        constant FairyLightIn *positions
                [[buffer(MXVertexIndexLocations)]],
        uint iid [[instance_id]])
{
   VertexOut out;
```

```
    // Transform location and offset
    float4 worldPos = u.model * float4(positions[iid].position,1.0);
    float4 screenPos = u.view * worldPos;
    // Offset the X/Y location. Note this is 5 times bigger than with
    // the fairy light shaders.
    screenPos.xy += v.position * 0.15 * positions[iid].size;
    out.position = screenPos;

    float2 screenXY = screenPos.xy / screenPos.w;

    // Pass through position, adjust for screen texture. Pass color
    out.lightPos = worldPos.xyz;
    out.xy = screenXY;
    out.uv = float2((1 + screenXY.x)/2,(1 - screenXY.y)/2);
    out.color = positions[iid].color;
    return out;
}
```

The illumination fragment shader function calculates the (x,y,z) position of each point on the screen, giving us the position of the point on the teapot. It then calculates the square of the distance (to attenuate the light as the point gets farther) as well as the dot product between the vertex pointing to the light and the normal of the teapot.

These are then used to calculate the illumination added by the light (by virtue of changing the alpha channel of the returned color).

```
fragment float4 fragment_illumination(VertexOut v [[stage_in]],
        constant MXUniforms &u [[buffer(MXVertexIndexUniforms)]],
        texture2d<float, access::sample> color
                [[texture(MXTextureIndexColor)]],
        texture2d<float, access::sample> normal
                [[texture(MXTextureIndexNormal)]],
        depth2d<float, access::sample> depth
                [[texture(MXTextureIndexDepth)]])
{
    constexpr sampler linearSampler (mip_filter::linear,
                                     mag_filter::linear,
                                     min_filter::linear);

    float z = depth.sample(linearSampler, v.uv);
    float4 pos = float4(v.xy,z,1);
    float4 spos = u.vinverse * pos;
    float3 ppos = spos.xyz / spos.w;

    /*
     *    Grab color, normal
     */

    float3 teapotNormal = normalize(normal.sample(linearSampler,v.uv).xyz);

    /*
     *    Find distance to the light
```

```
     */

    float3 delta = v.lightPos - ppos;
    float r2 = dot(delta,delta) * 100;
    if (r2 < 1) r2 = 1;
    else r2 = 1/r2;
    if (r2 < 0.01) r2 = 0;

    float3 posvec = normalize(delta);
    float diffuse = clamp(dot(posvec,teapotNormal) * r2, 0, 1);

    return float4(v.color,diffuse);
}
```

Create the Illumination Pipeline

Of course with new shader functions we need a new pipeline state. We also need to set the pipeline state for alpha blending, as our lights are blended with the color of the teapot.

```
MTLRenderPipelineDescriptor *fairyLightPipelineDescriptor =
        [MTLRenderPipelineDescriptor new];
fairyLightPipelineDescriptor.vertexFunction =
        self.fairyLightVertexFunction;
fairyLightPipelineDescriptor.fragmentFunction =
        self.fairyLightFragmentFunction;
fairyLightPipelineDescriptor.vertexDescriptor = fairyDescriptors;
fairyLightPipelineDescriptor.depthAttachmentPixelFormat =
        self.depthStencilPixelFormat;
fairyLightPipelineDescriptor.stencilAttachmentPixelFormat =
        self.depthStencilPixelFormat;
fairyLightPipelineDescriptor.colorAttachments[0].pixelFormat =
        self.colorPixelFormat;

fairyLightPipelineDescriptor.colorAttachments[0].blendingEnabled = YES;
fairyLightPipelineDescriptor.colorAttachments[0].rgbBlendOperation =
        MTLBlendOperationAdd;
fairyLightPipelineDescriptor.colorAttachments[0].alphaBlendOperation =
        MTLBlendOperationAdd;
fairyLightPipelineDescriptor.colorAttachments[0].sourceRGBBlendFactor =
        MTLBlendFactorSourceAlpha;
fairyLightPipelineDescriptor.colorAttachments[0].sourceAlphaBlendFactor =
        MTLBlendFactorSourceAlpha;
fairyLightPipelineDescriptor.colorAttachments[0].destinationRGBBlendFactor
= MTLBlendFactorOneMinusSourceAlpha;
fairyLightPipelineDescriptor.colorAttachments[0].destinationAlphaBlendFact
or = MTLBlendFactorOneMinusSourceAlpha;

self.fairyLightPipeline = [self.device
        newRenderPipelineStateWithDescriptor:fairyLightPipelineDescriptor
        error:nil];
```

Create the Depth Stencil State

We also need to create a new depth stencil state. The new state should ignore the z-buffer, but mask against the stencil (which is set only where the teapot is being drawn). This allows us to calculate the illumination for each light as it is added to the teapot.

Also note that we ignore the z-buffer depth because the visibility components of our calculations have already been performed.

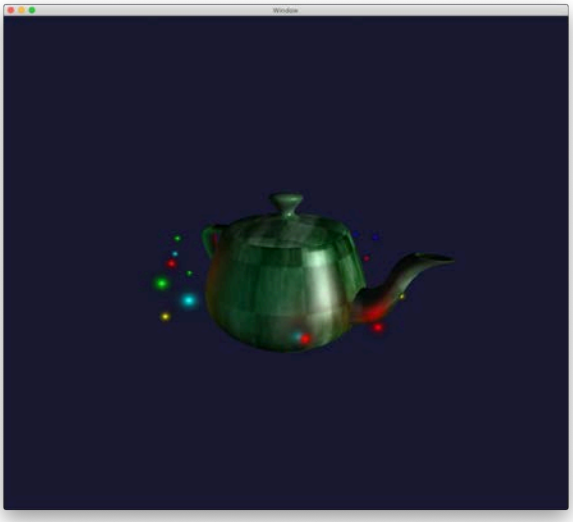
```
MTLDepthStencilDescriptor *illuminationStencilDescriptor =
        [[MTLDepthStencilDescriptor alloc] init];
illuminationStencilDescriptor.depthCompareFunction =
        MTLCompareFunctionAlways;
[illuminationStencilDescriptor setDepthWriteEnabled:NO];

MTLStencilDescriptor *illuminationStencil = [[MTLStencilDescriptor alloc]
        init];
illuminationStencil.stencilCompareFunction = MTLCompareFunctionEqual;
illuminationStencilDescriptor.backFaceStencil = illuminationStencil;
illuminationStencilDescriptor.frontFaceStencil = illuminationStencil;

self.illuminationStencil = [self.device
        newDepthStencilStateWithDescriptor:illuminationStencilDescriptor];
```

Rendering the Illumination

We render the illumination from our sprites after we've rendered the teapot but before the fairy lights themselves. The code passes in our G-Buffer to our shader so the appropriate lighting calculations can be performed.

```
[encoder setRenderPipelineState:self.fairyLightPipeline];
[encoder setDepthStencilState:self.illuminationStencil];
[encoder setVertexBuffer:self.square
                  offset:0
                 atIndex:MXVertexIndexVertices];
[encoder setVertexBytes:&u
                 length:sizeof(MXUniforms)
                atIndex:MXVertexIndexUniforms];
[encoder setVertexBytes:fairyLights
                 length:sizeof(fairyLights)
                atIndex:MXVertexIndexLocations];
[encoder setFragmentTexture:self.colorMap
                    atIndex:MXTextureIndexColor];
[encoder setFragmentTexture:self.normalMap
                    atIndex:MXTextureIndexNormal];
[encoder setFragmentTexture:self.depthStencilTexture
                   atIndex:MXTextureIndexDepth];
[encoder drawPrimitives:MTLPrimitiveTypeTriangle
            vertexStart:0
            vertexCount:6
          instanceCount:MAX_FAIRYLIGHTS];
```

Once we make the changes (which are also available on GitHub), we should see:



The effect is subtle but noticeable.

Updating the Fairy Lights using a Compute Kernel

The next example uses a compute kernel to update the location of the fairy lights. While this is an extremely simplistic example, it does show how we can mix a compute kernel (where the GPU performs a complex calculation) with rendering.

The point of this demo is to show creating a compute kernel and mixing it with a rendering kernel. The calculation of the lights around the teapot is exceedingly trivial, but demonstrates a process that can be applied to a much more computationally interesting simulation.

Creating the Compute Kernel

The compute kernel is executed once per thread, in the same way that a vertex function is executed once per vertex or a fragment function is executed once per pixel.

Adding Elapsed Time to the Uniforms

Our compute kernel takes in the array of fairy light locations, and updates the position depending on the elapsed time. The elapsed time is passed in as part of our uniforms structure:

```
typedef struct MXUniforms
{
    float aspect;
    float elapsed;
    matrix_float4x4 model;
    matrix_float4x4 view;
```

```
    matrix_float4x4 inverse;    // inverse of model
    matrix_float4x4 vinverse;   // inverse of view
    matrix_float4x4 shadow;     // matrix for light position
} MXUniforms;
```

The Fairy Light Kernel

The kernel calculates the location of each of our fairy lights. It's basically the code we had used previously to compute the location of our lights:

```
kernel void fairy_kernel(device MXFairyLocation *positions
                [[buffer(MXVertexIndexLocations)]],
        constant MXUniforms &u
                [[buffer(MXVertexIndexUniforms)]],
        uint ix [[thread_position_in_grid]])
{
    device MXFairyLocation *loc = positions + ix;

    float a = loc->angle[0] + loc->speed * u.elapsed;

    float sx = sin(a);
    float cx = cos(a);
    float sy = sin(loc->angle[1]);
    float cy = cos(loc->angle[1]);

    loc->position = float3(cx * cy * loc->radius,
                           sy * loc->radius,
                           sx * cy * loc->radius);
}
```

Note: the attribute [[thread_position_in_grid]] gives the index of the kernel in a 1D, 2D or 3D array of threads. There are other techniques that can be used to refer to the location of a compute thread.

Initializing for Our Compute Pipeline

We need to update our fairy light locations so they are stored in the GPU. We also need to create our compute pipeline to update our locations.

Copying the Fairy Light Locations to the GPU

We create a new MTLBuffer to store the data used by our compute kernel:

```
self.fkBuffer = [self.device newBufferWithBytes:fairyLights
                                         length:sizeof(fairyLights)
                                        options:MTLResourceOptionCPUCacheModeDefault];
```

Creating the Compute Pipeline State

Creating a compute pipeline is significantly simpler than a rendering pipeline, as our compute pipeline only requires the compute kernel function.

```
self.fkFunction = [self.library newFunctionWithName:@"fairy_kernel"];
self.fkPipeline = [self.device
        newComputePipelineStateWithFunction:self.fkFunction error:nil];
```

Executing the Compute Kernel

Instead of running our calculations on the CPU then copying the results into the GPU per frame, we can now run the calculation code on our GPU.

Obtaining the Compute Command Encoder

We first obtain a MTLComputeCommandEncoder, which like its MTLRenderCommandEncoder is used to initialize the parameters used to run our compute kernel. Our compute kernel does not require a descriptor to obtain, so the call is very simple:

```
compute = [buffer computeCommandEncoder];
```

Setting the Parameters

The compute command encoder must be passed in our parameters: the array of lights whose positions need updating, and the uniform (which contains our elapsed time). We must also provide a reference to our compute pipeline state:

```
[compute setBuffer:self.fkBuffer
            offset:0
           atIndex:MXVertexIndexLocations];
[compute setBytes:&u
           length:sizeof(MXUniforms)
          atIndex:MXVertexIndexUniforms];
[compute setComputePipelineState:self.fkPipeline];
```

Selecting our Thread Group Size and our Threads Per Group Parameters

For this example, we are processing a one-dimensional array of lights, and the number of lights is relatively small. So for our sample code we simply set the group size to 1 and the threads per group size to the number of lights in our system. Our light locations are then updated in a single pass.

```
MTLSize threadGroupSize = MTLSizeMake(MAX_FAIRYLIGHTS, 1, 1);
MTLSize threadsPerGroup = MTLSizeMake(MAX_FAIRYLIGHTS, 1, 1);
```

In general threads for our compute process are arranged in a 3-dimensional array of threads. Threads are further organized in groups, with each group of threads executed as as group, potentially sharing group-

local memory. We need to determine the size of our thread groups, and also pass in the total size of all of the threads we need for our compute loop.

The trick is that the maximum number of threads in our three-dimensional thread group can be no bigger than the value in our MTLComputePipelineState's maxTotalThreadsperThreadgroup method.

Executing our Compute Kernel

Once we know the thread group and threads per group sizes, we can now execute our compute kernel:

```
[compute dispatchThreads:threadsPerGroup
   threadsPerThreadgroup:threadGroupSize];
[compute endEncoding];
```

Using The Results

After our kernel has executed the results are stored in our *fkBuffer* buffer. We can now update the places in our code which uses the contents of our buffer (rather than copying the data to our GPU) by rewriting our code from:

```
[encoder setVertexBytes:fairyLights
                 length:sizeof(fairyLights)
                atIndex:MXVertexIndexLocations];
```

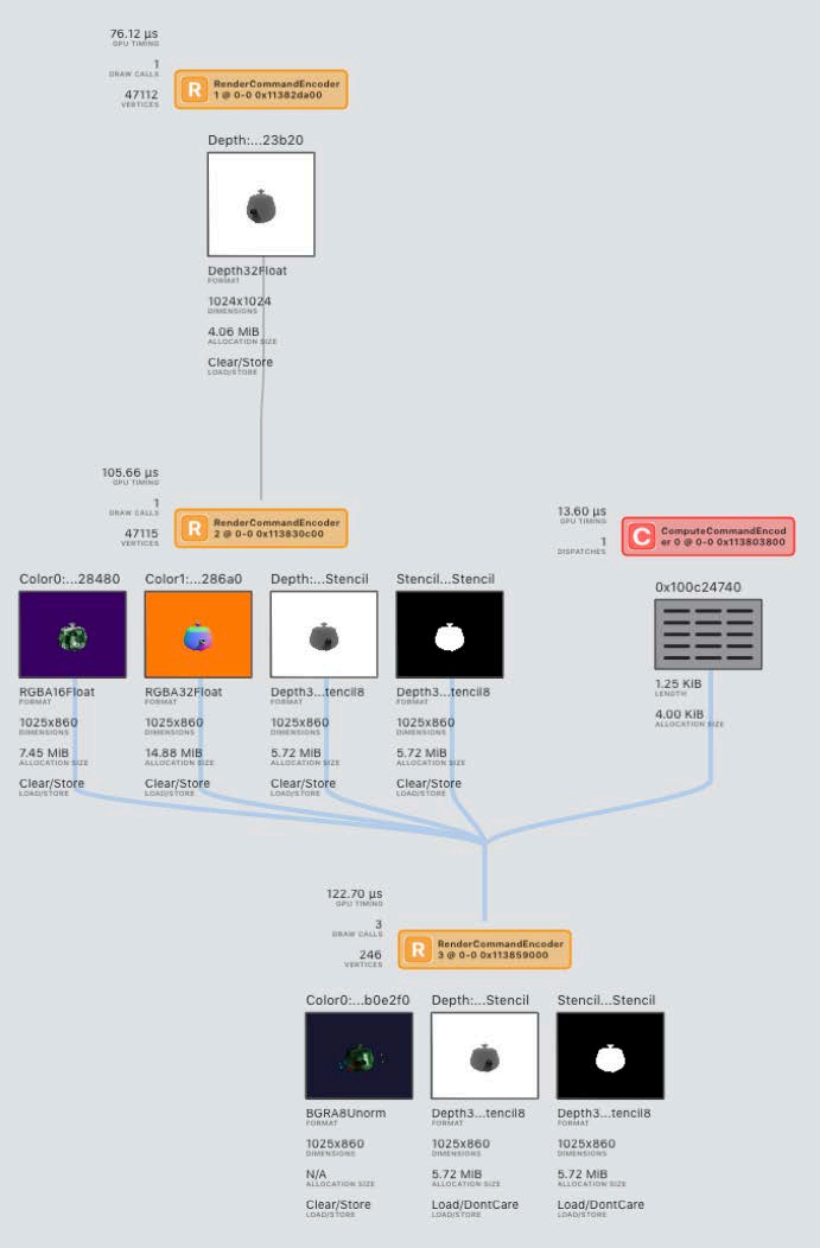
to:

```
[encoder setVertexBuffer:self.fkBuffer
                  offset:0
                 atIndex:MXVertexIndexLocations];
```

We don't need to make any other changes to our fairy light rendering code.

Once we've made the changes noted above (or on GitHub), our code should give the same results as in our prior example.

However, if we look at the process of rendering a frame in our debugger we can see how our compute kernel is now added to our rendering process:



Constructive Solid Geometry

Image-based constructive solid geometry can be implemented in Metal. By using the modified Goldfeather algorithm we can render complex shapes, representing intersections, differences and unions in real time.

In order to perform our operations we must create a separate z-buffer for temporary storage, as well as creating separate stencil buffers. Further, we must perform two full passes, obtaining the results of a first rendering pass before submitting the second.

Note: This may not be the most efficient way to perform CSG on Metal. However, it does work and it does prevent GPU/CPU copying of large buffers which permits real-time rendering.

This is also a demonstration of the steps used in Metal to render a CSG product. Any application using these steps would need to repeat them for each product, which may involve multiple buffer requests for a complex object.

A Brief Description of the Algorithm.

The modified Goldfeather algorithm operates in two principle passes. The first reorganizes the intersections, unions and differences into a collection of terms with unions at the top of the evaluation tree. The steps here are not discussed, only that at the end of the process results in a collection of products which represent an intersection or difference.

The second step involves rendering each product a layer at a time, with each "layer" indicating a part of the model which is visible--either the frontside or backside of each primitive drawn depending if the primitive is being subtracted or if its part of a difference. This means we must first perform a pass to count the maximum number of layers that are to be rendered.

Once we have the depth, we then iterate through each layer, rendering the primitives in order to make the front or back sides visible as appropriate depending on the operation.

CSG Operation Tree Normalization

We do not describe the process of tree normalization, but assume the contents are rendered in normal form. See the paper above for more information about CSG tree normalization.

What is important is that when we render our scene we can rely on the objects in our scene being a union of products P, where each product is an ordered list of one or more 3D objects that are either differenced or subtracted.

Layer Counting

The first step of our rendering process requires that we count the number of visible layers in our image. That is, for every pixel in our scene we determine the number of rendering objects that overlap on that pixel. This will be used for "layer extraction" as our algorithm scans the different rendering layers determining the appropriate surface to render on that layer.

One note about layer counting. Because we know that that the intersection of two geometric objects will show the front surfaces, while subtracting shows the back surfaces, we can use face culling to reduce the number of layers displayed.

We count layers by using stencils. Our layer counting algorithm does not use the depth buffer.

Layer Extraction

The next step is to render each of the layers into the depth buffer, then render the surfaces that are in front of (or equal to) the calculated depth buffer. We do this for each of the surfaces in a particular product, and we use even/odd pixel parity in order to determine if the inside or outside of a surface may be visible at a particular location.

Product Merging

Once we've rendered a layer, we merge the layers into a second destination z-buffer, forming the final union of objects being displayed by our system.

Algorithm Description

```
Given a list of products P (found during tree normalization)
Produce a rendered image representing our CSG operation

Initialize output z-buffer Z_out to z-far.
Initialize output color-buffer C_out to clear color.

For each product P in our normalized CSG operation tree
    Find the number of layers k_max in P
    For each layer k in k_max
        Initialize intermediate z-buffer Z_i to z-far.
        Initialize intermediate color-buffer c_i to clear color.
        Render k'th layer's z-buffer into z-far using stencil counting

        For each primitive A in P
            Clear stencil buffer
            Render A into z-buffer/stencil buffer with parity test and
                z-depth testing to <= z_i, and no face tests.
                (Note: parity testing flips a single bit if the pixel is
                visible.)
            if A is subtracted
                Clear all odd-parity pixels. (Only accept even parity
                pixels)
            else
                Clear all even-parity pixels. (Only accept odd parity
                pixels)
        End

        Merge Zi, Ci into Z_out C_out with < Z_out compare test.
    End For
End For

Display C_out.
```

Metal Features Demonstrated

Because of the complexity of the CSG rendering algorithm, this sample code demonstrates several features of Metal.

First, it demonstrates how we use semaphores in order to control the execution of multiple MTLCommandBuffer objects when rendering a single image frame. Second, it shows the use of an intermediate result from the first command buffer rendering request to construct a second rendering request. Third, it shows a rather complex system of rendering objects into off-screen buffers to handle various rendering effects.

> *Note: Recall from our discussion of the Metal architecture that our GPU doesn't even see the commands we ask to be executed until the MTLCommandBuffer is submitted for rendering to the GPU. This means if we need to get the results of a rendering operation to our CPU for further processing, we must submit the buffer to the GPU and wait for execution to complete.*

Coordinating Command Buffers

Because we are executing our rendering process using two buffers, with the results of the first set of operations loaded into our CPU driving the second set of operations, we need to set a semaphore to prevent our system from making unnecessary rendering calls.

Declare a Semaphore and Initialize It.

We declare our semaphore as part of our view:

```
@property (strong) dispatch_semaphore_t semaphore;
```

We then initialize our semaphore to pass through only 1 thread at a time.

```
self.semaphore = dispatch_semaphore_create(1);
```

Assuring Only One Thread at a Time Passes through drawInMTKView

We then test to make sure only one thread at a time passes through the rendering thread by inserting a wait test:

```
dispatch_semaphore_wait(self.semaphore, DISPATCH_TIME_FOREVER);
```

On the second buffer execution we then clear the semaphore (allowing another thread to render the display contents) by creating a completion handler on the second buffer:

```
[buffer addCompletedHandler:^(id<MTLCommandBuffer> cmdBuffer) {
    dispatch_semaphore_signal(self.semaphore);
}];
```

Finding the Number of Layers k_{max}

The first pass of our algorithm is to build the count of the total number of layers. This calculates a stencil map where each pixel location has the number of layers drawn at the pixel. We need to keep the stencil (as it will be used in the second step) as well as determine the maximum number of levels for our rendering phase.

Counting our Pixels

We create a separate stencil texture in *self.stencilTexture* because we will eventually pass our stencil to a compute kernel to rapidly scan the texture for the maximum value. Our *layerCountPipeline* only does a stencil count without any depth testing, though it uses the vertex function for 3D rendering of our objects.

```
pipelineDescriptor = [MTLRenderPipelineDescriptor new];
pipelineDescriptor.vertexFunction = vertexFunction;
pipelineDescriptor.fragmentFunction = nil;
pipelineDescriptor.vertexDescriptor =
        MTKMetalVertexDescriptorFromModelIO(d);
pipelineDescriptor.stencilAttachmentPixelFormat =
        MTLPixelFormatStencil8;

self.layerCountPipeline = [self.device
        newRenderPipelineStateWithDescriptor:pipelineDescriptor
        error:nil];
```

The layer count stencil simply increments the stencil for every pixel drawn:

```
stencil = [[MTLStencilDescriptor alloc] init];
stencil.stencilCompareFunction = MTLCompareFunctionAlways;
stencil.depthStencilPassOperation = MTLStencilOperationIncrementClamp;

descriptor = [[MTLDepthStencilDescriptor alloc] init];
descriptor.depthCompareFunction = MTLCompareFunctionAlways;
descriptor.backFaceStencil = stencil;
descriptor.frontFaceStencil = stencil;
descriptor.depthWriteEnabled = NO;
self.layerCountStencil = [self.device
        newDepthStencilStateWithDescriptor:descriptor];
```

Our rendering pass is explicitly written. Our geometry consists of the difference between a cube and sphere, with the cylinder objects subtracted from the results. For our rendering process we render the front of the cube and sphere, and the back of the cylinder objects. (Essentially because the cylinder "scoops out" from our object--meaning you'll only see the back side of the cylinders as they scoop content from the cube/sphere intersection.)

```
[encoder setCullMode:MTLCullModeFront];
[self renderMesh:self.cube inEncoder:encoder];
[self renderMesh:self.sphere inEncoder:encoder];
[encoder setCullMode:MTLCullModeBack];
[self renderMesh:self.cylinders inEncoder:encoder];
```

Scanning the Results to Find the Largest Value

Once we've rendered the pixel count into our stencil buffer we use a compute kernel to quickly add the columns of stencil counts. We do this by selecting a thread group size that is 1 thread high; this will cause our GPU to sweep down through the stencil buffer without any thread contention when updating the count size.

The summary is then small enough (one byte per pixel width) to simply copy back to the CPU.

Our layer count kernel is dead simple:

```
kernel void layer_count(texture2d<ushort, access::read> stencil
        [[texture(0)]],
                        uint2 ix [[thread_position_in_grid]],
                        device uint8_t *c [[buffer(1)]])
{
    ushort val = stencil.read(ix).x;
    if (c[ix.x] < val) c[ix.x] = val;
}
```

We execute the compute kernel with a sums buffer that will store the results:

```
threadGroupSize = MTLSizeMake(width, [self.stencilTexture height], 1);
s = self.countPipeline.maxTotalThreadsPerThreadgroup;
threadsPerGroup = MTLSizeMake(s, 1, 1);
[compute dispatchThreads:threadGroupSize
   threadsPerThreadgroup:threadsPerGroup];
```

Once we've finished the compute kernel request, we close the buffer and submit it to the GPU with a completion handler that finds the maximum value in the column list, and calls the code to handle part two of our rendering process.

> *Note: Remember that our commands are not actually executed by the GPU until after we commit our command buffer.*

```
[buffer addCompletedHandler:^(id<MTLCommandBuffer> cmdBuffer) {
    uint8_t klen = 0;
    uint8_t *buf = (uint8_t *)self.sumsBuffer.contents;
    for (NSInteger i = 0; i < width; ++i) {
        if (klen < buf[i]) klen = buf[i];
    }

    [self renderPhaseTwoWithKLen:klen];
}];

[buffer commit];
```

Rendering The Products

The second phase takes the count from the first phase and uses it to render our product.

Clear the Output Z-Buffer and Color Buffer

We first clear the output z-buffer and color buffer, both stored as separate texture buffers created in the mtkView:drawableSizeWillChange: method by using a compute kernel.

> *Note: This operation would only be done once, and for a full CSG implementation would be done before the first product layer count is performed. But for this demo we perform the initialization in the second phase.*

Our kernel function is trivial:

```
kernel void layer_cleardepth(texture2d<float, access::write> outColor
        [[texture(MXTextureIndexOutColor)]],
                              texture2d<float, access::write> outDepth
        [[texture(MXTextureIndexOutDepth)]],
                         uint2 index [[thread_position_in_grid]])
{
    outColor.write(float4(0,0,0,1),index);
    outDepth.write(1.0,index);
}
```

Build the Intermediate Z-Buffer For The Kth Layer

We render our intermediate z-buffer and color buffer for the kth layer inside of a for loop. The rendering uses a stencil to count the number of layers drawn, and while we update the depth buffer we do not use depth buffer testing.

> *Note: An integral part of this algorithm is that all products and all primitives in a product are rendered in the same order every time.*

The stencil testing used is:

```
stencil = [[MTLStencilDescriptor alloc] init];
stencil.stencilCompareFunction = MTLCompareFunctionEqual;
stencil.depthStencilPassOperation = MTLStencilOperationIncrementClamp;
stencil.stencilFailureOperation = MTLStencilOperationIncrementClamp;
stencil.depthFailureOperation = MTLStencilOperationIncrementClamp;

descriptor = [[MTLDepthStencilDescriptor alloc] init];
descriptor.depthCompareFunction = MTLCompareFunctionAlways;
descriptor.backFaceStencil = stencil;
descriptor.frontFaceStencil = stencil;
descriptor.depthWriteEnabled = YES;
self.layerExtractStencil = [self.device
        newDepthStencilStateWithDescriptor:descriptor];
```

Note that our stencil always increments the stencil count, but only populates our depth stencil when the stencil count in a pixel matches our target value in our loop.

Draw Our Primitives, Clearing Pixels that are Not Visible

The next step is to render each primitive in the same order as before, without any surface culling, and using a depth test that renders pixels if less than or equal to our current depth, tracking the parity (that is, if the number of surfaces rendered is even or odd).

Our rendering step does not update the depth buffer; we're only interested in counting the number of surfaces are in front of or equal to the current surface:

```
pipelineDescriptor = [MTLRenderPipelineDescriptor new];
pipelineDescriptor.vertexFunction = vertexFunction;
pipelineDescriptor.fragmentFunction = nil;
pipelineDescriptor.vertexDescriptor =
        MTKMetalVertexDescriptorFromModelIO(d);
pipelineDescriptor.colorAttachments[0].pixelFormat =
        self.colorPixelFormat;
pipelineDescriptor.colorAttachments[0].writeMask = MTLColorWriteMaskNone;
pipelineDescriptor.stencilAttachmentPixelFormat =
        MTLPixelFormatDepth32Float_Stencil8;
pipelineDescriptor.depthAttachmentPixelFormat =
        MTLPixelFormatDepth32Float_Stencil8;

self.layerParityPipeline = [self.device
        newRenderPipelineStateWithDescriptor:pipelineDescriptor
        error:nil];
```

Our stencil flips a single bit to track if the number of pixels rendered is even or odd:

```
stencil = [[MTLStencilDescriptor alloc] init];
stencil.depthStencilPassOperation = MTLStencilOperationInvert;
stencil.stencilFailureOperation = MTLStencilOperationKeep;
stencil.depthFailureOperation = MTLStencilOperationKeep;
stencil.readMask = 1;
stencil.writeMask = 1;

descriptor = [[MTLDepthStencilDescriptor alloc] init];
descriptor.depthCompareFunction = MTLCompareFunctionLessEqual;
descriptor.depthWriteEnabled = NO;
descriptor.backFaceStencil = stencil;
descriptor.frontFaceStencil = stencil;
self.layerParityStencil = [self.device
        newDepthStencilStateWithDescriptor:descriptor];
```

We render each primitive. (This shows the first object being rendered.)

```
[encoder setRenderPipelineState:self.layerParityPipeline];
[encoder setDepthStencilState:self.layerParityStencil];
[encoder setStencilReferenceValue:1];
[encoder setVertexBuffer:self.uniforms
                  offset:0
                 atIndex:MXVertexIndexUniforms];
[encoder setCullMode:MTLCullModeNone];
[self renderMesh:self.cube inEncoder:encoder];
```

We then perform a second pass, clearing all pixels that do not match our even/odd test. (This shows the first object being cleared.)

```
[encoder setRenderPipelineState:self.layerClearPipeline];
[encoder setDepthStencilState:self.layerClearStencil];
[encoder setStencilReferenceValue:1];  // Odd: not subtracted
[encoder setVertexBuffer:self.square
                  offset:0
                 atIndex:MXVertexIndexVertices];
[encoder setFragmentBytes:&color
                   length:sizeof(color)
                  atIndex:MXFragmentIndexColor];
[encoder drawPrimitives:MTLPrimitiveTypeTriangle
            vertexStart:0
            vertexCount:6];
```

We repeat this sequence for each of our three primitives.

Merging the Intermediate Depth/Color Buffer into the Output Depth/Color Buffer

For merging our intermediate and output depth and color buffers we use a compute kernel. The compute kernel is relatively simple:

```
kernel void layer_merge(texture2d<float, access::read> inColor
        [[texture(MXTextureIndexInColor)]],
                        depth2d<float, access::read> inDepth
        [[texture(MXTextureIndexInDepth)]],
                        texture2d<float, access::write> outColor
        [[texture(MXTextureIndexOutColor)]],
                        texture2d<float, access::read_write> outDepth
        [[texture(MXTextureIndexOutDepth)]],
                        uint2 index [[thread_position_in_grid]])
{
    float4 inc = inColor.read(index);
    float ind = inDepth.read(index);
    float curd = outDepth.read(index).r;

    if (ind < curd) {
        outDepth.write(ind,index);
        outColor.write(inc,index);
    }
}
```

We then invoke our compute kernel at the bottom of the loop, accumulating the results in our output buffer.

```
compute = [buffer computeCommandEncoder];
[compute setComputePipelineState:self.layerMergePipeline];
[compute setTexture:self.colorTexture atIndex:MXTextureIndexInColor];
[compute setTexture:self.depthTexture atIndex:MXTextureIndexInDepth];
[compute setTexture:self.outTexture atIndex:MXTextureIndexOutColor];
[compute setTexture:self.screenDepth atIndex:MXTextureIndexOutDepth];
```

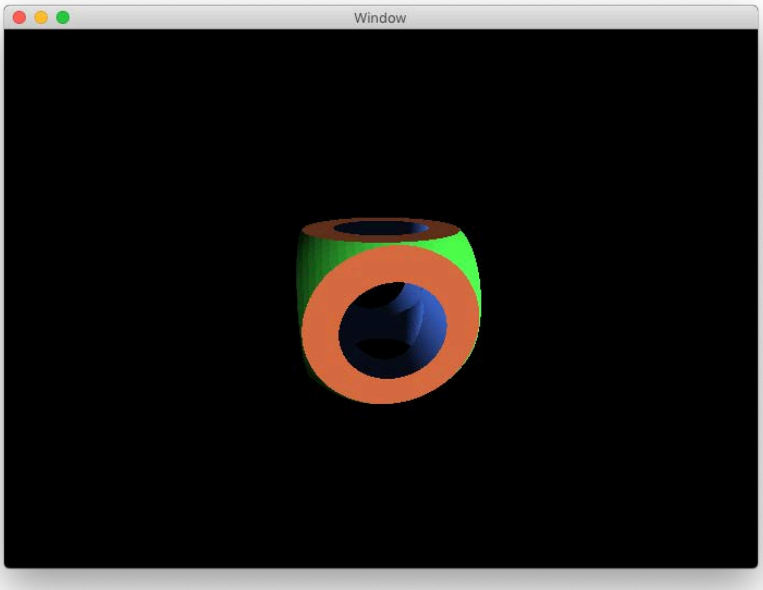
```
MTLSize threadGroupSize = MTLSizeMake([self.colorTexture width],
        [self.colorTexture height], 1);
uint s = sqrt(self.layerMergePipeline.maxTotalThreadsPerThreadgroup);
MTLSize threadsPerGroup = MTLSizeMake(s, s, 1);

[compute dispatchThreads:threadGroupSize
   threadsPerThreadgroup:threadsPerGroup];
[compute endEncoding];
```

Displaying the Results

Once the results are accumulated into our output buffer we render them to the screen as our final step:

```
descriptor = self.currentRenderPassDescriptor;
encoder = [buffer renderCommandEncoderWithDescriptor:descriptor];

[encoder setRenderPipelineState:self.outputResultPipeline];
[encoder setVertexBuffer:self.square offset:0
atIndex:MXVertexIndexVertices];
[encoder setFragmentTexture:self.outTexture
                    atIndex:MXTextureIndexInColor];
[encoder drawPrimitives:MTLPrimitiveTypeTriangle
            vertexStart:0
            vertexCount:6];
[encoder endEncoding];
```

Once all this work is done (the code is in GitHub), we should see:



Work That Needs To Be Done.

In order to turn this into a working CSG rendering system for Metal, some additional work needs to be done.

Normalizing the CSG Operation Tree

The first thing that needs to be done is the work to normalize the CSG operation tree into normal form for our algorithm. This also includes creating bounding boxes for our primitives and doing intersection tests so that unnecessary rendering is not performed. Further, any product P which only has one primitive in it can be rendered into the destination buffer directly.

Looping Across All Products

Second, we would need to create a loop to loop across all products in our normalized tree. Our rendering example above only shows the rendering of a single product. But we would need to do a few things (such as moving the initialization of the output z-buffer and color buffer to the top of the loop and tracking multiple buffers as our product is rendered) in order to turn this into a final product.

Shaders and Metal Functions

The core element of Metal which makes it so powerful is the ability to write programs that run directly on the GPU.

The Metal language is based on the C++14 language specification, with some key differences, mostly arranged around the needs of compiling for a GPU.

> *Note: This is not a complete description of the Metal Shader Language. This describes only a subset of the language as a sort of "introduction" to the highlights of the language.*

About GPUs

A GPU is a specialized processing unit that contains specialized computing hardware for the vector processing necessary for 3D rendering (including manipulating 4x4 matrices, vectors and arrays of color maps) along with dozens or even hundreds of separate processors which can perform a collection of math operations simultaneously. (As an example, each of the two built-in GPUs on the 2013 Mac Pro can execute up to 1024 separate threads simultaneously.)

Threads and Thread Groups

GPU processors generally execute a massive number of short programs simultaneously, organized around the notion of a "thread". Each "thread" executes a short program--generally a few dozen lines of code performing a particular mathematical operation such as calculating the color of a pixel in an image.

Threads can be grouped in "thread groups"--groups of threads which work on a block of data simultaneously, and threads are generally organized as either 1 dimensional, 2 dimensional, or 3 dimensional arrays, depending on the nature of the compute task. For example, an image processing kernel which converts an image from color to black and white may use a 2-dimensional array of threads to correspond to the 2-dimensional array of pixels in your image.

SIMD Groups

Further, threads are organized in "SIMD Groups". The idea is that since generally most compute functions execute the same code path, it is advantageous to have a single processor sequence instructions for a very wide vector or array of inputs and outputs at the same time. That is, if you have a program:

```
uint8_t program1(uint8_t a, uint8_t b)
{
    return a * 2 + b;
}
```

A processor which can process 256-bit wide vectors as a single operation can reorganize the code behind this function so that it can essentially run 32 calls to program1() simultaneously by packing 32 8-bit inputs into the 256-bit wide vector, and run the operation to a 256-bit wide output with 32 simultaneous results.

This is key to understanding why conditional branches can be so complicated to handle in a GPU, and why conditional statements may not necessarily save any compute cycles. If you have the program:

```
uint8_t program2(uint8_t a, uint8_t b)
{
    if (a < 32) {
        return b * 2;         // Statement A
    } else {
        return b * 2 + a;     // Statement B
    }
}
```

The problem is that if even 1 of our 32 inputs take the first branch and the other 31 inputs take the second branch, our SIMD instructions wind up being organized so that both statements A and B have to be executed. (That is, we get no savings by skipping instructions.)

Of course in a massively parallel processor the savings of running things in parallel win out over the inefficiency of having to execute both branches of an instruction. But it is worth remembering, since each branch potentially adds further complexity when writing code that runs on a GPU, since the GPU has to track the results of both branches.

And in the process the number of potential parallel calls that can be made may drop from 32 to 16 as additional bits in our 256 bit wide vector has to be used for branch housekeeping purposes.

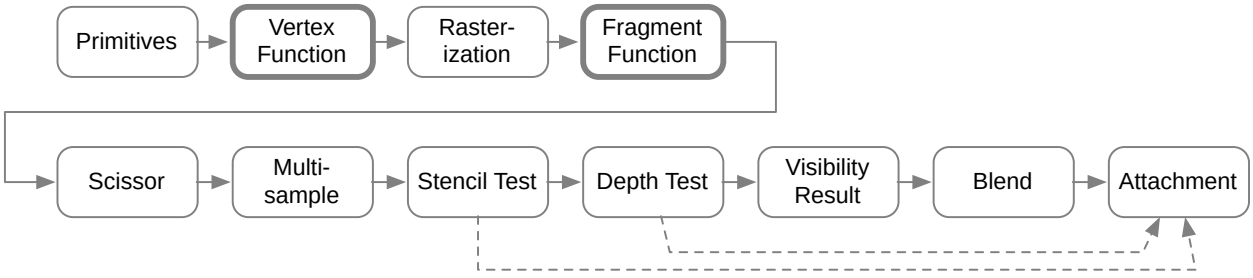
And this means the number of threads you can execute at the same time is dependent upon the program you write.

> *Note: the number of threads that can be executed for your program can be queried from the MTLComputePipelineState using the maxTotalThreadsPerThreadGroup method and the threadExecutionWidth method.*

Graphics Rendering

GPUs are primarily designed to accelerate the rendering of graphics images. As such they are designed around concepts such as "textures" and "buffers" and "depth buffers." (This is why even for compute kernels you still must deal with passing information in and out using textures and the like.)

The graphics pipeline used for graphics rendering involves the following steps:



The Compute Command encoder bypasses these steps, essentially providing a method for invoking a kernel across a one, two or three dimensional array of values.

Functions

The primary entry point for Metal functions (that is, the functions exposed to the CPU for invocation in a Metal application on your iOS or MacOS device) are either *kernel*, *vertex* or *fragment* functions.

Other C++14 functions may be declared but they may not be accessed outside of the GPU.

```
shader_function:
    shader_type return_type IDENTIFIER '(' parameter_list ')'
            compound_statement;

shader_type: 'vertex' | 'fragment' | 'kernel' ;
```

Function Types

There are three types of GPU functions that are exposed as entry points:

Vertex Functions

Vertex functions are called as part of a render pass encoded using the MTLRenderCommandEncoder, and calls this function once per vertex in the model. The purpose of the vertex model is to calculate the location of the input primitives in screen coordinates, which is the destination for projection of a virtual 3D world. Screen coordinates are x: -1 to 1 left to right, y: -1 to 1 bottom to top, and z: 0 to 1 from to back.

Fragment Functions

Fragment functions are called as part of a render pass encoded using the MTLRenderCommandEncoder, and is called once per pixel for each collection of pixels encoded in a polygon represented by a collection of vertices. The purpose of the vertex model is to calculate the color at each pixel.

Kernel Functions

Kernel functions are called as part of a compute command encoder, and is called the number of times specified in the MTLComputeCommandEncoder, as part of a one, two or three dimensional array of threads.

Passing In Resources

The function parameters of the vertex, fragment and kernel functions specify the resources that are used when invoking each function. Arguments are passed in as integer-indexed resources in a resource table

and are specified in the parameters to the function. The way this is done is described in greater detail in Apple's Metal documentation.

Specifying Location In The Parameter Description

The location of each parameter passed to a function can be specified as an attribute:

```
parameter_list: parameter_declaration
              | parameter_list ',' parameter_declaration
              ;
parameter_declaration: declaration_specifiers declarator attributes
                     ;

attributes: '[[' attribute_list ']]
          ;

attribute_list: attribute_item
              | attribute_list ',' attribute_item
              ;
```

Example:

```
fragment float4 fragment_fairy(VertexOut v [[stage_in]],
        texture2d<float> fairyTexture [[ texture(MXTextureIndex0) ]])
{
}
```

Specifying Location In A Structure

The locations can be passed in via a structure, passed in by value to the function:

```
structure_declarator_list
    : structure_declarator
    | structure_declarator_list ',' structure_declarator
    ;

structure_declarator
    : struct_declarator attributes
    ;
```

Example:

```
struct Foo {
    texture2d<float>  a [[texture(0)]];
    depth2d<float>    b [[texture(1)]];
};

kernel void my_kernel(Foo f)
{
}
```

Note: Nested structures are also supported; see section 4.3.2 of the Metal Shading Language Specification for more information.

Valid attributes (as listed in section 4.3 of the Metal Shading Language Specification):

attribute	corresponding data type	used in	description
[[buffer(index)]]	any	vertex, fragment, kernel	Specifies the data is attached using a setBuffer call (or equivalent) in the command encoder.
[[texture(index)]]	any	vertex, fragment, kernel	Specifies the data is atttached using a setTexture call (or equivalent) in the command encoder.
[[sampler(index)]]	any	vertex, fragment, kernel	Specifies the sampler is attached using a setSamplerState call (or equivalent) in the command encoder.
[[threadgroup(index)]]	any, with threadgroup memory attribute	kernel	Specifies the data buffer for the thread group memory at index specified in the MTLComputeCommandEncoder setThreadgroupMemoryLength:atIndex: method.
[[vertex_id]]	ushort, uint	vertex	The per-vertex identifier.
[[instance_id]]	ushort, uint	vertex	The per-instance identifier, when a vertex drawing method including an instance parameter is called.
[[stage_in]]	any	vertex, fragment, kernel	Per-item input, used in vertex and kernel functions in conjunction with the attributes buffer to pass in per-vertex or per-kernel inputs. For fragment functions handles the per-vertex generated outputs passed in from the vertex functions, extrapolated across the pixels in the fragment.
[[thread_position_in_grid]]	ushort, ushort2, ushort3, uint, uint2, uint3	kernel	The thread's position in the overall N-dimensional grid of threads.
[[thread_position_in_threadgroup]]	ushort, ushort2, ushort3, uint, uint2, uint3	kernel	The thread's position in the threadgroup
[[threadgroup_position_in_grid]]	ushort, ushort2, ushort3, uint, uint2, uint3	kernel	The threadgroup position in the grid.

Per-Vertex Attributes

Attributes can be used in a structure declaration to specify the location of each field of the structure, for values that are passed in for per-vertex and per-thread parameters using `[[stage_in]]`.

Example:

```
struct VertexIn {
    float3 position  [[attribute(MXAttributeIndexPosition)]];
    float3 normal    [[attribute(MXAttributeIndexNormal)]];
    float2 texture   [[attribute(MXAttributeIndexTexture)]];
};

vertex VertexOut vertex_function(VertexIn v [[stage_in]])
{
}
```

attribute	corresponding data type	description
[[attribute(index)]]	any	Specifies the attribute index which gives the data format and length of the field in the structure.

Vertex Function Return Attributes

A structure used to declare the return of a vertex function may contain additional attributes which specify how the values are returned.

> *Note: If a vertex function does not return a structure, it must return either a void or a float4, which is assumed to be the position, and the [[position]] attribute does not need to be specified.*

attribute	corresponding data type	description
[[clip_distance]]	float or float[n], with n known at compile time	Distance from vertex to clipping plane
[[position]]	float4	**Required.** The transformed vertex position. Used by the fragment shader to calculate screen position and depth.

Fragment Function Input Attributes

These attributes may be combined with the vertex function attribute list (above) to specify additional parameters passed in to the fragment function.

attribute	corresponding data type	description
[[color(index)]]	floatn, halfn, intn, uintn, shortn, or ushortn	The input value read from a color attachment. The index indicates from which color attachment to read from.
[[front_facing]]	bool	True if the fragment belongs to a front-facing primitive.
[[position]]	float4	The transformed vertex position. Used by the fragment shader to calculate screen position and depth.

Fragment Function Return Attributes

A structure used to declare the return of a fragment function may contain additional attributes which specify how values are returned.

If a fragment returns a `float3` or `float4` type, they correspond to the color output in color index 0. The fragment function may specify a `void` return type, and it may specify a structure with attributes from the list below.

> *Note: See section 4.3.4.4 of the Metal Shading Language Specification for more information.*

attribute	corresponding data type	description
[[color(m)]] [[color(m), index(i)]]	floatn, halfn, intn, uintn, shortn, or ushortn	Color value output for a color attachment. If index specified, index i can be used to specify one or more colors output by a fragment function for a given color attachment and is an input to the blend equation.
[[depth(depth_argument)]]	float	Depth value output using the function specified by `depth_argument`.
[[sample_mask]]	uint	Coverage mask.

Data Types

The Metal shading language extends the C++14 language to add support for a variety of scalar, vector and matrix data types, as well as support for textures maps. Data types are enumerated in the Metal Shading Language Specification in Chapter 2.

> *Note: The following list is not exhaustive.*

Commonly used types include:

Scalar Types

Type	Description	Size (bytes)
bool	boolean value, either true (1) or false (0).	1
char, int8_t	Signed 8-bit integer	1
unsigned char, uchar, uint8_t	Unsigned 8-bit integer	1
short, int16_t	Signed 16-bit integer	2
unsigned short, ushort, uint16_t	Unsigned 16-bit integer	2
int, int32_t	Signed 32-bit integer	4
unsigned int, uint, uint32_t	Unsigned 32-bit integer	4
half	16-bit floating point value	2
float	32-bit floating point value	4
size_t	Unsigned integer type, result of sizeof operator: a 64-bit unsigned integer	8
ptrdiff_t	Signed 64-bit integer, result of subtracting two pointers.	8
void	Empty value or no value; used to indicate no parameter or return type.	

Vector Types

In the list of types below, 'N' is either 2, 3 or 4, representing a 2-, 3- or 4-component vector type. (For example, a 4 item float vector would be written `float4`.)

> *Note: There are also packed variants of the vector types below; see section 2.2.3 of the Metal Shading Language Specification.*

Type	Description	Byte Size For N:		
		2	3	4
boolN	Array of booleans	2	4	4
charN	Array of signed 8-bit integers	2	4	4

Type	Description	Byte Size For N:		
		2	3	4
ucharN	Array of unsigned 8-bit integers	2	4	4
shortN	Array of signed 16-bit integers	4	8	8
ushortN	Array of unsigned 16-bit integers	4	8	8
intN	Array of signed 32-bit integers	8	16	16
uintN	Array of unsigned 32-bit integer	8	16	16
halfN	Array of 2-byte floating point values	4	8	8
floatN	Array of 4-byte floating point values	8	16	16

*Note: There is no **doubleN** vector declaration in Metal as of this writing.*

Vector Component Access

Vector components may be accessed using an array index:

```
int4 vector = int4(1,2,3,4);

vector[2] == 3
```

Vector components may also be accessed using a '.' operator and the fields x, y, z, w and r, g, b, a.

```
vector.z == vector.b == vector[2] == 3
```

Vector components accessed with a '.' operator may also be used to extract a vector or to assign components in a vector:

```
int2 subvect = vector.xz;    // subvect is the vector (1, 3)

vector.ba = int2(5,6);       // vector now is (1,2,5,6)
```

Matrix Types

Matrix types are 2 dimensional arrays of numbers which can be used to multiply against other matrices and against vectors. In the list below, M and N are values from 2 to 4. So, for example, a 3x4 matrix of half floating point values would be declared `half3x4`.

Note: For size and alignment, see table 5 of section 2.3 of the Metal Shading Language Specification.

Type	Description
halfMxN	Matrix of 2-byte floating point values M columns wide and N rows tall.
floatMxN	Matrix of 4-byte floating point values M columns wide and N rows tall.

Accessing Matrix Components

Matrix components may be accessed using array subscript syntax in order column, row:

```
float4x3 m;

m[1][2] = 5;          // set the item at column index 1, row index 2 to 5.

float3 col = m[0];    // set col to the column at index 0.
```

Buffers

Metal implements buffers as a pointer to a built-in or user-defined type that are declared in program scope or passed in as arguments to a function.

Note: See Section 2.7 of the Metal Shading Language Specification for limitations

Buffers may be declared in in the following address spaces, specified as a prefix to the data type declaration (see section 4.3 of the Metal Shading Language Specification for more information):

Address Space	Used In	Description
device	kernel, vertex, fragment	Memory allocated from the device memory pool that is both readable and writeable.
constant	kernel, vertex, fragment	Memory allocated from the device memory pool that is read-only. Variables in program scope declared constant must be initialized during the declaration.
threadgroup	kernel	Memory shared amongst all threads in a single thread group. Memory allocated in the threadgroup memory space only exists so long as the threads within a thread group exist and may be used to share information between threads in a thread group.
thread	kernel	Memory allocated that is only visible in the current thread. Variables declared inside a graphics or kernel function is allocated by default in thread address space.

Textures

The texture data type is a handle to a one-, two- or three-dimensional texture data. The following texture declarations are commonly used:

> *Note: Other texture types exist; see section 2.8 of the Metal Shading Language Specification for more information, as well as the MTLTexture object.*

Address Space	Description
texture1d<T, access a = access:sample>	A one dimensional texture array
texture2d<T, access a = access:sample>	A two dimensional texture array
texture3d<T, access a = access:sample>	A three dimensional texture array
texturecube<T, access a = access::sample>	A texture cube. This is 6 separate 2D textures forming a cube.
depth2d<T, access a = access::sample>	A two dimensional depth texture
depthcube<T, access a = access::sample>	A depth cube. This is 6 separate 2D depth textures forming a cube.

Depth textures may only use `float` for type T. Textures may use any one of `half`, `float`, `short`, `ushort`, `int` or `uint`.

The access parameter of the texture declaration is optional. If not provided it is assumed to be sample. Valid access attributes are:

Access	Used In	Description
sample	texture, depth	The texture object can be sampled. Implies the texture can be read with and without a sampler.
read	texture, depth	The texture object can only be read from.
write	texture only	The texture object can be written to.
read_write	texture only	The texture object can be read from or written to

Example of using access qualifiers:

```
void foo (texture2d<float> imgA [[texture(0)]],   // access::sample
          texture2d<float, access::read> imgB [[texture(1)]],
          texture2d<float, access::write> imgC [[texture(2)]])
```

Samplers

For textures accessed with a sampler, a sampler specifies how the contents of a texture map are accessed or interpolated.

Samplers are described in greater detail in section 2.9 of the Metal Shading Language Specification, and with the MTLSamplerState class. In short, however, a sampler may be declared by creating an MTLSamplerState class and passing it into the shader function through the MTLRenderCommandEncoder or MTLComputeCommandEncoder classes, or directly declared within the shader function using the `sampler` object.

Examples:

```
// Declare a sampler passed in through the MTLComputeCommandEncoder
kernel void my_kernel(device float4 *p [[buffer(0)]],
                      texture2d<float> img [[texture(0)]],
                      sampler smp [[sampler(3)]])
{
}

// Locally declare a simple sampler
fragment GBufferOut fragment_gbuffer(VertexOut v [[stage_in]],
        texture2d<float, access::sample> texture [[texture(0)]],
        depth2d<float, access::sample> shadowMap [[texture(1)]])
{
    constexpr sampler linearSampler (mip_filter::linear,
                                     mag_filter::linear,
                                     min_filter::linear);
...
}
```

Reading and Writing to a Texture

Functions used to access the contents of a texture is given in section 5.10 of the Metal Shading Language Specification. In short, however, the following methods are commonly used:

Method	Description
t.sample(sampler, coord)	Sample a color in the 2D or 3D texture. Coordinate width must match the number of dimensions in the texture.
t.read(coord)	Read from the texture at coordinate provided. The coordinate must be an integral vector with the same number of dimensions as the texture.
t.write(color,coord)	Write the color to the texture at coordinate provided. The coordinate must be an integral vector with the same number of dimensions as the texture.

A Brief Introduction To Homogeneous Coordinates

Homogeneous coordinates or projective coordinates is a system of coordinates used in projective geometry. It has the advantage that the coordinates of points, including those at infinity, can be represented using finite coordinates. They are useful because often equations using homogeneous coordinates can be far simpler than using traditional cartesian coordinates.

Homogeneous coordinates are used extensively in computer graphics for two primary reasons:

1. **Translations can be represented using a matrix multiply.** This has the advantage of allowing a collection of coordinate transformations that include scaling, rotations and translations to be represented with a single matrix, and further, allows us to reverse the coordinate transformations through matrix inversion.

2. **Perspective transformations may be represented.** Depth perspective (where farther away objects seem smaller)

Homogeneous Representation of 3D Coordinates

A 3D coordinate is represented in homogeneous coordinates by appending an extra dimension to the coordinate, with the property that a homogeneous point

$$(x,y,z,w)$$

represents the cartesian point

$$\left(\frac{x}{w},\frac{y}{w},\frac{z}{w}\right).$$

We generally convert a cartesian coordinate (x,y,z) to a homogeneous coordinate by appending w = 1: (x,y,z,1).

> *Note: We can use the same principle with 2D coordinates for transformations and the like by extending a coordinate (x,y) to a homogeneous coordinate (x,y,w).*

Homogeneous Coordinate Transformations

Coordinates in one coordinate system can be transformed using matrix multiplication. For our examples we use pre-multiplication:

$$P'=MP$$

where P is the original point, P' is the transformed point, and M is the 4x4 matrix representing a transformation.

Coordinates may be transformed back from P' to P by pre-multiplying by the inverse of the matrix M.

Transforming Normal Vectors

Normal vectors; that is, vectors which represent the normal of a surface, obey the property that the dot product of the normal and all points on a flat surface are the same constant:

$$N \cdot P = C$$

This implies if we wish to transform the normal of a surface (for lighting effects) we need to post-multiply our normal vector by the inverse of our matrix. This is because any normal N' must obey the property that the dot product against any transformed points P' have the same constant value.

$$\begin{aligned} N \cdot P &= N \cdot M^{-1} M P \\ &= N M^{-1} P' \\ &= N' \cdot P' \end{aligned}$$

By construction we see that $N' = NM^{-1}$.

Common Coordinate Transformations

Within our system of coordinate transformations the following matrices are used to represent translations, scaling and rotation.

Translation

The translation matrix moves a point by a specified amount along the x, y and z axis:

$$T(x,y,z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix}$$

Scale

The scale matrix scales the object along the x, y and z axis:

$$S(x,y,z) = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate

There are three separate rotation matrices, one for each axis x, y and z:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) & 0 \\ 0 & -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\alpha) = \begin{bmatrix} \cos(\alpha) & 0 & -\sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The general rotation matrix which rotates around an axis specified as (x,y,z) by the angle α is given by:

$$R_A(\alpha) = \begin{bmatrix} tx^2+c & txy+zs & txz-ys & 0 \\ txy-zs & ty^2+c & tyz+xs & 0 \\ txz+ys & tyz-xs & tz^2+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where axis A is (x,y,z), $c = \cos(\alpha)$, $s = \sin(\alpha)$, and $t = 1 - c$.

Perspective

The perspective transformation makes use of the fact that as the farther away an object gets, the larger the *w* component should become, making the apparent sizes smaller as things recede into the distance. The customary perspective matrix used by most third party libraries take a field of view, an aspect ratio, a near and far clipping plane, and transform objects to the screen bounding box defined by Metal:

$$P(fov,aspect,n,f) = \begin{bmatrix} \frac{fov}{aspect} & 0 & 0 & 0 \\ 0 & fov & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & -1 \\ 0 & 0 & \frac{2nf}{n-f} & 0 \end{bmatrix}$$