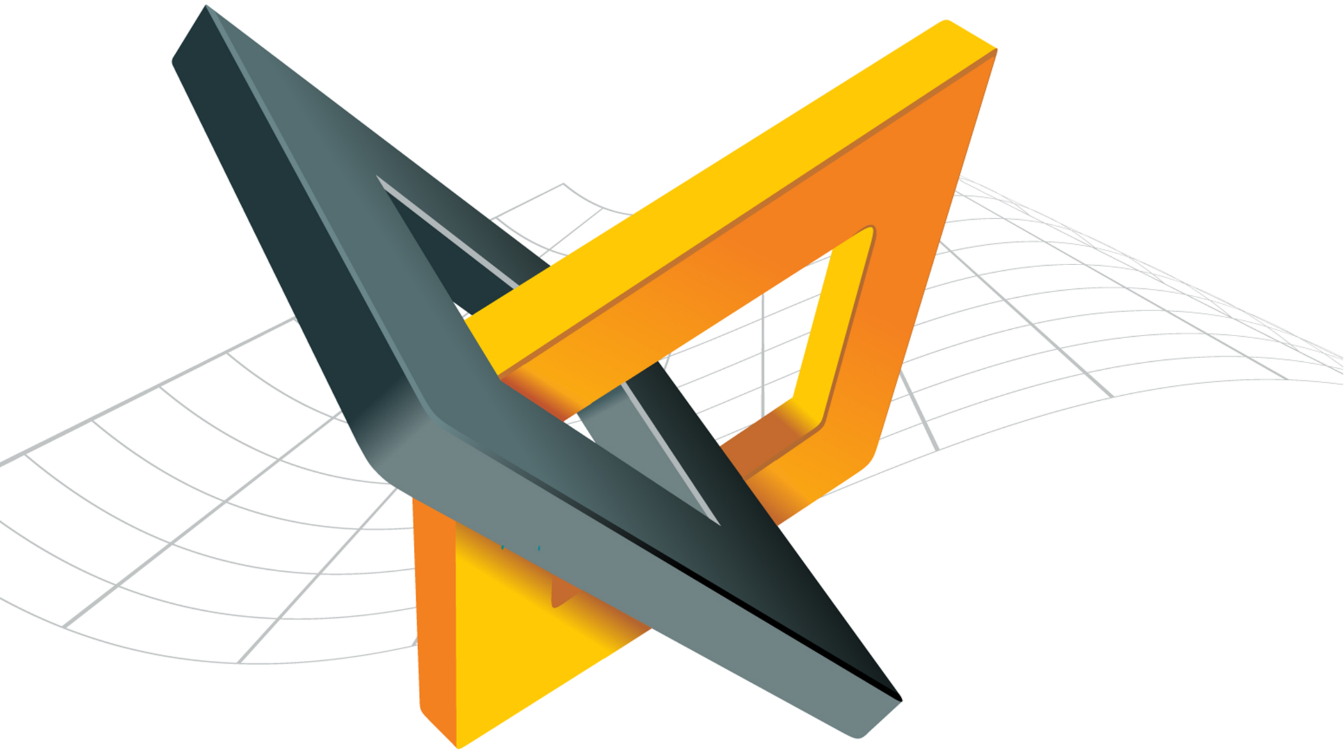
WARREN MOORE



METAL BY EXAMPLE

High-performance graphics and data-parallel programming for iOS

Contents

Foreword

Since the dawn of the iPhone, OpenGL ES has been the *de facto* graphics standard for gamers and artists on the iOS platform. Apple's introduction of Metal with iOS 8 instantly changed that. Metal gives developers the power to create console-level quality graphics on mobile devices with an order of magnitude improvement in performance over OpenGL ES. I saw this first-hand as the founder of a mobile gaming video company. Many of our clients were mobile game developers, and the transition from OpenGL ES to Metal served to separate mediocre game developers from the cream of the crop. In order to build standout games, graphics, and art in today's competitive mobile ecosystem, one must have a deep understanding of Metal.

Metal By Example is the first book of its kind focused solely on teaching the ins and outs of Metal. The author takes an example-based pedagogical style that teaches not only the theory behind Metal's design, but also how to apply it in practical real-world situations. Each chapter comes complete with sample code and introduces standard graphics techniques in the context of Metal. Topics include shaders, image processing, environment mapping, text rendering, and much more.

If you're an experienced mobile graphics engineer looking to harness the power that Metal has to offer, *Metal By Example* is the perfect resource to get you up to speed quickly.

Kevin Wang
Co-Founder, Kamcord

Preface

What This Book is About

This book is about Metal, a framework for 3D graphics rendering and data-parallel computation introduced by Apple in iOS 8. In contrast to preceding application programming interfaces (API) for graphics programming on these platforms, Metal offers unprecedented access to the graphics processing units (GPUs) inside iOS devices. It achieves this with a slim, elegant interface while incurring minimal CPU overhead. At the same time, Metal's low-level nature requires graphics programmers to be more familiar with topics such as concurrency and memory management in order to create stable and performant applications.

By exploring Metal with the examples provided in this book, you will learn how to achieve common tasks in 3D graphics and data-parallel programming, which will enable you do create applications of your own that take full advantage of the GPU. Metal is unique in that it is the lowest level of abstraction you can access to do graphics programming on iOS. That is what gives it such power, and what can make it intimidating to learn. This book exists to help you master Metal and unlock its potential.

Who This Book is For

We will assume you have intermediate experience with Objective-C and some familiarity with 3D graphics fundamentals. A background in linear algebra and a dash of calculus will be helpful. Without these foundations, developing an understanding of the topics covered in this book will be very difficult.

Perhaps the best book on the relevant mathematics is (Lengyel 2011). For a good, succinct overview of the fundamentals of 3D computer graphics, consult (Gortler 2012). For deeper coverage of topics in real-time rendering, see (Hughes et al. 2013).

What You'll Need

To run the sample projects associated with this book on an iOS device, you will need an Apple computer running OS X Mavericks or newer with Xcode 6.3 or newer and a device with an A7 processor or newer running iOS 8 or newer. Note that as of this writing, you cannot run Metal code in the iOS Simulator. You will need physical hardware to run Metal code for iOS.

Sample Code

The sample code for this book is freely available online at http://metalbyexample.com. It is licensed under The MIT License, which allows for modification and redistribution under the condition of acknowledgement of the original authorship.

The sample code is an integral part of reading this book. Many sections in the book refer the reader to the sample code, and reading and modifying it (or writing your own) is the only way to build expertise.

Deciding on a license is a difficult matter. Sample code is, in general, not production-ready, and should not be copied without modification into a product meant for broad consumption. Even so, we strongly believe that software licenses are not the correct place to legislate this type of behavior, and chose to be as liberal as reasonable. *Caveat emptor.* Please take the sample code as a suggestion rather than a finished product.

Why Not Swift?

For many years, Objective-C has been the language of choice for developing applications on the Mac and iOS. On the same day that Metal was announced, Apple also introduced a new programming language named Swift. We chose to write this book using Objective-C for two reasons.

At the time this project began, Swift was still a new language undergoing substantial changes. Swift has largely settled down, but its syntax continues to evolve, and it would have created an undue burden to keep up with the changes to Swift and Metal simultaneously.

The other reason to use Objective-C is that many games and game engines are written in languages derived from C, such as C++, and Objective-C is the closest match for these languages. Even so, the concepts presented here transfer readily, and Metal bridges nearly seamless to Swift. Without a doubt, Swift is the language of the future across all of Apple's platforms, but we hope that by writing this book in Objective-C we can be inclusive to the existing community while not falling too far into the past.

Acknowledgements

This book would not have come into existence without the constant support of the wondrous crowd of people I'm fortunate to be surrounded by.

Family first. Thanks to my sister Lisa and my dad Clark for lifelong love and support. Wish you could've seen this, Mom.

Thanks to Liz K. for being a keen sounding board and helping shoulder the weight of my uncertainty. More than anyone else, this wouldn't have happened without you.

Thanks to Cathy for ongoing lessons in the value of being true to oneself. Hope to see *your* book on *my* shelf soon.

Thanks to Scott Stevenson and Elliott Harris for being enthusiastic allies and supportive friends.

Thanks to my main goons, who are always down for anything: Arjun, Benson, Daniel, Gareth, John W., Larry, Liz S., Maria, Maureen, Niraj, Rachel, and Tom. Beer is thicker than water. Special thanks to Kristen for your support through some of the best and worst times.

I owe a debt of gratitude to early readers of the material in this book. Thanks in particular to Simon Gladman, Ash Oakenfold, Steven Nie, Michael G. Roberts, Ken Ferry, Andrew Long, Andrew Kharchyshyn, Jim Wrenholt, Matthew Lintlop, and Shannon Potter for early feedback that improved the finished product. Thanks to the Apple engineers who frequent the Developer Forums for providing assistance.

Chapter 1

Welcome

Welcome to *Metal by Example*! Throughout this book, we'll explore the Metal framework, using it to dive deeply into the worlds of real-time rendering and data-parallel computation. This chapter sets the stage for the rest of the book.

The purpose of this book is to explain in detail how to achieve common GPU programming tasks in Metal. Although many of the principles discussed here have been in use for decades, implementing them with a new framework isn't always easy. That's especially true when a new framework, in an effort to be more efficient, takes away many of the conveniences of higher-level libraries.

As a creator of games or other graphics-intensive apps, you have a lot of choices when it comes to graphics. You can use Unreal Engine or Unity and get cross-platform support without ever touching code. You can write SpriteKit or SceneKit code and take advantage of a tower of abstraction that lets you express yourself in terms of scene graphs and materials. You can write OpenGL or OpenGL ES code that requires you to be a bit more aware of how the GPU actually does its work. And as of iOS 8, you can dispense with most of the abstraction and use Metal.

It's important to realize that although Metal is low-level, it is still a layer of abstraction. This is a good thing, because without such a layer we would be responsible for knowing all the minuscule details of GPU programming, which is both a huge amount of knowledge and subject to change as new hardware is released. In fact, Metal itself sits above the abstraction of the driver architecture that serves as the common base for all graphics frameworks available on iOS. It is, to use a tired expression, *turtles all the way down*.

On the other hand, Metal makes the programmer responsible for resource management and concurrency to a greater extent than ever before. In addition to the various Metal classes and protocols, and the new Metal shading language, you have to maintain explicit control over when you manipulate memory, lest you crash or get undefined results. The sample code provided with this book provides a gentle introduction not just to Metal itself, but the low-level programming constructs that are necessary to use it well.

This book breaks down into roughly three sections. In chapters 2 through 4, we introduce Metal and teach you how to get 3D figures animating on the screen. In chapters 5 through 12, we explore the implementation of numerous common techniques such as model loading, lighting, texturing, cube maps, alpha blending, and text rendering. The two final chapters are an introduction to data-parallel programming which takes advantage of the GPU to do more generalized computation. At the end, you'll have the skills and confidence to use Metal to really unleash your creative potential and unlock the full power of the GPU.

Let's get started!

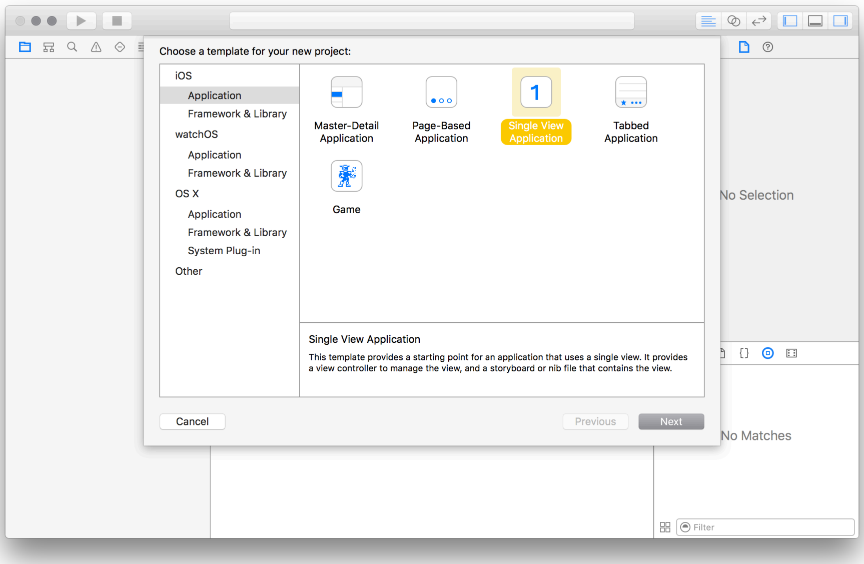
Chapter 2

Setting the Stage and Clearing the Screen

This chapter covers the bare minimum needed to clear the screen to a solid color in Metal. Even this simple operation requires many of the concepts exposed by the Metal framework. The following chapters will build upon this material and demonstrate how to do 3D rendering and more.

Creating a New Project

Let's create a new project in Xcode. We prefer to start with the Single View template, as it creates a view controller and wires it up to a window for us.



Single View Application

This template provides a starting point for an application that uses a single view. It provides a view controller to manage the view, and a storyboard or nib file that contains the view.

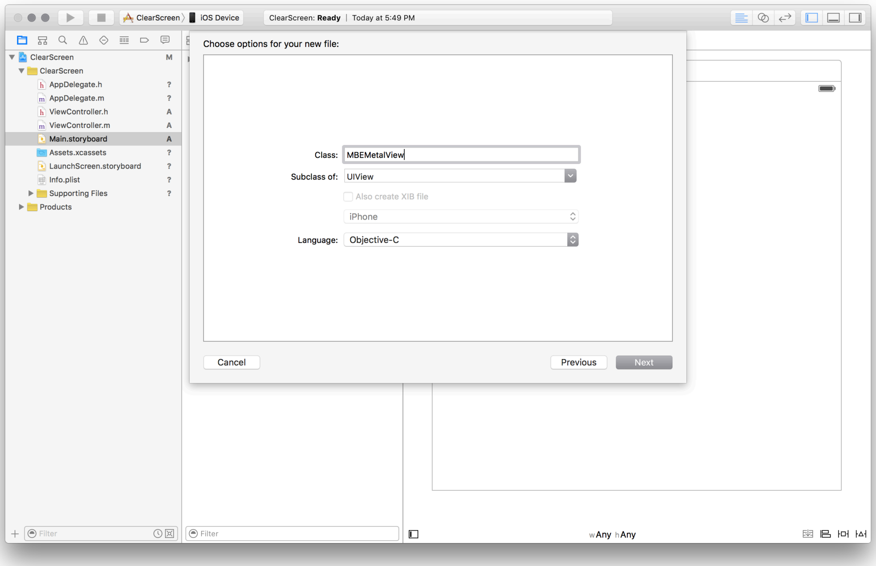
Figure 2.1: Creating a new single-view project in Xcode

Interfacing with UIKit

Each `UIView` on iOS uses a Core Animation layer as its backing store. In other words, the view's layer holds the actual contents that get drawn on screen. We say that the such a view is *backed* by a `CALayer` instance.

You can tell a `UIView` to change the type of layer it instantiates as its backing layer by overriding the `+layerClass` method on your `UIView` subclass.

You can follow along in your own project by using the "File • New • File…" menu in Xcode and generating a new Cocoa Touch class which is a subclass of UIView. We'll call it `MBEMetalView`.



Choose options for your new file:

Class: MBEMetalView

Subclass of: UIView

Language: Objective-C

Figure 2.2: Adding a new UIView subclass in Xcode

`CAMetalLayer` is provided not by the Metal framework, but by Core Animation. `CAMetalLayer` is the glue that binds UIKit and Metal together, and it provides some very nice features that we'll be seeing shortly.

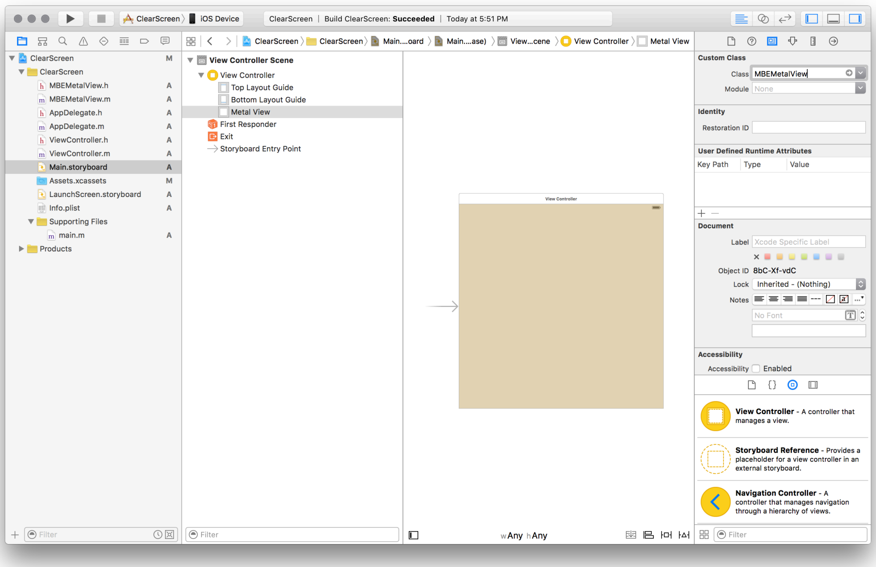
Let's implement `+layerClass` in our UIView subclass so it knows we want a Metal layer instead of a stock `CALayer`. Here's the complete `MBEMetalView` implementation as it stands:

```
@implementation MBEMetalView

+ (id)layerClass
{
    return [CAMetalLayer class];
}

@end
```

Change the Custom Class of the view in the main storyboard file to `MBEMetalView`. This will cause the subclass to be instantiated when the storyboard is loaded. This in turn gives us a proper Metal layer-backed view.



Custom Class
Class MBEMetalView
Module None

Identity
Restoration ID

User Defined Runtime Attributes
Key Path Type Value

Document
Label Xcode Specific Label
Object ID 8bC-Xf-vdC
Lock Inherited - (Nothing)
Notes

Accessibility
Accessibility Enabled

Figure 2.3: Setting a custom class for the view controller's root view

For the sake of convenience, you can add a property to your view class that is of type `CAMetalLayer`:

```
@property (readonly) CAMetalLayer *metalLayer;
```

This prevents you from having to repeatedly cast from the type of the `layer` property (i.e., `CALayer`) to the actual subclass (`CAMetalLayer`), since `CAMetalLayer` offers a few methods not found on `CALayer`. You can implement this property as follows:

```
- (CAMetalLayer *)metalLayer {
    return (CAMetalLayer *)self.layer;
}
```

If you build and run this project on your device, you will see nothing more than a plain white screen. To actually do any drawing, we need to learn about Metal *devices* and all the other objects they help us create. First, a word on the use of protocols in Metal.

Protocols

A common theme in the Metal API is the use of *protocols*, rather than concrete classes, to expose Metal functionality. Many Metal APIs return objects conforming to particular protocols, with the concrete type being secondary. This has the advantage that you don't need to care about the exact class implementing the functionality.

The syntax for declaring an object conforming to protocol `MTLDevice` will look like this:

```
id <MTLDevice> device;
```

Now, let's look at how to retrieve and use a device.

Devices

A device is an abstraction of the GPU. It provides methods for creating objects like command queues, render states, and libraries. We'll look at each of these in turn shortly.

Metal provides a C function, `MTLCreateSystemDefaultDevice`, that creates and returns a device that will suit our needs. This function takes no parameters, as there are no device properties that can be specified.

Our Metal layer needs to know which device will be rendering into it. We also need to configure a pixel format on the layer so everyone is in agreement about the size and order of its color components. `MTLPixelFormatBGRA8Unorm` is a good choice. With this pixel format, each pixel will be comprised of a blue, green, red, and alpha component, and each component will be an 8-bit unsigned integer (between 0 and 255).

It's helpful to create a `device` property on your view subclass, as we will need the device to create various resources for us in the remainder of the code:

```
@interface MBEMetalView ()
@property (readonly) id<MTLDevice> device;
@end
```

Before we can use functions from Metal, we need to import the Metal module with the following line:

```
@import Metal;
```

Here is the complete implementation of `-init` showing all of the necessary configuration:

```
- (instancetype)initWithCoder:(NSCoder *)aDecoder
{
    if ((self = [super initWithCoder:aDecoder]))
    {
        _metalLayer = (CAMetalLayer *)[self layer];
        _device = MTLCreateSystemDefaultDevice();
        _metalLayer.device = _device;
        _metalLayer.pixelFormat = MTLPixelFormatBGRA8Unorm;
    }

    return self;
}
```

Note that we override `initWithCoder:` because we know we're loading our view from a storyboard. For completeness, we should also include an override of `init` so that we can instantiate this class programmatically.

The redraw method

In the coming chapters, we'll delegate the responsibility of drawing to a separate class. For the time being, the `-redraw` method inside our view class is where we'll issue drawing commands. Also, we won't be redrawing the screen repeatedly, just clearing it once to a solid color. Therefore, it is sufficient to call `-redraw` just once. We can call it from our override of the `-didMoveToWindow` method, since this method will be called once as the app starts.

```
- (void)didMoveToWindow
{
    [self redraw];
}
```

The `redraw` method itself will do all of the work required to clear the screen. All code for the rest of the chapter is contained in this method.

Textures and Drawables

Textures in Metal are containers for images. You might be used to thinking of a texture as a single image, but textures in Metal are a little more abstract. Metal also permits a single texture object to represent an *array* of images, each of which is called a *slice*. Each image in a texture has a particular size and a pixel format. Textures may be 1D, 2D, or 3D.

We don't need any of these exotic types of textures for now. Instead, we will be using a single 2D texture as our renderbuffer (i.e., where the actual pixels get written). This texture will have the same resolution as the screen of the device our app is running on. We get a reference to this texture by using one of the features provided by Core Animation: the `CAMetalDrawable` protocol.

A *drawable* is object vended by a Metal layer that can hand us a renderable texture. Each time we draw, we will ask our Metal layer for a drawable object, from which we can extract a texture that acts as our framebuffer. The code is very straightforward:

```
id<CAMetalDrawable> drawable = [self.metalLayer nextDrawable];
id<MTLTexture> texture = drawable.texture;
```

We will also use the drawable to signal to Core Animation when we're done rendering into the texture, so it can be presented on the screen. To actually clear the framebuffer's texture, we need to set up a render pass descriptor that describes the actions to take each frame.

Render Passes

A *render pass descriptor* tells Metal what actions to take while an image is being rendered. At the beginning of the render pass, the `loadAction` determines whether the previous contents of the texture are cleared or retained. The `storeAction` determines what effect the rendering has on the texture: the results may either be stored or discarded. Since we want our pixels to wind up on the screen, we select our store action to be `MTLStoreActionStore`.

The pass descriptor is also where we choose which color the screen will be cleared to before we draw any geometry. In the case below, we choose an opaque red color (red = 1, green = 0, blue = 0, alpha = 1).

```
MTLRenderPassDescriptor *passDescriptor = [MTLRenderPassDescriptor
    renderPassDescriptor];
passDescriptor.colorAttachments[0].texture = texture;
passDescriptor.colorAttachments[0].loadAction = MTLLoadActionClear;
passDescriptor.colorAttachments[0].storeAction = MTLStoreActionStore;
passDescriptor.colorAttachments[0].clearColor = MTLClearColorMake(1, 0, 0, 1);
```

The render pass descriptor will be used below to create a command encoder for performing render commands. We will sometimes refer to a command encoder itself as a *render pass*.

Queues, Buffers, and Encoders

A *command queue* is an object that keeps a list of render command buffers to be executed. We get one by simply asking the device. Typically, a command queue is a long-lived object, so in more advanced scenarios, we would hold onto the queue we create for more than one frame.

```
id<MTLCommandQueue> commandQueue = [self.device newCommandQueue];
```

A *command buffer* represents a collection of render commands to be executed as a unit. Each command buffer is associated with a queue:

```
id<MTLCommandBuffer> commandBuffer = [commandQueue commandBuffer];
```

A *command encoder* is an object that is used to tell Metal what drawing we actually want to do. It is responsible for translating these high-level commands (set these shader parameters, draw these triangles, etc.) into low-level instructions that are then written into its corresponding command buffer. Once we have issued all of our draw calls (which we will discuss in the next chapter), we send the `endEncoding` message to the command encoder so it has the chance to finish its encoding.

```
id <MTLRenderCommandEncoder> commandEncoder =
    [commandBuffer renderCommandEncoderWithDescriptor:passDescriptor];
[commandEncoder endEncoding];
```

As its last action, the command buffer will signal that its drawable will be ready to be shown on-screen once all preceding commands are complete. Then, we call `commit` to indicate that this command buffer is complete and ready to be placed in command queue for execution on the GPU. This, in turn, will cause our framebuffer to be filled with our selected clear color, red.

```
[commandBuffer presentDrawable:drawable];
[commandBuffer commit];
```

The Sample App

The sample code for this chapter is in the 02-ClearScreen directory.

Figure 2.4: The results of clearing the screen to a solid red color

Conclusion

This chapter set the groundwork for more exciting topics, such as 3D rendering. Hopefully, you now have a sense for a few of the objects you'll see when working with Metal. In the next chapter, we'll look at drawing geometry in 2D.

Chapter 3

Drawing in 2D

In the previous chapter, we got a glimpse of many of the essential moving parts of the Metal framework: devices, textures, command buffers, and command queues. Even though it introduced a lot of Metal's moving parts, we couldn't cover everything at once. This chapter will add a little more depth to the discussion of the parts of Metal that are used when rendering geometry. In particular, we will take a trip through the Metal rendering pipeline, introduce functions and libraries, and issue our first draw calls.

The end goal of this chapter is to show you how to start rendering actual geometry with Metal. The triangles we draw will only be in 2D. Subsequent chapters will introduce the math necessary to draw 3D shapes and eventually animate 3D models.

Setup

The initializer of the `MBEMetalView` class has been refactored to call a sequence of methods that will do all the work necessary to get us ready to render:

```
[self makeDevice];
[self makeBuffers];
[self makePipeline];
```

`-makeDevice` is precisely the same code that used to be included directly in the `-init` method, as we saw in Chapter 2:

```
- (void)makeDevice
{
    device = MTLCreateSystemDefaultDevice();
    self.metalLayer.device = device;
    self.metalLayer.pixelFormat = MTLPixelFormatBGRA8Unorm;
}
```

The definitions of the other two methods will be given in the following sections.

Using Buffers to Store Data

Metal provides a protocol, `MTLBuffer`, for representing an untyped buffer of bytes with a fixed length. Its interface is very similar to `NSData`, in that it has a `contents` property (like `NSData`'s `bytes` property) and a `length` property. However, Metal buffers are created by requesting a buffer of a particular size from a Metal device.

We will use one Metal buffer to store both the positions and colors of the vertices we want to draw. This is called an *interleaved* buffer, because the position and color data is woven together into a contiguous stream, instead of specifying separate buffers for each of the vertex properties.

We define a structure to hold the position and color of each vertex. The members of this struct are of type `vector_float4`, which is a type offered by Apple's SIMD (which stands for "single-instruction, multiple-data") framework. In some instances, SIMD types can be operated on more efficiently than regular arrays of floats. Metal shaders operate most naturally on SIMD data, so we use them in client code too.

Here's our vertex type, representing a vertex with position and color:

```
typedef struct
{
    vector_float4 position;
    vector_float4 color;
} MBEVertex;
```

Our buffer will hold three vertices, enough to draw one triangle. The vertices are colored red, green, and blue, respectively. Since these values will not change, we declare them statically, then ask Metal for a buffer that contains a copy of the values:

```
- (void)makeBuffers
{
    static const MBEVertex vertices[] =
    {
        { .position = {  0.0,  0.5, 0, 1 }, .color = { 1, 0, 0, 1 } },
        { .position = { -0.5, -0.5, 0, 1 }, .color = { 0, 1, 0, 1 } },
        { .position = {  0.5, -0.5, 0, 1 }, .color = { 0, 0, 1, 1 } }
    };

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

You might have noticed that each vertex position is composed of four components, even though we only care about the x and y positions (since we're drawing a 2D triangle). This is because Metal works most naturally in 4D *homogeneous coordinates*, where each point has x, y, z, and w coordinates, with w fixed at 1. We use homogeneous coordinates because they make transformations like translation and projection more natural. We will cover these transformations in the following chapter.

In order to simplify the math we need to do in our shaders, the points are specified in *clip space coordinates*, where the x-axis runs from -1 to 1 going left to right, and the y-axis runs from -1 to 1 from bottom to top.

Each color is composed of the familiar red, green, blue and alpha components.

Now, let's look at the vertex and fragment functions that will be processing this data when we draw.

Functions and Libraries

Functions

As discussed in Chapter 1, modern graphics APIs provide a "programmable pipeline," meaning that they allow a lot of the operations carried out by the GPU to be specified by small programs applied to each vertex or pixel. These are commonly referred to as "shaders". This is a poor name, since shaders are responsible for much more than calculating the shading (color) of pixels. The core Metal framework does not use the name "shader," but we will use it interchangeably with "function" throughout this text.

To incorporate Metal shader code into our project, we need to add a Metal source file to our project to contain the shaders that will process our vertices and fragments.

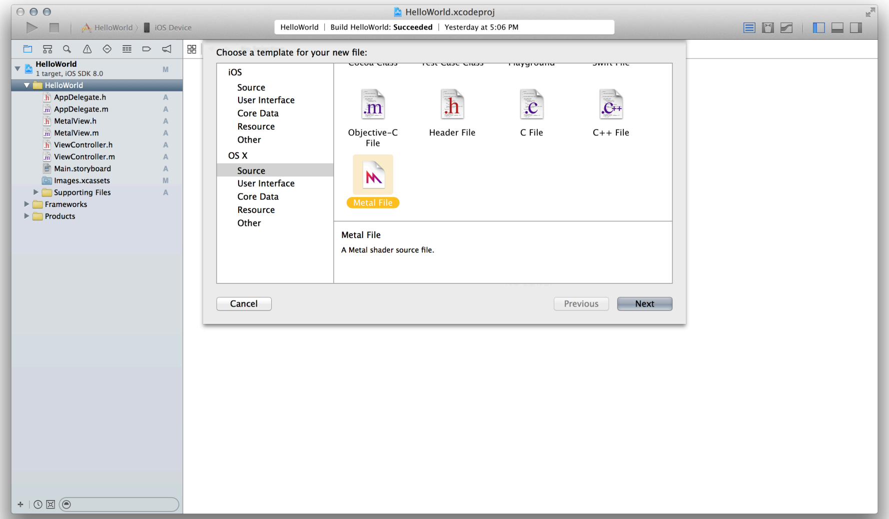


Figure 3.1: Adding a Metal shader source file in Xcode

Here is the preamble of Shaders.metal:

```
using namespace metal;

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

We define a struct named `Vertex` that looks very similar to the vertex struct we created in our Objective-C client code. One difference is that each member is of type `float4` rather than `vector_float4`. These are conceptually the same type (a SIMD vector of four 32-bit floats), but since the Metal shading language derives from C++, it is written differently. Vector and matrix types in Metal shader code belong to the `simd` namespace, which is inferred by the shader compiler, allowing us to simply write the type as `float4`.

One other difference is the presence of the `[[position]]` attribute on the `position` struct member. This attribute is used to signify to Metal which value should be regarded as the clip-space position of the vertex returned by the vertex shader. When returning a custom struct from a vertex shader, exactly one member of the struct must have this attribute. Alternatively, you may return a `float4` from your vertex function, which is implicitly assumed to be the vertex's position.

The definition of Metal shader functions must be prefixed with a *function qualifier*, which is one of *vertex*, *fragment*, or *kernel*. The *vertex* function qualifier denotes that a function will be used as a vertex shader, which is run once on each vertex in the geometry that is being drawn. In this case, our vertex shader is named `vertex_main`, and it looks like this:

```
vertex Vertex vertex_main(device Vertex *vertices [[buffer(0)]],
                          uint vid [[vertex_id]])
{
    return vertices[vid];
}
```

Since we are not transforming the position or color of the vertex in the vertex shader, we can simply return (by copy) the vertex at the index specified by the parameter with the `[[vertex_id]]` attribute, which runs from 0 to 2 as the vertex function is called for each vertex. If we were doing more sophisticated work, such as projecting the vertex or per-vertex lighting, we would do it here. We will see much more advanced vertex shaders in subsequent chapters.

Since each triangle that we draw to the screen might cover many pixels, there must be a pipeline stage that takes the values returned by the vertex function and *interpolates* them to produce a value for each possible pixel (*fragment*) in the current triangle. This stage is called the *rasterizer*. The rasterizer effectively "chops up" triangles into their constituent fragments and produces an interpolated value for each of the properties (position and color, in our case) at each pixel that might be affected by the current triangle. These interpolated values are then handed to the fragment function, which performs any necessary per-fragment work (such as texturing and per-pixel lighting).

The source for the fragment function is quite brief:

```
fragment float4 fragment_main(Vertex inVertex [[stage_in]])
{
    return inVertex.color;
}
```

The fragment function, `fragment_main`, takes a `Vertex` qualified with the attribute qualifier `[[stage_in]]`, which identifies it as per-fragment data rather than data that is constant across a draw call. This vertex does not correspond exactly to an instance of `Vertex` as returned by the vertex function. Rather, as described above, it is an interpolated vertex generated by the rasterizer.

It is the responsibility of the fragment function to return a color to be written into the renderbuffer, so `fragment_main` simply extracts the (interpolated) fragment color and returns it.

Libraries

Often, graphics libraries require the source of each shader to be compiled separately. Shaders must then be linked together to form a *program*. Metal introduces a nicer abstraction around shaders: the *library*. A library is nothing more than a logical group of functions written in the Metal shading language. In the case of this chapter's sample project, we have lumped our vertex and fragment function into a single file (Shaders.metal). This file will be compiled along with the rest of the project, and the compiled functions are included in the app bundle. Metal allows us to easily look up functions by name at runtime as shown in the next listing.

```
id<MTLLibrary> library = [self.device newDefaultLibrary];

id<MTLFunction> vertexFunc = [library newFunctionWithName:@"vertex_main"];
id<MTLFunction> fragmentFunc = [library newFunctionWithName:@"fragment_main"];
```

We could also have chosen to dynamically compile the application's shaders into a library at runtime and request the function objects from it. Since the shader code is known at application compile time, this is a unnecessary step, and we just use the default library.

Once you have references to your vertex and fragment functions, you configure your pipeline to use them, and the necessary linkage is performed implicitly by Metal.
This is illustrated in the following sections. The code above is incorporated in the `-makePipeline` method in the sample code.

The Render Pipeline

Previous generations of graphics libraries have led us to think of graphics hardware as a state machine: you set some state (like where to pull data from, and whether draw calls should write to the depth buffer), and that state affects the draw calls you issue afterward. Most of the API calls in a graphics library like OpenGL exist to allow you to set various aspects of state. This was especially true of the "fixed-function pipeline" as discussed in Chapter 1.

Metal provides a somewhat different model. Rather than calling API that acts on some global "context" object, much of Metal's state is built into pre-compiled objects comprising a virtual pipeline that takes vertex data from one end and produces a rasterized image on the other end.

Why does this matter? By requiring expensive state changes to be frozen in precompiled render state, Metal can perform validation up-front that would otherwise have to be done every draw call. A complex state machine like OpenGL spends a substantial amount of CPU time simply making sure the state configuration is self-consistent. Metal avoids this overhead by requiring that we fix the most expensive state changes as part of pipeline creation, rather than allowing them to arbitrarily change at any time. Let's look at this in more depth.

Render Pipeline Descriptors

In order to build the pipeline state that describes how Metal should render our geometry, we need to create a descriptor object that joins together a few things: the vertex and fragment functions we want to use when drawing, and the format of our framebuffer attachments.

A *render pipeline descriptor* is an object of type `MTLRenderPipelineDescriptor` that holds configuration options for a pipeline. Here is how we create a pipeline descriptor object:

```
MTLRenderPipelineDescriptor *pipelineDescriptor = [MTLRenderPipelineDescriptor
    new];
pipelineDescriptor.vertexFunction = vertexFunc;
pipelineDescriptor.fragmentFunction = fragmentFunc;
pipelineDescriptor.colorAttachments[0].pixelFormat =
    self.metalLayer.pixelFormat;
```

The vertex and fragment function properties are set to the function objects we previously requested from the default library.

When drawing, we can target many different kinds of *attachments*. Attachments describe the textures into which the results of drawing are written. In this case, we have just one attachment: the color attachment at index 0. This represents the texture that will actually be displayed on the screen. We set the attachment's pixel format to the pixel format of the `CAMetalLayer` that backs our view so that Metal knows the color depth and component order of the pixels in the texture to which it will be drawing.

Now that we have a pipeline descriptor, we need to ask Metal to do the work of creating the render pipeline state.

Render Pipeline State

A *render pipeline state* is an object conforming to protocol `MTLRenderPipelineState`. We create a pipeline state by providing our newly-created pipeline descriptor to our device, storing the result in another property named `pipeline`.

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

The pipeline state encapsulates the compiled and linked shader program derived from the shaders we set on the descriptor. Therefore, it is a somewhat costly object. Ideally, you will only create one render pipeline state for each pair of shader functions, but sometimes the target pixel formats of your attachments will vary, necessitating additional render pipeline state objects. The point is that creating a render pipeline state object is expensive, so you should avoid creating them on a per-frame basis, and cache them as much as possible.

The code above is the central portion of the `-buildPipeline` method in the sample code. Since the render pipeline state object will live for the lifetime of our application, we store it in a property. We also create a command queue and store it in a propery, as described in the previous chapter.

Encoding Render Commands

In the previous chapter, we didn't interact much with the render command encoder, since we weren't performing any drawing. This time around, we need to encode actual draw calls into our render command buffer. First, let's revisit the retrieval of a drawable from the layer and the configuration of a render pass descriptor. This is the beginning of the `-redraw` method for this sample project:

```
id<CAMetalDrawable> drawable = [self.metalLayer nextDrawable];
id<MTLTexture> framebufferTexture = drawable.texture;
```

If for some reason we don't get a drawable or a renderable texture from our drawable, the actual draw calls are guarded by an `if (drawable)` conditional. Trying to create a command buffer with a pass descriptor whose texture is `nil` will result in an exception.

The `renderPass` object is constructed exactly as before, except that we have chosen a light gray color to clear the screen with, instead of the obnoxious red we used last time.

```
MTLRenderPassDescriptor *passDescriptor = [MTLRenderPassDescriptor
    renderPassDescriptor];
passDescriptor.colorAttachments[0].texture = framebufferTexture;
passDescriptor.colorAttachments[0].clearColor = MTLClearColorMake(0.85, 0.85,
    0.85, 1);
passDescriptor.colorAttachments[0].storeAction = MTLStoreActionStore;
passDescriptor.colorAttachments[0].loadAction = MTLLoadActionClear;
```

Now, we start a render pass by creating a render command encoder with our render pass descriptor and encode our draw call:

```
id<MTLRenderCommandEncoder> commandEncoder = [commandBuffer
    renderCommandEncoderWithDescriptor:passDescriptor];
[commandEncoder setRenderPipelineState:self.pipeline];
[commandEncoder setVertexBuffer:self.vertexBuffer offset:0 atIndex:0];
[commandEncoder drawPrimitives:MTLPrimitiveTypeTriangle vertexStart:0
    vertexCount:3];
[commandEncoder endEncoding];
```

The `-setVertexBuffer:offset:atIndex:` method is used to map from the `MTLBuffer` objects we created earlier to the parameters of the vertex function in our shader code. Recall that the `vertices` parameter was attributed with the `[[buffer(0)]]` attribute, and notice that we now provide an index of `0` when preparing our draw call.

We call the `-drawPrimitives:vertexStart:vertexCount:instanceCount:` method to encode a request to draw our triangle. We pass `0` to the `vertexStart` parameter so that we begin drawing from the very start of the buffer. We pass `3` for `vertexCount` because a triangle has three points.

We finish up as before by triggering display of the drawable and committing the command buffer:

```
[commandBuffer presentDrawable:drawable];
[commandBuffer commit];
```

These lines round out the new implementation of the `-redraw` method.

Staying In-Sync With `CADisplayLink`

Now that our `-redraw` implementation is complete, we need to figure out how to call it repeatedly. We could use a regular old `NSTimer`, but Core Animation provides a much better way: `CADisplayLink`. This is a special kind of timer that is synchronized with the display loop of the device, leading to more consistent timing. Each time the display link fires, we'll call our `-redraw` method to update the screen.

We override `-didMoveToSuperview` to configure the display link:

```
- (void)didMoveToSuperview
{
    [super didMoveToSuperview];
    if (self.superview)
    {
        self.displayLink = [CADisplayLink displayLinkWithTarget:self
             selector:@selector(displayLinkDidFire:)];
        [self.displayLink addToRunLoop:[NSRunLoop mainRunLoop]
             forMode:NSRunLoopCommonModes];
    }
    else
    {
        [self.displayLink invalidate];
        self.displayLink = nil;
    }
}
```

This creates a display link and schedules it with the main run loop. If we are removed from our superview, we invalidate the display link and nil it out.

Sixty times per second, the display link fires and invokes its target method, `-displayLinkDidFire:`, which in turn calls `-redraw`:

```
- (void)displayLinkDidFire:(CADisplayLink *)displayLink
{
    [self redraw];
}
```

The Sample Project

The sample project for this chapter is in the 03-DrawingIn2D directory. If you build and run it, you should see a very colorful triangle appear on the screen.

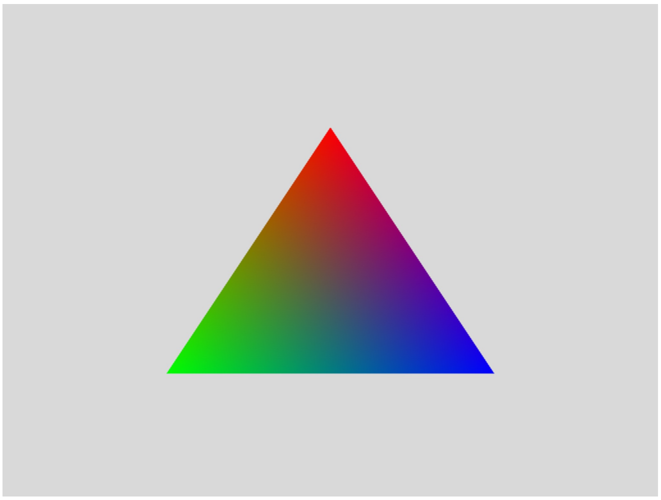


Figure 3.2: A multicolored triangle, rendered by Metal

Note that if you rotate the display, the triangle is distorted. This is because we specified in the coordinates of the triangle in clip space, which always runs from -1 to 1 along each axis regardless of the aspect ratio. In the next chapter we will discuss how to compensate for the aspect ratio when rendering three-dimensional figures.

Chapter 4

Drawing in 3D

Building on what we learned about the rendering pipeline in the previous chapter, we will now begin our coverage of rendering in three dimensions.

Specifying Geometry in 3D

Cube Geometry

The object we will render in this chapter is a simple cube. It is easy to write the vertices of a cube in code, avoiding the complexity of loading a 3D model for now. Here are the vertices for the cube mesh:

```
const MBEVertex vertices[] =
{
    { .position = { -1,  1,  1, 1 }, .color = { 0, 1, 1, 1 } },
    { .position = { -1, -1,  1, 1 }, .color = { 0, 0, 1, 1 } },
    { .position = {  1, -1,  1, 1 }, .color = { 1, 0, 1, 1 } },
    { .position = {  1,  1,  1, 1 }, .color = { 1, 1, 1, 1 } },
    { .position = { -1,  1, -1, 1 }, .color = { 0, 1, 0, 1 } },
    { .position = { -1, -1, -1, 1 }, .color = { 0, 0, 0, 1 } },
    { .position = {  1, -1, -1, 1 }, .color = { 1, 0, 0, 1 } },
    { .position = {  1,  1, -1, 1 }, .color = { 1, 1, 0, 1 } }
};
```

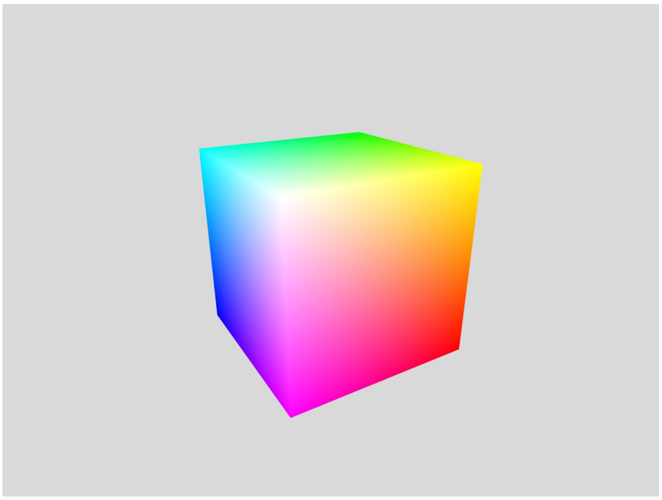


Figure 4.1: The cube rendered by the sample app

We reuse the same `MBEVertex` struct from the previous chapter, which has a position and color for each vertex. Since we have not introduced lighting yet, giving each vertex a distinct color provides an important depth cue. As before, we create a buffer to hold the vertices:

```
vertexBuffer = [device newBufferWithBytes:vertices
    length:sizeof(vertices)
    options:MTLResourceOptionCPUCacheModeDefault];
```

Index Buffers

In the previous chapter, we stored the vertices of our triangle in the order they were to be drawn, and each vertex was used only once. In the case of a cube, each vertex belongs to several triangles. Ideally, we would reuse those vertices instead of storing additional copies of each vertex in memory. As models grow in size, vertex reuse becomes even more important.

Fortunately, like most graphics libraries, Metal gives us the ability to provide an *index buffer* along with our vertex buffer. An index buffer contains a list of indices into the vertex buffer that specifies which vertices make up each triangle.

First, we define a couple of `typedef`s that will simplify working with indices:

```
typedef uint16_t MBEIndex;
const MTLIndexType MBEIndexType = MTLIndexTypeUInt16;
```

Starting out, we will use 16-bit unsigned indices. This allows each mesh to contain up to 65536 distinct vertices, which will serve our purposes for quite a while. If we need to accommodate more vertices in the future, we can change these definitions, and our code will adapt to the larger index size. Metal allows 16- and 32-bit indices.

Each square face of the cube is broken up into two triangles, comprising six indices. We specify them in an array, then copy them into a buffer:

```
const MBEIndex indices[] =
{
    3, 2, 6, 6, 7, 3,
    4, 5, 1, 1, 0, 4,
    4, 0, 3, 3, 7, 4,
    1, 5, 6, 6, 2, 1,
    0, 1, 2, 2, 3, 0,
    7, 6, 5, 5, 4, 7
};


indexBuffer = [device newBufferWithBytes:indices
    length:sizeof(indices)
    options:MTLResourceOptionCPUCacheModeDefault];
```

Now that we have defined some geometry to work with, let’s talk about how to render a 3D scene.

Dividing Work between the View and the Renderer

In the previous chapter, we gave the `MBEMetalView` class the responsibility of rendering the triangle. Now, we would like to move to a more sustainable model, by fixing the functionality of the view, and offloading the job of resource management and rendering to a separate class: the renderer.

Responsibilities of the View Class

The view class should only be concerned with getting pixels onto the screen, so we remove the command queue, render pipeline, and buffer properties from it. It retains the responsibility of listening to the display link and managing the texture(s) that will be attachments of the render pass.

The new `MBEMetalView` provides properties named `currentDrawable`, which vends the `CAMetalDrawable` object for the current frame, and `currentRenderPassDescriptor`, which vends a render pass descriptor configured with the drawable's texture as its primary color attachment.

The Draw Protocol

In order to do drawing, we need a way for the view to communicate with us that it's time to perform our draw calls. We decouple the notion of a view from the notion of a renderer through a protocol named `MBEMetalViewDelegate` which has a single required method: `-drawInView:`.

This draw method will be invoked once per display cycle to allow us to refresh the contents of the view. Within the delegate's implementation of the method, the `currentDrawable` and `currentRenderPassDescriptor` properties can be used to create a render command encoder (which we will frequently call a *render pass*) and issue draw calls against it.

Responsibilities of the Renderer Class

Our renderer will hold the long-lived objects that we use to render with Metal, including things like our pipeline state and buffers. It conforms to the `MBEMetalViewDelegate` protocol and thus responds to the `-drawInView:` message by creating a command buffer and command encoder for issuing draw calls. Before we get to that, though, we need to talk about the work the draw calls will be doing.

Transforming from 3D to 2D

In order to draw 3D geometry to a 2D screen, the points must undergo a series of transformations: from object space, to world space, to eye space, to clip space, to normalized device coordinates, and finally to screen space.

From Object Space to World Space

The vertices that comprise a 3D model are expressed in terms of a local coordinate space (called *object space*). The vertices of our cube are specified about the origin, which lies at the cube's center. In order to orient and position objects in a larger scene, we need to specify a transformation that scales, translates, and rotates them into *world space*.

You may recall from linear algebra that matrices can be multiplied together (*concatenated*) to build up a single matrix that represents a sequence of linear transformations. We will call the matrix that gathers together the sequence of transformations that move an object into world space the *model matrix*. The model matrix of our cube consists of a scale transformation followed by two rotations. Each of these individual transformations varies with time, to achieve the effect of a pulsing, spinning cube.

Here is the code for creating the sequence of transformations and multiplying them together to create the world transformation:

```
float scaleFactor = sinf(5 * self.time) * 0.25 + 1;
vector_float3 xAxis = { 1, 0, 0 };
vector_float3 yAxis = { 0, 1, 0 };
matrix_float4x4 xRot = matrix_float4x4_rotation(xAxis, self.rotationX);
matrix_float4x4 yRot = matrix_float4x4_rotation(yAxis, self.rotationY);
matrix_float4x4 scale = matrix_float4x4_uniform_scale(scaleFactor);
matrix_float4x4 modelMatrix = matrix_multiply(matrix_multiply(xRot, yRot),
    scale);
```

We use the convention that matrices are applied from right-to-left to column vectors, so `modelMatrix` scales a vertex, then rotates it about the Y axis, then rotates it about the X axis. The `rotationX`, `rotationY` and `time` properties are updated each frame so that this transformation is animated.

From World Space to View Space

Now that we have our scene (the scaled, rotated cube) in world space, we need to position the entire scene relative to the eye point of our virtual camera. This transformation is called the *view space* (or, equivalently, the *eye space* or *camera space*) transformation. Everything that will eventually be visible on screen is contained in a pyramidal shape called the *view frustum*, illustrated below. The position of the virtual camera's eye is the apex of this viewing volume, the point behind the middle of the near plane of the viewing frustum.

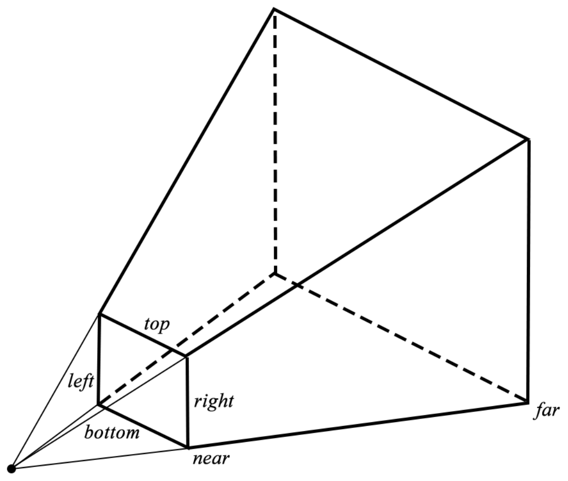


Figure 4.2: The view frustum. The apex of the frustum coincides with the eye of the virtual camera.

Constructing the transformation for the camera requires us to think backwards: moving the camera farther back is equivalent to moving the scene deeper into the screen, and rotating the scene counterclockwise about the Y axis is equivalent to the camera orbiting the scene clockwise.

In our sample scene, we want to position the camera a few units back from the cube. We use the convention that world space is "right-handed," with the Y axis pointing up, meaning that the Z axis points out of the screen. Therefore, the correct transformation is a translation that moves each vertex a *negative* distance along the Z axis. Equivalently, this transformation moves the camera a positive distance along the Z axis. It's all relative.

Here is the code for building our view matrix:

```
vector_float3 cameraTranslation = { 0, 0, -5 };
matrix_float4x4 viewMatrix = matrix_float4x4_translation(cameraTranslation);
```

From View Space to Clip Space

The projection matrix transforms view space coordinates into *clip space* coordinates.

Clip space is the 3D space that is used by the GPU to determine visibility of triangles within the viewing volume. If all three vertices of a triangle are outside the clip volume, the triangle is not rendered at all (it is *culled*). On the other hand, if one or more of the vertices is inside the volume, it is *clipped* to the bounds, and one or more modified triangles are used as the input to the vertex shader.

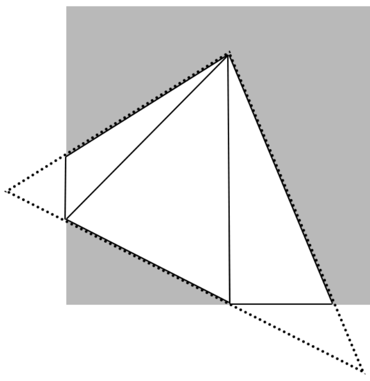


Figure 4.3: An illustration of a triangle being clipped. Two of its vertices are beyond the clipping bounds, so the face has been clipped and retriangulated, creating three new triangles.

The perspective projection matrix takes points from view space into clip space via a sequence of scaling operations. During the previous transformations, the w component remained unchanged and equal to 1, but the projection matrix affects the w component in such a way that if the absolute values of the x, y, or z component is greater than the absolute value of the w component, the vertex lies outside the viewing volume and is clipped.

The perspective projection transformation is encapsulated in the `matrix_float4x4_perspective` utility function. In the sample code, fix a vertical field of view of about 70 degrees, choose a far and near plane value, and select an aspect ratio that is equal to the ratio between the current drawable width and height of our Metal view.

```
float aspect = drawableSize.width / drawableSize.height;
float fov = (2 * M_PI) / 5;
float near = 1;
float far = 100;
matrix_float4x4 projectionMatrix = matrix_float4x4_perspective(aspect, fov,
    near, far);
```

We now have a sequence of matrices that will move us all the way from object space to clip space, which is the space that Metal expects the vertices returned by our vertex shader to be in. Multiplying all of these matrices together produces a *model-view-projection* (MVP) matrix, which is what we will actually pass to our vertex shader so that each vertex can be multiplied by it on the GPU.

The Perspective Divide: From Clip Space to NDC

In the case of perspective projection, w component is calculated so that the perspective divide produces foreshortening, the phenomenon of farther objects being scaled down more.

After we hand a projected vertex to Metal from our vertex function, it divides each component by the w component, moving from clip-space coordinates to *normalized device coordinates* (NDC), after the relevant clipping is done against the viewing volume bounds. Metal's NDC space is a cuboid $[-1, 1] \times [-1, 1] \times [0, 1]$, meaning that x and y coordinates range from -1 to 1, and z coordinates range from 0 to 1 as we move *away from the camera*.

The Viewport Transform: From NDC to Window Coordinates

In order to map from the half-cube of NDC onto the pixel coordinates of a view, Metal does one final internal transformation by scaling and biasing the normalized device coordinates such that they cover the size of the *viewport*. In all of our sample code, the viewport is a rectangle covering the entire view, but it is possible to resize the viewport such that it covers only a portion of the view.

3D Rendering in Metal

Uniforms

A *uniform* is a value that is passed as a parameter to a shader that does not change over the course of a draw call. From the point of view of a shader, it is a constant.

In the following chapters, we will bundle our uniforms together in a custom structure. Even though we only have one such value for now (the MVP matrix), we will establish the habit now:

```
typedef struct
{
    matrix_float4x4 modelViewProjectionMatrix;
} MBEUniforms;
```

Since we are animating our cube, we need to regenerate the uniforms every frame, so we put the code for generating the transformations and writing them into a buffer into a method on the renderer class named `-updateUniforms`.

The Vertex Shader

Now that we have some of the foundational math for 3D drawing in our toolbox, let's discuss how to actually get Metal to do the work of transforming vertices.

Our vertex shader takes a pointer to an array of vertices of type `Vertex`, a struct declared in the shader source. It also takes a pointer to a uniform struct of type `Uniforms`. The definition of these types are:

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

struct Uniforms
{
    float4x4 modelViewProjectionMatrix;
};
```

The vertex shader itself is straightforward. In order to find the clip-space coordinates of the vertex, it multiplies the position by the MVP matrix from the uniforms and assigns the result to the output vertex. It also copies the incoming vertex color to the output vertex without modification.

```
vertex Vertex vertex_project(device Vertex *vertices [[buffer(0)]],
                             constant Uniforms *uniforms [[buffer(1)]],
                             uint vid [[vertex_id]])
{
    Vertex vertexOut;
    vertexOut.position = uniforms->modelViewProjectionMatrix *
         vertices[vid].position;
    vertexOut.color = vertices[vid].color;

    return vertexOut;
}
```

Notice that we are using two different address space qualifiers in the parameter list of the function: `device` and `constant`. In general, the `device` address space should be used when indexing into a buffer using per-vertex or per-fragment offset such as the parameter attributed with `vertex_id`. The `constant` address space is used when many invocations of the function will access the same portion of the buffer, as is the case when accessing the uniform structure for every vertex.

The Fragment Shader

The fragment shader is identical to the one used in the previous chapter:

```
fragment half4 fragment_flatcolor(Vertex vertexIn [[stage_in]])
{
    return half4(vertexIn.color);
}
```

Preparing the Render Pass and Command Encoder

Each frame, we need to configure the render pass and command encoder before issuing our draw calls:

```
    [commandEncoder setDepthStencilState:self.depthStencilState];
    [commandEncoder setFrontFacingWinding:MTLWindingCounterClockwise];
    [commandEncoder setCullMode:MTLCullModeBack];
```

The `depthStencilState` property is set to the previously-configured stencil-depth state object.

The front-face *winding order* determines whether Metal considers faces with their vertices in clockwise or counterclockwise order to be front-facing. By default, Metal considers clockwise faces to be front-facing. The sample data and sample code prefer counterclockwise, as this makes more sense in a right-handed coordinate system, so we override the default by setting the winding order here.

The *cull mode* determines whether front-facing or back-facing triangles (or neither) should be discarded (culled). This is an optimization that prevents triangles that cannot possibly be visible from being drawn.

Issuing the Draw Call

The renderer object sets the necessary buffer properties on the render pass' argument table, then calls the appropriate method to render the vertices. Since we do not need to issue additional draw calls, we can end encoding on this render pass.

```
[renderPass setVertexBuffer:self.vertexBuffer offset:0 atIndex:0];

NSUInteger uniformBufferOffset = sizeof(MBEUniforms) * self.bufferIndex;
[renderPass setVertexBuffer:self.uniformBuffer offset:uniformBufferOffset
     atIndex:1];

[renderPass drawIndexedPrimitives:MTLPrimitiveTypeTriangle
    indexCount:[self.indexBuffer length] / sizeof(MBEIndex)
    indexType:MBEIndexType
    indexBuffer:self.indexBuffer
    indexBufferOffset:0];

[renderPass endEncoding];
```

As we saw in the previous chapter, the first parameter tells Metal what type of primitive we want to draw, whether points, lines, or triangles. The rest of the parameters tell Metal the count, size, address, and offset of the index buffer to use for indexing into the previously-set vertex buffer.

The Sample Project

The sample code for this chapter is in the 04-DrawingIn3D directory. Bringing together everything we learned above, it renders a pulsing, rotating cube in glorious 3D.

Chapter 5

Lighting

In this chapter, we'll start to increase the realism in our virtual scenes by loading 3D models from files and lighting them with directional lights.

Up until now, we've been hardcoding our geometry data directly into the program. This is fine for tiny bits of geometry, but it becomes unsustainable pretty quickly. By storing model data on the file system in a standard format, we can begin to work with much larger datasets.

Loading OBJ Models

One of the most popular formats for storing basic 3D model data is the OBJ format (Wavefront Technologies 1991). OBJ files store lists of vertex positions, normals, texture coordinates, and faces in a human-readable format. The sample code includes a rudimentary OBJ parser that can load OBJ models into memory in a suitable format for rendering with Metal. We will not describe the loader in detail here.

The model we use in this chapter is a teapot. The teapot shape is one of the most recognizable figures from the early days of computer graphics. It was originally modeled after a Melitta teapot by Martin Newell at the University of Utah in 1975. It frequently makes appearances in graphics tutorials because of its interesting topology and asymmetry.

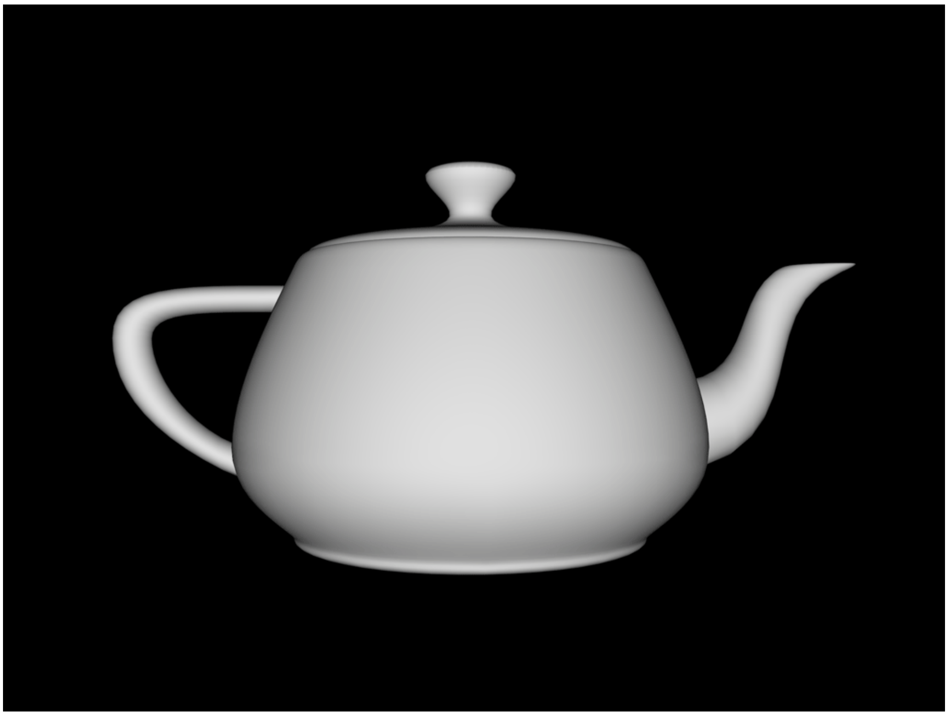


Figure 5.1: A rendered computer model of the Utah teapot

You should feel free to recompile the sample project, replacing the included model with your own OBJ file. Models that are normalized to fit in a $1 \times 1 \times 1$ cube around the origin will be the easiest to work with.

The Structure of OBJ Models

Models in OBJ format are logically divided into *groups*, which are essentially named lists of polygons. Loading a model is as simple as finding the OBJ file in the app bundle, initializing an `MBEOBJModel` object, and asking for a group by index.

```
NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"teapot"
                                          withExtension:@"obj"];
MBEOBJModel *teapot = [[MBEOBJModel alloc] initWithContentsOfURL:modelURL];
MBEOBJGroup *group = [teapot groupAtIndex:1];
```

Index 1 corresponds to the first group in the file; polygons that don't belong to any group are added to an implicit, unnamed group at index 0.

`MBEOBJGroup` is a struct with two pointer members: `vertices` and `indices`. These lists of vertices and indices don't do us any good by themselves; we need to put them into buffers. Since lists of vertices and indices are so often used together, we need an object to hold this pair of buffers. This is what we will call a *mesh*.

The Mesh Class

The interface of the mesh class is very straightforward, consisting of just two buffer properties.

```
@interface MBEMesh : NSObject
@property (readonly) id<MTLBuffer> vertexBuffer;
@property (readonly) id<MTLBuffer> indexBuffer;
@end
```

We subclass the mesh class to produce application-specific mesh types. For now, we need a mesh that represents an OBJ group, so we create an `MBEOBJMesh` class with the following initializer:

```
- (instancetype)initWithGroup:(MBEOBJGroup *)group
                       device:(id<MTLDevice>)device;
```

This method takes an OBJ group and a Metal device and generates a buffer pair with storage for the group's vertices and indices. We can render such a mesh by binding its vertex buffer to the argument table and encoding an indexed draw call as we did in the previous chapter:

```
[renderPass setVertexBuffer:mesh.vertexBuffer offset:0 atIndex:0];
[renderPass drawIndexedPrimitives:MTLPrimitiveTypeTriangle
    indexCount:[mesh.indexBuffer length] / sizeof(MBEIndex)
    indexType:MBEIndexType
    indexBuffer:self.mesh.indexBuffer
    indexBufferOffset:0];
```

Normals

Our lighting calculations will be dependent on the *normal vectors* of our objects. A normal vector is a *unit vector* (vector of length 1) that is perpendicular to a surface. If you imagine a plane that touches a convex surface at just one point (what we call a *tangent* *plane*), the normal vector is the vector that is perpendicular to the plane. It represents the "outward" direction of the surface.

Our OBJ model loader does the work necessary to generate a normal vector for each vertex in the model. We also added a member to the vertex struct to hold the normal, so that its declaration in the shader source now looks like this:

```
struct Vertex
{
    float4 position;
    float4 normal;
};
```

Lighting

In order to start constructing realistic scenes, we need a way to model how light interacts with surfaces. Fortunately, much of computer graphics concerns itself with figuring out clever ways to approximate the behavior of light. We will use an approximation consisting of three terms to model different species of light that occur in a scene: ambient, diffuse, and specular. Each pixel's color will be the sum of these three terms. Here is the equation representing that relationship at the highest level:

$I_{total} = I_{ambient} + I_{diffuse} + I_{specular}$

Above, I stands for *intensity*, specifically, outgoing radiant intensity. We'll use L to signify properties of the light source, and M to signify properties of the material. None of these values have a precise physical basis. Our first attempt at approximating illumination involves a lot of shortcuts. Therefore, the values we choose for these quantities will be based on aesthetics rather than physical exactitude.

The types of lights we will be dealing with are *directional*; they do not have a defined position in space. Instead, we imagine that they are far enough away to be characterized exclusively by the direction in which they emit light. The sun is an example of such a directional light.

Ambient Light

Ambient light is a non-directional kind of light that is not associated with a particular light source. Instead, it is an approximation used to model indirect light (light that has bounced off other surfaces in the scene), which is not captured by the other terms. Ambient lighting prevents the parts of geometry that are not directly lit from being completely black. The contribution of ambient light is normally subtle.

Ambient light is calculated as the product of the ambient intensity of the light and the ambient response of the surface:

$$I_{ambient} = L_{ambient} M_{ambient}$$

It is often the case that one models the ambient contribution as a property of the *scene* rather than a light. This signifies the fact that ambient light doesn't really come from one source; rather, it is the sum of light that has bounced around the scene from whatever light sources are present. We choose to make ambient light a property of the light itself, for the sake of convenience and formulaic consistency.

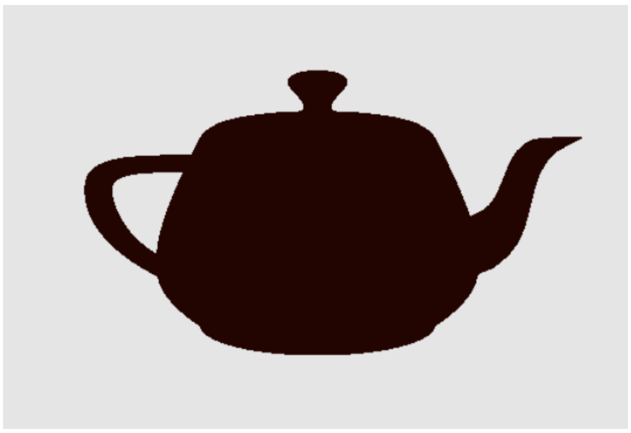


Figure 5.2: The ambient light term usually makes a small contribution

Diffuse Light

Diffuse light follows *Lambert's cosine law*, which states that the intensity of reflected light is directly proportional to the cosine of the angle between the direction of the incident light and the normal of the surface. The normal is a vector that is perpendicular to the surface. Light that is head-on is reflected to a greater degree than light that arrives at a shallow angle of incidence.

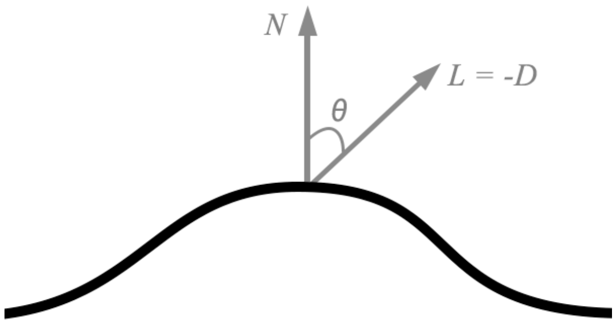


Figure 5.3: In diffuse lighting, the quantity of light reflected is proportional to the angle between the normal and the angle of incidence

Diffuse reflection is modeled by the following simplified equation:

$I_{diffuse} = cos\theta \cdot M_{diffuse} \cdot L_{diffuse}$

If we assume that the normal vector and incident light direction vector are unit vectors, we can use the familiar dot product to calculate this term:

$I_{diffuse} = \mathbf{N} \cdot \mathbf{L} \cdot M_{diffuse} \cdot L_{diffuse}$

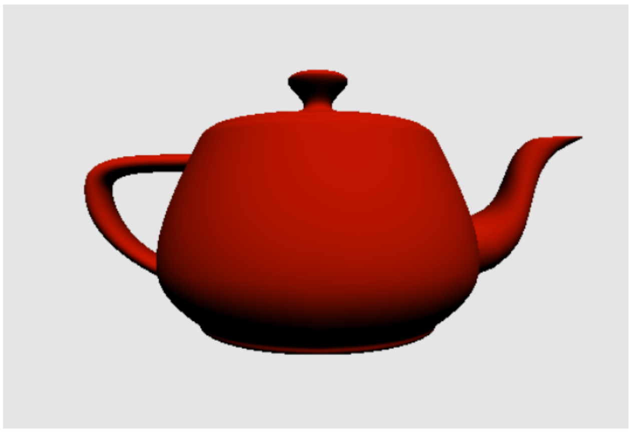


Figure 5.4: The diffuse term represents the tendency of surfaces to reflect light equally in all directions

Specular Light

The specular light term is used to model "shiny" surfaces. It describes the tendency of the material to reflect light in a particular direction rather than scattering it in all directions. Shiny materials create specular *highlights*, which are a powerful visual cue for illustrating how rough or shiny surfaces are. "Shininess" is quantified by a parameter called the *specular power*. For example, a specular power of 5 corresponds with a rather matte surface, while a specular power of 50 corresponds to a somewhat shiny surface.

There are a couple of popular ways to compute the specular term, but here we will use a Blinn-Phong approximation (Blinn 1977). The Blinn-Phong specular term uses a vector called the *halfway vector* that points halfway between the direction to the light source and the direction from which the surface is being viewed:

$$\mathbf{H} = \frac{1}{2}(\mathbf{D} + \mathbf{V})$$

Once we have the halfway vector in hand, we compute the dot product between the the surface normal and the halfway vector, and raise this quantity to the *specular power* of the material.

$$I_{specular} = (\mathbf{N} \cdot \mathbf{H})^{specularPower} L_{specular} M_{specular}$$

This exponentiation is what controls how "tight" the resulting specular highlights are; the higher the specular power, the sharper the highlights will appear.

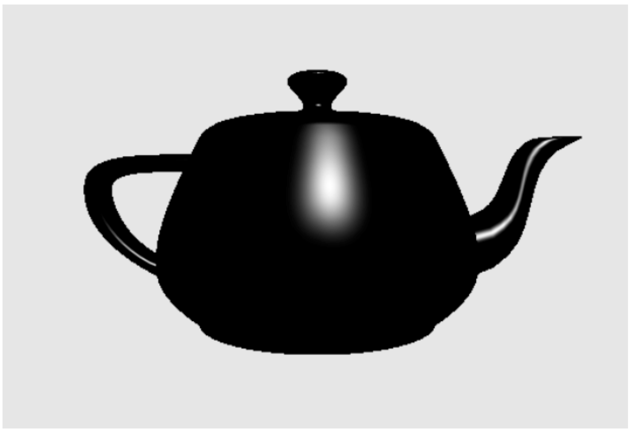


Figure 5.5: The specular term represents the tendency of certain surfaces to reflect light in a particular direction

The Result

Now that we've computed all three terms of our lighting equation, we sum together the results at each pixel to find its final color. In the next few sections, we'll talk about how to actually achieve this effect with the Metal shading language.

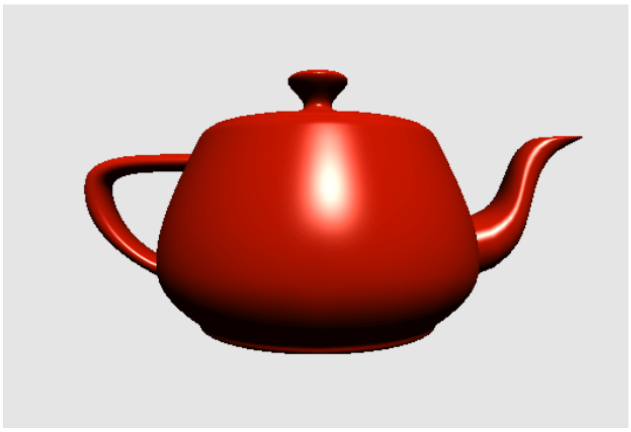


Figure 5.6: Adding up the contributions of ambient, diffuse, and specular terms produces the final image

Lighting in Metal

Representing Lights and Materials

We will use two structs to wrap up the properties associated with lights and materials. A light has three color properties, one for each of the light terms we discussed in detail.

```
struct Light
{
    float3 direction;
    float3 ambientColor;
    float3 diffuseColor;
    float3 specularColor;
};
```

A material also has three color properties; these model the *response* of the material to incoming light. The color of light reflected from a surface is dependent on the color of the surface and the color of the incoming light. As we've seen, the color of light and the color of the surface are multiplied together to determine the color of the surface, and this color is then multiplied by some intensity (1 in the case of ambient, $N \cdot L$ in the case of diffuse, and $N \cdot H$ raised to some power in the case of specular). We model the specular power as a float.

```
struct Material
{
    float3 ambientColor;
    float3 diffuseColor;
    float3 specularColor;
    float specularPower;
};
```

Uniforms

Much as we did in the previous chapter, we need to pass some uniform values to our shaders to transform our vertices. This time, however, we need a couple of additional matrices to do calclulations relevant to lighting.

```
struct Uniforms
{
    float4x4 modelViewProjectionMatrix;
    float4x4 modelViewMatrix;
    float3x3 normalMatrix;
};
```

In addition to the model-view-projection matrix (MVP), we also store the model-view matrix, which transforms vertices from object space to eye space. We need this because specular highlights are view-dependent. We will discuss more of the implications of this when dissecting the shader functions below.

We also include a *normal matrix*, which is used to transform the model's normals into eye space. Why do we need a separate matrix for this? The answer is that normals don't transform like positions. This seems counterintuitive at first, but consider this: when you move an object around in space without rotating it, the "outward" direction at any point on the surface is constant.

This gives us our first clue about how to construct the normal matrix: we need to strip out any translation from the model view matrix. We do this by taking just the upper-left 3×3 portion of our model-view matrix, which is the portion responsible for rotation and scaling.

It turns out that as long as the model-view matrix consists only of rotations and uniform scales (i.e., no stretching or shearing), this upper-left matrix is exactly the correct matrix for transforming normals. In more complex situations, we instead need to use the *inverse transpose* of this matrix, but the reasons for this are beyond the scope of this book.

The Vertex Function

To store the per-vertex values we will need for our lighting calculations, we declare a separate "projected vertex" structure, which will be calculated and returned by the vertex function:

struct ProjectedVertex { float4 position [[position]]; float3 eye; float3 normal; };

The `eye` member is the vector pointing from the vertex's position in view space to the eye of the virtual camera. The `normal` member is the view space normal of the vertex.

The vertex function computes these two new vectors, along with the projected (clip-space) position of each vertex:

```
vertex ProjectedVertex vertex_project(device Vertex *vertices [[buffer(0)]],
    constant Uniforms &uniforms [[buffer(1)]],
    uint vid [[vertex_id]])
{
    ProjectedVertex outVert;
    outVert.position = uniforms.modelViewProjectionMatrix *
        vertices[vid].position;
    outVert.eye =  -(uniforms.modelViewMatrix * vertices[vid].position).xyz;
    outVert.normal = uniforms.normalMatrix * vertices[vid].normal.xyz;

    return outVert;
}
```

The Fragment Function

For each fragment, we receive an interpolated "projected vertex" struct and use it to calculate our three lighting terms:

```
fragment float4 fragment_light(ProjectedVertex vert [[stage_in]],
    constant Uniforms &uniforms [[buffer(0)]])
{
    float3 ambientTerm = light.ambientColor * material.ambientColor;
    float3 normal = normalize(vert.normal);
    float diffuseIntensity = saturate(dot(normal, light.direction));
    float3 diffuseTerm = light.diffuseColor * material.diffuseColor *
        diffuseIntensity;

    float3 specularTerm(0);
    if (diffuseIntensity > 0)
    {
        float3 eyeDirection = normalize(vert.eye);
        float3 halfway = normalize(light.direction + eyeDirection);
        float specularFactor = pow(saturate(dot(normal, halfway)),
            material.specularPower);
        specularTerm = light.specularColor * material.specularColor *
            specularFactor;
    }

    return float4(ambientTerm + diffuseTerm + specularTerm, 1);
}
```

The ambient term, as discussed, is simply the product of the ambient light intensity and the material's ambient response.

Since the normal and eye vectors we receive are not unit-length, we normalize them before use. This has to be done on a per-fragment basis because even if we had normalized them in the vertex function, the interpolation done by the rasterizer does not preserve the unit-length property.

To compute the diffuse term, we first take the dot product of the view-space normal and the light direction. We then use the `saturate` function to clamp it between 0 and 1. Otherwise, faces pointing away from the light would wind up with a *negative* diffuse intensity, which is nonphysical. We multiply this cosine-rule intensity by the product of the light's diffuse contribution and the material's diffuse response to get the complete diffuse term.

To compute the specular factor, we compute the "halfway" vector by summing the light direction and eye direction and normalizing the result. This has the effect of averaging them, since each was unit-length before the addition. We take the dot product between the surface normal and the halfway vector to get the specular base, then raise it to the material's specular power to get the specular intensity. As before, we multiply this with the light's specular color and the material's specular response to get the complete specular term.

Finally, we sum all of three terms together and return the result as the color value of the fragment.

The Sample App

The sample code for this chapter is in the 05-Lighting directory. It loads a teapot model from an OBJ file and shows it tumbling in 3D, much like the cube from the previous chapter. This time, though, per-pixel lighting is applied to the model, giving it a much more plausible (and very shiny) appearance.

Chapter 6

Textures

Textures are a central topic in rendering. Although they have many uses, one of their primary purposes is to provide a greater level of detail to surfaces than can be achieved with vertex colors alone. Using texture maps allows us to specify color on an essentially per-pixel basis, greatly increasing the visually fidelity of our rendered images.

Over the next few chapters we will talk about several topics related to the use of textures in Metal.

In this chapter, we'll talk about texture mapping, which helps us bring virtual characters to life. We'll also introduce *samplers*, which give us powerful control over how texture data is interpreted while drawing. Along the way, we will be assisted by a cartoon cow named Spot.

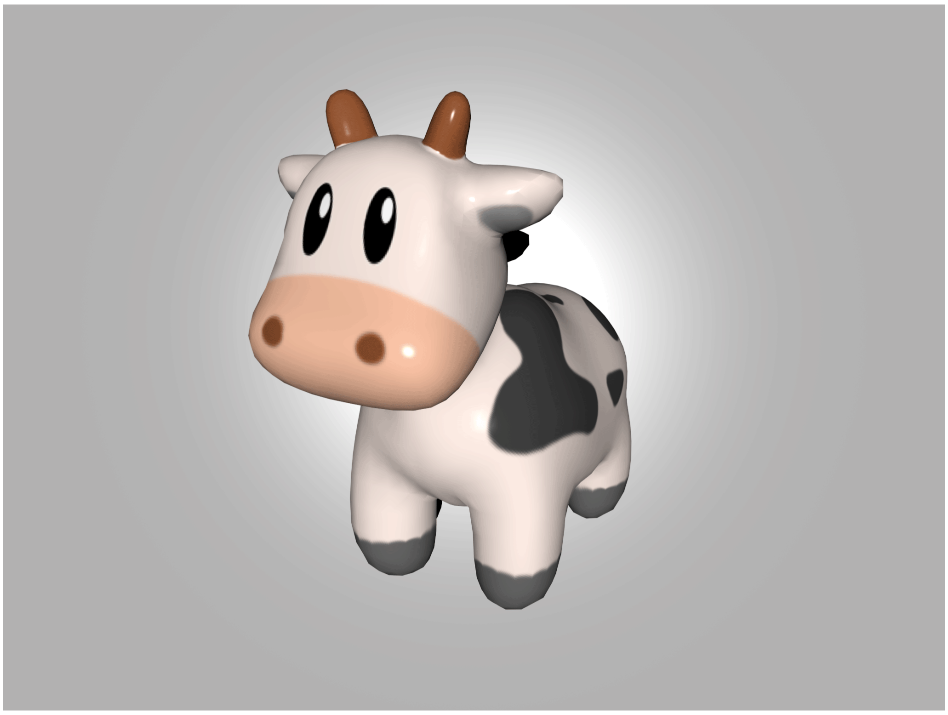


Figure 6.1: Meet Spot. (Model provided by Keenan Crane)

Textures

Textures are formatted image data. This makes them distinct from buffers, which are unstructured blocks of memory. The types of textures we will be working with in this chapter are 2D images. Although Metal supports several other kinds of textures, we can introduce almost all of the important concepts by walking through an example of texture mapping.

Texture Mapping

Texture mapping is the process of associating each vertex in a mesh with a point in a texture. This is similar to wrapping a present, in that a 2D sheet of wrapping paper (a texture) is made to conform to a 3D present (the mesh).

Texture mapping is usually done with a specialized editing tool. The mesh is *unwrapped* into a planar graph (a 2D figure that maintains as much of the connectivity of the 3D model as possible), which is then overlaid on the texture image.

The following figure shows a 3D model of a cow (with the edges of its constituent faces highlighted) and its corresponding texture map. Notice that the various parts of the cow (head, torso, legs, horns) have been separated on the map even though they are connected on the 3D model.

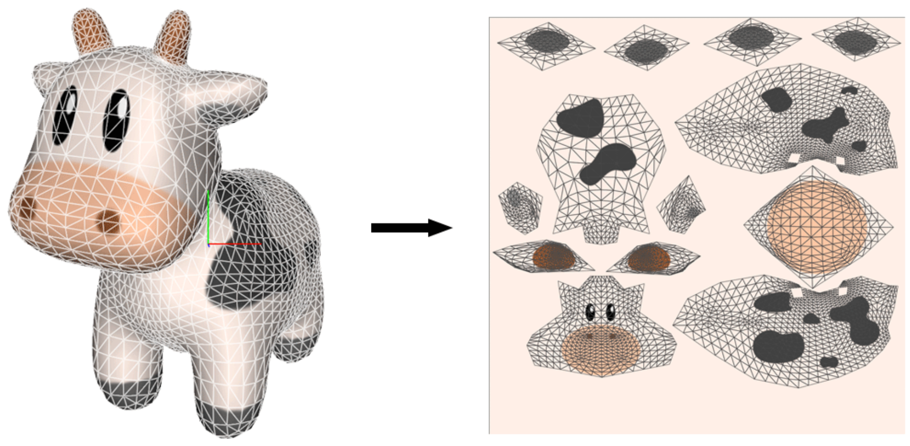


Figure 6.2: How to unwrap a cow. The mesh is flattened into a 2D surface to make it easier to assign texture coordinates to vertices

Coordinate Systems

In Metal, the origin of the pixel coordinate system of a texture coincides with its top left corner. This is the same as the coordinate system in UIKit. However, it differs from the default texture coordinate system in OpenGL, where the origin is the bottom left corner.

The axes of the texture coordinate system are often labeled `s` and `t` (or `u` and `v`) to distinguish them from the `x` and `y` axes of the world coordinate system.

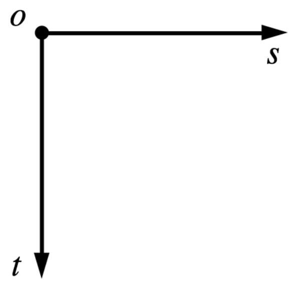


Figure 6.3: Metal's texture coordinate system has its origin in the top left corner.

The coordinate system of the texture coordinates must agree with the coordinate system of the image data. Core Graphics contexts follow the bottom-left origin convention, so in order to get image data that is suitable to use with Metal's upper-left origin, we will draw into a "flipped" Core Graphics context. The code for this is shown in a later section.

Pixel Coordinates Versus Normalized Coordinates

Metal is flexible enough to allow us to specify texture coordinates in pixel coordinates or normalized coordinates. Pixel coordinates range from $(0, 0)$ to $(width - 1, height - 1)$. They therefore depend on the dimensions of the texture image. Normalized coordinates range from $(0, 0)$ to $(1, 1)$, which makes them independent of image size.

We have chosen to use normalized coordinates throughout this chapter and the rest of the book.

Filtering

Textures are discrete images composed of a finite number of pixels (called *texels*). However, when drawing, a texture may be drawn at a resolution that is higher or lower than its native size. Therefore, it is important to be able to determine what the color of a texture should be *between* its texels, or when many texels are crunched into the same space. When a texture is being drawn at a size higher than its native size, this is called *magnification*. The inverse process, where a texture is drawn below its native resolution, is called *minification*.

The process of reconstructing image data from a texture is called *filtering*. Metal offers two different filtering modes: *nearest* and *linear*.

Nearest (also called "nearest-neighbor") filtering simply selects the closest texel to the requested texture coordinate. This has the advantage of being very fast, but it can cause the rendered image to appear blocky when textures are magnified (i.e., when each texel covers multiple pixels).

Linear filtering selects the four nearest texels and produces a weighted average according to the distance from the sampled coordinate to the texels. Linear filtering produces much more visually-pleasing results than nearest-neighbor filtering and is sufficiently fast to be performed in real-time.

Mipmaps

When a texture is minified, multiple texels may coincide with a single pixel. Even slight motion in the scene can cause a shimmering phenomenon to appear. Linear filtering does not help the situation, as the set of texels covering each pixel changes from frame to frame.

One way to abate this issue is to prefilter the texture image into a sequence of images, called *mipmaps*. Each mipmap in a sequence is half the size of the previous image, down to a size of 1×1. When a texture is being minified, the mipmap at the closest resolution is substituted for the original texture image.

Mipmapping is covered in-depth in the next chapter.

Addressing

Typically, when associating texture coordinates with the vertices of a mesh, the values are constrained to $[0, 1]$ along both axes. However, this is not always the case. Negative texture coordinates, or coordinates greater than 1 can also be used. When coordinates outside the $[0, 1]$ range are used, the *addressing mode* of the sampler comes into effect. There are a variety of different behaviors that can be selected.

Clamp-to-Edge Addressing

In clamp-to-edge addressing, the value along the edge of the texture is repeated for out-of-bounds values.

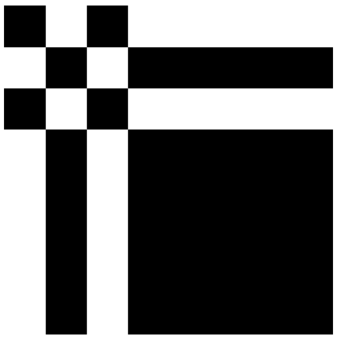


Figure 6.4: Clamp-to-edge filtering repeats the outer edge of the texture image for out of bounds coordinates

Clamp-to-Zero Addressing

In clamp-to-zero addressing, the sampled value for out-of-bounds coordinates is either black or clear, depending on whether the texture has an alpha color component.

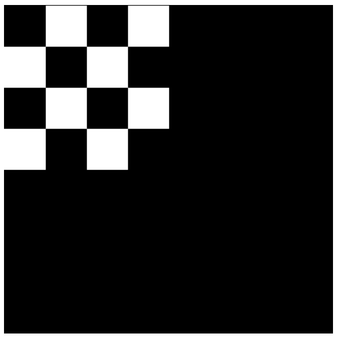


Figure 6.5: Clamp-to-zero filtering produces a black color outside the bounds of the texture image

Repeat Addressing

In repeat addressing, out-of-bounds coordinates wrap around the corresponding edge of the texture and repeat from zero. In other words, the sampled coordinates are the fractional parts of the input coordinates, ignoring the integer parts.

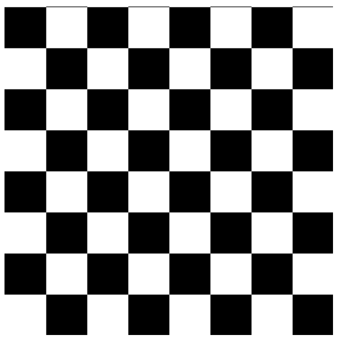


Figure 6.6: Repeat addressing tiles the texture by repeating it

Mirrored Repeat Addressing

In mirrored repeat addressing, the sampled coordinates first increase from 0 to 1, then decrease back to 0, and so on. This causes the texture to be flipped and repeated across every other integer boundary.

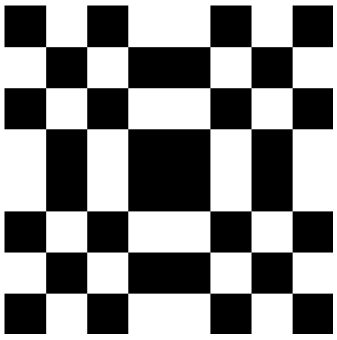


Figure 6.7: Mirrored repeat addressing first repeats the texture mirrored about the axis, then repeats it normally, and so on

Creating Textures in Metal

Before we actually ask Metal to create a texture object, we need to know a little more about pixel formats, as well as how to get texture data into memory.

Pixel Formats

A *pixel format* describes the way color information is laid out in memory. There are a few different aspects to this information: the color *components*, the color component *ordering*, the color component *size*, and the presence or absence of *compression*.

The common color components are red, green, blue, and alpha (transparency). These may all be present (as in an RGBA format), or one or more may be absent. In the case of a fully-opaque image, alpha information is omitted.

Color component ordering refers to which color components appear in memory: BGRA or RGBA.

Colors may be represented with any degree of precision, but two popular choices are 8 bits per component and 32 bits per component. Most commonly, when 8 bits are used, each component is an unsigned 8-bit integer, a value between 0 and 255. When 32 bits are used, each component is usually a 32-bit float ranging from 0.0 to 1.0. Obviously, 32 bits offer far greater precision than 8 bits, but 8 bits is usually sufficient for capturing the perceivable differences between colors, and is much better from a memory standpoint.

The pixel formats supported by Metal are listed in the `MTLPixelFormat` enumeration. The supported formats vary based on operating system version and hardware, so consult the Metal Programming Guide for further details.

Loading Image Data

We will use the powerful utilities provided by UIKit to load images from the application bundle. To create a `UIImage` instance from an image in the bundle, we only need one line of code:

```
UIImage *image = [UIImage imageNamed:@"my-texture-name"];
```

Unfortunately, UIKit does not provide a way to access the underlying bits of a `UIImage`. Instead, we have to draw the image into a Core Graphics bitmap context that has the same format as our desired texture. As part of this process, we transform the context (with a translation followed by a scale) such that the resulting image bits are flipped vertically. This causes the coordinate space of our image to agree with Metal's texture coordinate space.

```
CGImageRef imageRef = [image CGImage];

// Create a suitable bitmap context for extracting the bits of the image
NSUInteger width = CGImageGetWidth(imageRef);
NSUInteger height = CGImageGetHeight(imageRef);
```

```
CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
uint8_t *rawData = (uint8_t *)calloc(height * width * 4, sizeof(uint8_t));
NSUInteger bytesPerPixel = 4;
NSUInteger bytesPerRow = bytesPerPixel * width;
NSUInteger bitsPerComponent = 8;
CGContextRef context = CGBitmapContextCreate(rawData, width, height,
    bitsPerComponent, bytesPerRow, colorSpace,
    kCGImageAlphaPremultipliedLast | kCGBitmapByteOrder32Big);
CGColorSpaceRelease(colorSpace);

// Flip the context so the positive Y axis points down
CGContextTranslateCTM(context, 0, height);
CGContextScaleCTM(context, 1, -1);

CGContextDrawImage(context, CGRectMake(0, 0, width, height), imageRef);
CGContextRelease(context);
```

This code is wrapped in the `MTLTextureLoader` utility class.

Texture Descriptors

A *texture descriptor* is a lightweight object that specifies the dimensions and format of a texture. When creating a texture, you provide a texture descriptor and receive an object that conforms to the `MTLTexture` protocol, which is a subprotocol of `MTLResource`. The properties specified on the texture descriptor (texture type, dimensions, and format) are immutable once the texture has been created, but you can still update the *content* of the texture as long as the pixel format of the new data matches the pixel format of the receiving texture.

The `MTLTextureDescriptor` class provides a couple of factory methods for building common texture types. To describe a 2D texture, you must specify the pixel format, texture dimensions in pixels, and whether Metal should allocate space to store the appropriate mipmap levels for the texture.

```
[MTLTextureDescriptor
     texture2DDescriptorWithPixelFormat:MTLPixelFormatRGBA8Unorm
    width:width
    height:height
    mipmapped:YES];
```

Creating a Texture Object

It is now quite straightforward to create a texture object. We simply request a texture from the device by supplying a valid texture descriptor:

```
id<MTLTexture> texture = [self.device
    newTextureWithDescriptor:textureDescriptor];
```

Updating Texture Contents

Setting the data in the texture is also quite simple. We create a `MTLRegion` that represents the entire texture and then tell the texture to replace that region with the raw image bits we previously retrieved from the context:

```
MTLRegion region = MTLRegionMake2D(0, 0, width, height);
[texture replaceRegion:region mipmapLevel:0 withBytes:rawData
    bytesPerRow:bytesPerRow];
```

Passing Textures to Shader Functions

This texture is now ready to be used in a shader. To pass a texture to a shader function, we set it on our command encoder right before we issue our draw call. Textures have their own slots in the argument table separate from buffers, so the indices we use start from 0.

```
[commandEncoder setFragmentTexture:texture atIndex:0];
```

This texture can now be bound to a parameter with the attribute `[[texture(0)]]` in a shader function's parameter list.

Samplers

In Metal, a *sampler* is an object that encapsulates the various render states associated with reading textures: coordinate system, addressing mode, and filtering. It is possible to create samplers in shader functions or in application code. We will discuss each in turn in the following sections.

Creating Samplers in Shaders

We will use samplers in our fragment function, because we want to produce a different color for each pixel in the rendered image. So, it sometimes makes sense to create samplers directly inside a fragment function.

The following code creates a sampler that will sample in the `normalized` coordinate space, using the `repeat` addressing mode, with `linear` filtering:

```
constexpr sampler s(coord::normalized,
                    address::repeat,
                    filter::linear);
```

Samplers that are local to a shading function must be qualified with `constexpr`. This keyword, new in C++11, signifies that an expression may be computed at compile-time rather than runtime. This means that just one sampler struct will be created for use across all invocations of the function.

The `coord` value may either be `normalized` or `pixel`.

The possible values for `address` are `clamp_to_zero`, `clamp_to_edge`, `repeat`, `mirrored_repeat`.

The possible values for `filter` are `nearest` and linear', to select between nearest-neighbor and linear filtering.

All of these values belong to strongly-typed enumerations. Strongly typed enumerations are a new feature in C++11 that allow stricter type-checking of enumerated values. It is an error to omit the type name of the value (i.e., you must say `filter::linear` and not simply `linear`).

The parameters may be specified in any order, since the constructor of the sampler is implemented as a variadic template function (another new feature of C++11).

Using a Sampler

Getting a color from a sampler is straightforward. Textures have a `sample` function that takes a sampler and a set of texture coordinates, returning a color. In a shader function, we call this function, passing in a sampler and the (interpolated) texture coordinates of the current vertex.

```
float4 sampledColor = texture.sample(sampler, vertex.textureCoords);
```

The sampled color may then be used in whatever further computation you want, such as lighting.

Creating Samplers in Application Code

To create a sampler in application code, we fill out a `MTLSamplerDescriptor` object and the ask the device to give us a sampler (an object conforming to `MTLSamplerState`).

```
MTLSamplerDescriptor *samplerDescriptor = [MTLSamplerDescriptor new];
samplerDescriptor.minFilter = MTLSamplerMinMagFilterNearest;
samplerDescriptor.magFilter = MTLSamplerMinMagFilterLinear;
samplerDescriptor.sAddressMode = MTLSamplerAddressModeClampToEdge;
samplerDescriptor.tAddressMode = MTLSamplerAddressModeClampToEdge;

samplerState = [device newSamplerStateWithDescriptor:samplerDescriptor];
```

Note that we had to individually specify the magnification and minification filters. When creating samplers in shader code, we used the `filter` parameter to specify both at once, but we could also use `mag_filter` and `min_filter` separately to get the same behavior as above.

Similarly, the address mode must be set separately for each texture axis, whereas we did this with the sole `address` parameter in shader code.

Passing Samplers as Shader Arguments

Passing samplers looks very similar to passing textures, but samplers reside in yet another set of argument table slots, so we use a different method to bind them to indices starting at 0:

```
[commandEncoder setFragmentSamplerState:samplerState atIndex:0];
```

We can now refer to this sampler by attributing it with `[[sampler(0)]]` in shader code.

The Sample Project

The sample project is in the 06-Texturing directory. It borrows heavily from prior chapters, so the shared concepts are not reiterated in this chapter.

The major change is the use of a texture/sampler pair to determine the diffuse color of the mesh at each pixel instead of interpolating a single color across the whole surface. This allows us to draw our textured cow model, producing a greater sense of realism and detail in the scene.

Chapter 7

Mipmapping

In this chapter we will learn about mipmapping, an important technique for rendering textured objects at a distance. We will find out why mipmapping is important, how it complements regular texture filtering, and how to use the blit command encoder to generate mipmaps efficiently on the GPU.

A Review of Texture Filtering

In the previous chapter we used texture filtering to describe how texels should be mapped to pixels when the screen-space size of a pixel differs from the size of a texel. This is called *magnification* when each texel maps to more than one pixel, and *minification* when each pixel maps to more than one texel. In Metal, we have a choice of what type of filtering to apply in each of these two regimes.

Nearest filtering just selects the closest texel to represent the sampled point. This results in blocky output images, but is computationally cheap. *Linear* filtering selects four adjacent texels and produces a weighted average of them.

In Metal, we specify type of filtering to use in the `minFilter` and `magFilter` properties of a `MTLSamplerDescriptor`.

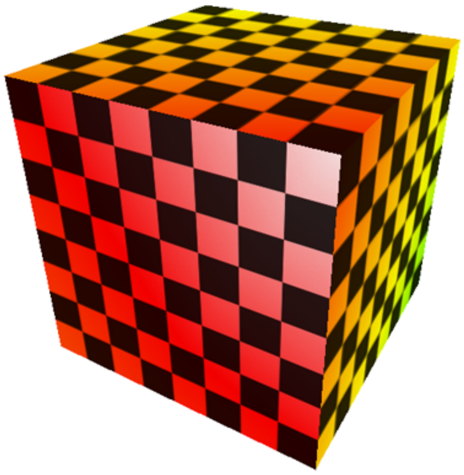


Figure 7.1: The sample app illustrates the effect of various mipmapping filters. The mipmap used here is artificially colored for demonstration purposes.

Mipmap Theory

The name “mipmap” comes from the Latin phrase “multum in parvo”, roughly meaning “much in little”. This alludes to the fact that each texel in a mipmap combines several of the texels in the level above it. Before we talk about how to build mipmaps, let’s spend some time talking about why we need them in the first place.

The Aliasing Problem

You might think that because we’ve handled the case of texture minification and magnification, our texture mapping should be perfect and free of visual artifacts. Unfortunately, there is a sinister effect at play when a texture is minified beyond a certain factor.

As the virtual camera pans across the scene, a different set of texels are used each frame to determine the color of the pixels comprising distant objects. This occurs regardless of the minification filter selected. Visually, this produces unsightly shimmering. The problem is essentially one of *undersampling* a high-frequency signal (the signal in this case is the texture). If there were a way to smooth the texture out in the process of sampling, we could instead trade a small amount of blurriness for a significant reduction in the distracting shimmering effect during motion.

The difference between using a linear minification filter and a linear minification filter combined with a mipmap is shown below, to motivate further discussion.

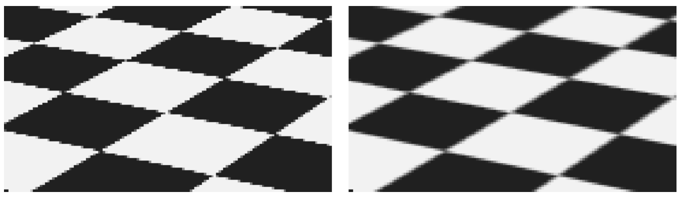


Figure 7.2: A side-by-side comparison of basic linear filtering and mipmapped linear filtering

The Mipmap Solution

Mipmapping is a technique devised to solve this aliasing problem. Rather than downsampling the image on the fly, a sequence of prefiltered images (*levels*) are generated, either offline or at load time. Each level is a factor of two smaller (along each dimension) than its predecessor. This leads to a 33% increase in memory usage for each texture, but can greatly enhance the fidelity of the scene in motion.

In the figure below, the levels generated for a checkerboard texture are shown.

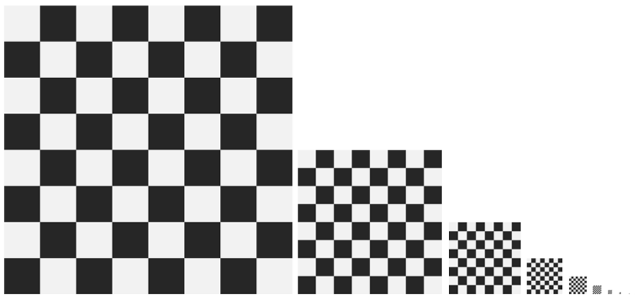


Figure 7.3: The various levels that comprise a mipmap. Each image is half the size of its predecessor, and ¼ the area, down to 1 pixel.

Mipmap Sampling

When a mipmapped texture is sampled, the projected area of the fragment is used to determine which mipmap level most nearly matches the texture's texel size. Fragments that are smaller, relative to the texel size, use mipmap levels that have been reduced to a greater degree.

In Metal, the mipmap filter is specified separately from the minification filter, in the `mipFilter` property of the descriptor, but the min and mip filters interact to create four possible scenarios. They are described below in order of increasing computational cost.

When `minFilter` is set to `MTLSamplerMinMagFilterNearest` and `mipFilter` is set to `MTLSamplerMipFilterNearest`, the closest-matching mipmap level is selected, and a single texel from it is used as the sample.

When `minFilter` is set to `MTLSamplerMinMagFilterNearest` and `mipFilter` is set to `MTLSamplerMipFilterLinear`, the two closest-matching mipmap levels are selected, and one sample from each is taken. These two samples are then averaged to produce the final sample.

When `minFilter` is set to `MTLSamplerMinMagFilterLinear` and `mipFilter` is set to `MTLSamplerMipFilterNearest`, the closest-matching mipmap level is selected, and four texels are averaged to produce the sample.

When `minFilter` is set to `MTLSamplerMinMagFilterLinear` and `mipFilter` is set to `MTLSamplerMipFilterLinear`, the two closest-matching mipmap levels are selected, and four samples from each are averaged to create a sample for the level. These two averaged samples are then averaged again to produce the final sample.

The figure below shows the difference between using a nearest and linear mip filter when a linear min filter is used:

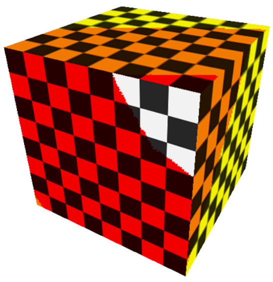


Figure 7.4: The effect of using a linear min filter and nearest mip filter. Mip levels have been artificially colored to show different levels more clearly.

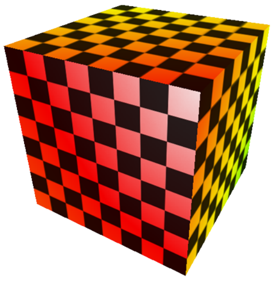


Figure 7.5: The effect of using a linear min filter and linear mip filter. Mip levels have been artificially colored to show different levels more clearly.

Mipmapped Textures in Metal

Building a mipmapped texture in Metal is a two-part process: creating the texture object and copying image data into the mipmap levels. Metal does not automatically generate mipmap levels for us, so we'll look at two ways to do the generation ourselves below.

Creating the Texture

We can use the same convenience method for creating a 2D mipmapped texture descriptor as for non-mipmapped textures, passing `YES` as the final parameter.

```
[MTLTextureDescriptor
     texture2DDescriptorWithPixelFormat:MTLPixelFormatRGBA8Unorm
    width:width
    height:height
    mipmapped:YES];
```

When the `mipmapped` parameter is equal to `YES`, Metal computes the `mipmapLevelCount` property of the descriptor. The formula used to find the number of levels is $floor(log_2(max(width, height))) + 1$. For example, a 512×256 texture has 10 levels.

Once we have a descriptor, we request a texture object from the Metal device:

```
id<MTLTexture> texture = [device newTextureWithDescriptor:descriptor];
```

Generating Mipmap Levels Manually

Creating mipmap levels involves creating smaller and smaller versions of the base image, until a level has a dimension that is only one pixel in size.

iOS and OS X share a framework called Core Graphics that has low-level utilities for drawing shapes, text, and images. Generating a mipmap level consists of creating a bitmap context with `CGBitmapContextCreate`, drawing the base image into it with `CGContextDrawImage`, and then copying the underlying data to the appropriate level of the Metal texture as follows:

```
MTLRegion region = MTLRegionMake2D(0, 0, mipWidth, mipWidth);
[texture replaceRegion:region
    mipmapLevel:level
    withBytes:mipImageData
    bytesPerRow:mipWidth * bytesPerPixel]
```

For level 1, `mipWidth` and `mipHeight` are equal to half of the original image size. Each time around the mipmap generation loop, the width and height are halved, `level` is incremented, and the process repeats until all levels have been generated.

In the sample app, a tint color is applied to each mipmap level generated with Core Graphics so that they can be easily distinguished. The figure below shows the images that comprise the checkerboard texture, as generated by Core Graphics.

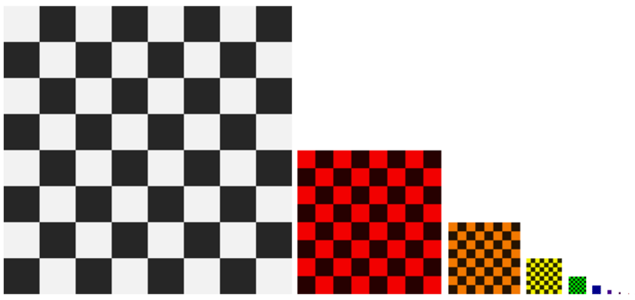


Figure 7.6: The mipmap levels generated by the CPU; a tint color has been applied to each level to distinguish it when rendered

The Blit Command Encoder

The chief disadvantage to generating mipmaps on the CPU is speed. Using Core Graphics to scale the image down can easily take **ten times longer** than using the GPU. But how do we offload the work to the GPU? Happily, Metal includes a special type of command encoder whose job is to leverage the GPU for image copying and resizing operations: the *blit command encoder*. The term "blit" is a derivative of the phrase "block transfer."

Capabilities of the Blit Command Encoder

Blit command encoders enable hardware-accelerated transfers among GPU resources (buffers and textures). A blit command encoder can be used to fill a buffer with a particular value, copy a portion of one texture into another, and copy between a buffer and texture.

We won't explore all of the features of the blit command encoder in this chapter. We'll just use it to generate all of the levels of a mipmap.

Generating Mipmaps with the Blit Command Encoder

Generating mipmaps with a blit command encoder is very straightforward, since there is a method on the `MTLBlitCommandEncoder` protocol named `generateMipmapsForTexture:`. After calling this method, we add a completion handler so we know when the command finishes. The process is very fast, taking on the order of one millisecond for a 1024×1024 texture on the A8 processor.

```
id<MTLBlitCommandEncoder> commandEncoder = [commandBuffer blitCommandEncoder];
[commandEncoder generateMipmapsForTexture:texture];
[commandEncoder endEncoding];
[commandBuffer addCompletedHandler:^(id<MTLCommandBuffer> buffer) {
    // texture is now ready for use
}];
[commandBuffer commit];
```

When the completion block is called, the texture is ready to be used for rendering.

The Sample App

The sample code for this chapter is in the 07-Mipmapping directory.

The sample app shows a rotating, textured cube. You can use a tap gesture to alternate among several different modes: no mipmapping, mipmapping with a GPU-generated texture, mipmapping with a CPU-generated texture and linear mip filtering, and mipmapping with a CPU-generated texture and nearest mip filtering. The CPU-generated texture has a differently-colored tint applied to each level to make it obvious which levels are being sampled.

You can use a pinch gesture to zoom the cube closer and farther, which will cause different mipmap levels to be sampled, if mipmapping is enabled. You can also observe the degradation caused by not using mipmaps when the cube is nearly edge-on, or as it moves away from the camera.

Chapter 8

Reflection and Refraction with Cube Maps

In this chapter, we'll talk about some of the more advanced features of texturing in Metal. We'll apply a cube map to a skybox to simulate a detailed environment surrounding the scene. We'll also introduce a technique called *cube environment mapping* to simulate reflection and refraction, to further enhance the realism of our virtual world.

First, let's talk about the sample scene.

Setting the Scene

The scene for the sample app consists of a *skybox* surrounding a *torus knot*. A skybox is a cube that has a texture applied to the inside to create the illusion of a large outdoor environment surrounding the camera. You are probably familiar with the torus, a geometric figure that consists of a circle swept perpendicularly around another circle. A torus knot introduces an additional complication by twisting the path taken by the orbiting circle, creating a figure that is more visually interesting.

Instead of loading models from disk to construct the sample scene, we will be generating the necessary geometry procedurally.

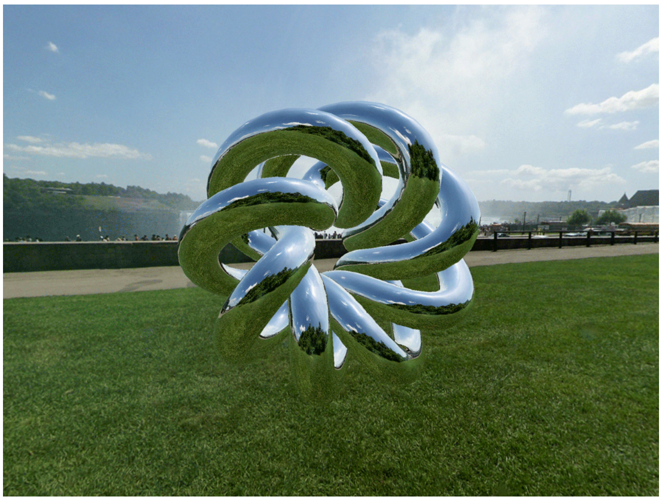


Figure 8.1: The environment map reflected off a torus knot. Textures courtesy of Emil Persson (http://humus.name)

The Vertex Format

Although the objects in our scene will be textured, we do not need to explicitly store texture coordinates in our vertices. Instead, we will generate texture coordinates for the skybox and torus knot in specially-crafted vertex shaders. Therefore, each vertex has only two properties: `position` and `normal`:

```
typedef struct
{
    vector_float4 position;
    vector_float4 normal;
} MBEVertex;
```

The Mesh Class

As we did previously, we use a mesh object to wrap up a vertex buffer and an index buffer so that we can conveniently issue indexed draw calls. The base class, `MBEMesh`, provides an abstract interface to these two buffers.

The Skybox Mesh

Generating the skybox mesh is straightforward. The skybox is a unit cube about the origin, so the skybox class uses a static list of positions, normals, and indices to build its buffers. Note that the indices are ordered to produce cube faces that point *inward*, since the virtual camera will be *inside* the skybox at all times.

Figure 8.2: A cubic environment map applied to a skybox can produce the illusion of a detailed backdrop.

The Torus Knot Mesh

The `MBETorusKnotMesh` class generates vertices as it sweeps along the path of a torus knot, then generates indices that knit together the vertices into a solid mesh. You can produce a large variety of interesting knots by varying the parameters passed to the mesh's initializer method.

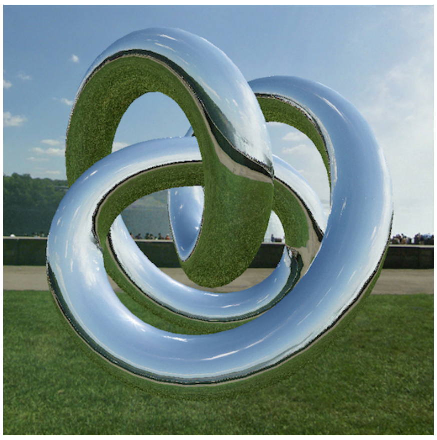


Figure 8.3: This trefoil knot is one kind of knot that can be produced by the torus knot mesh class.

Now that we have some geometry to render, let's talk about a new kind of texture in Metal.

Cube Textures

Cube textures (also called "cube maps") are a special type of texture. In contrast to regular 2D textures, cube textures are collections of six images that have a spatial relationship to one another. In particular, each image corresponds to a direction along one of the axes of the 3D coordinate system.

In the figure below, the six textures comprising a cube map are laid out to show the cube from the inside. The -Y face is in the middle of the cross.

The Cube Map Coordinate Space

Cube maps in Metal are *left-handed*, meaning that if you imagine yourself inside the cube, with the +X face to your right, and the +Y face above you, the +Z axis will be the one directly ahead. If instead you are positioned outside the cube, the coordinate system appears to be right-handed, and all of the images are mirrored.

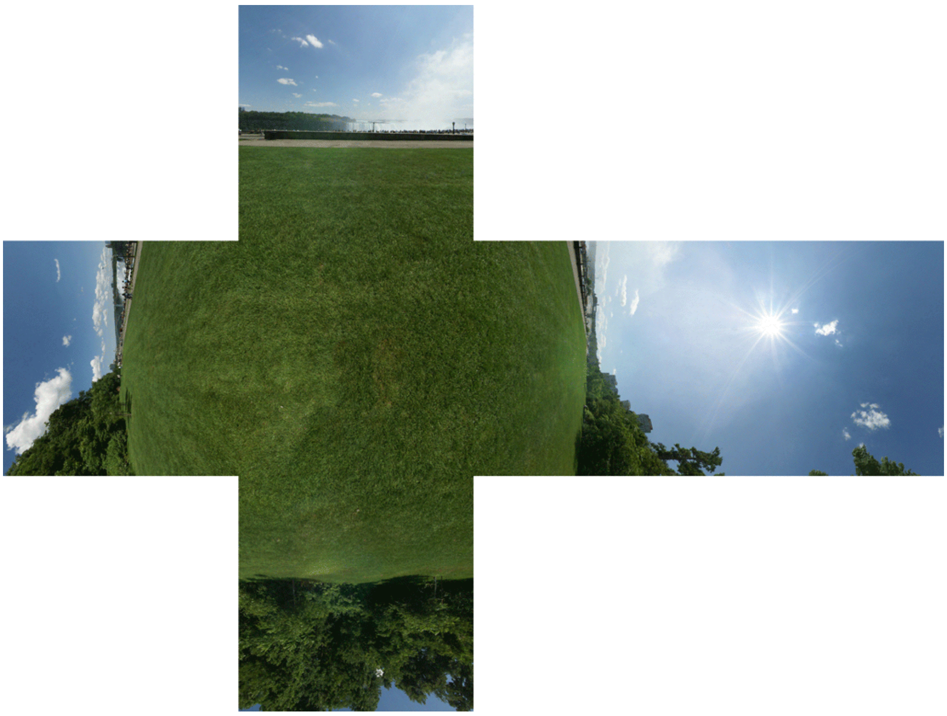


Figure 8.4: An unwrapped cube map.

Creating and Loading a Cube Map Texture

Creating a cube texture and loading it with image data is similar to creating and loading a 2D texture, with a couple of exceptions.

The first difference is the texture type. `MTLTextureDescriptor` has a convenience method for creating a texture descriptor describing a cube texture. The `textureType` property of the descriptor is set to `MTLTextureTypeCube`, and the `height` and `width` properties are set equal to the `size` parameter. The width and height of all constituent textures of the cube texture must be equal.

```
[MTLTextureDescriptor
     textureCubeDescriptorWithPixelFormat:MTLPixelFormatRGBA8Unorm
    size:cubeSize
    mipmapped:NO];
```

Creating the texture object from the descriptor is done in the same manner as when creating a 2D texture:

```
id<MTLTexture> texture = [device newTextureWithDescriptor:textureDescriptor];
```

The other difference between 2D textures and cube textures is how image data is loaded. Because a cube texture actually contains six distinct images, we have to specify a `slice` parameter when setting the pixel data for each face. Each slice corresponds to one cube face, with the following mapping:

Face number	Cube texture face
0	Positive X
1	Negative X
2	Positive Y
3	Negative Y
4	Positive Z
5	Negative Z

To load the data into every cube face, we iterate over the slices, replacing the slice's entire region with the appropriate image data:

```
for (int slice = 0; slice < 6; ++slice)
{
    // ...
    // fetch image data for current cube face
    // ...

    [texture replaceRegion:region
               mipmapLevel:0
                     slice:slice
                 withBytes:imageData
               bytesPerRow:bytesPerRow
             bytesPerImage:bytesPerImage];
}
```

`imageData` must be in whatever pixel format was specified in the texture descriptor. `bytesPerRow` is the width of the cube multiplied by the number of bytes per pixel. `bytesPerImage`, in turn, is the number of bytes per row multiplied by the height of the cube. Since the width and height are equal, you can substitute the cube side length in these calculations.

Sampling a Cube Map

Cube map texture coordinates work somewhat differently than 2D texture coordinates. The first difference is that three coordinates are required to sample a cube map, rather than two. The three coordinates are treated as a ray originating at the center of the cube, intersecting the face at a particular point on the cube. In this way, cube texture coordinates represent a *direction* rather than a particular point.

Some interesting edge cases can arise when sampling in this way. For example, the three vertices of a triangle might span the corner of the cube map, sampling three different faces. Metal handles such cases automatically by correctly interpolating the samples along the edges and corners of the cube.

Applications of Cube Maps

Now that we know how to create, load, and sample a cube map, let's talk about some of the interesting effects we can achieve with cube maps. First, we need to texture our skybox.

Texturing the Skybox

Recall that we created our skybox mesh from a set of static data representing a unit cube about the origin. The positions of the corners of a unit cube map exactly to the corner texture coordinates of a cube map, so it is possible to reuse the cube's vertex positions as its texture coordinates, ignoring the face normals.

Drawing the Skybox

When rendering a skybox, one very important consideration is its effect on the depth buffer. Since the skybox is intended to be a backdrop for the scene, all of the other objects in the scene must appear in front of it, lest the illusion be broken. For this reason, the skybox is always drawn before any other objects, and writing to the depth buffer is *disabled*. This means that any objects drawn after the skybox replace its pixels with their own.

Drawing the skybox first also introduces a small optimization: because the skybox covers every pixel on the screen, it is not necessary to clear the render buffer's color texture before drawing the scene.

The Skybox Vertex Shader

The vertex shader for the skybox is very straightforward. The position is projected according to the combined model-view-projection matrix, and the texture coordinates are set to the model space positions of the cube corners.

```
vertex ProjectedVertex vertex_skybox(device Vertex *vertices     [[buffer(0)]],
                                     constant Uniforms &uniforms [[buffer(1)]],
                                     uint vid                    [[vertex_id]])
{
    float4 position = vertices[vid].position;

    ProjectedVertex outVert;
    outVert.position = uniforms.modelViewProjectionMatrix * position;
    outVert.texCoords = position;
    return outVert;
}
```

The Fragment Shader

We will use the same fragment shader for both the skybox and the torus knot figure. The fragment shader simply inverts the z-coordinate of the texture coordinates and then samples the texture cube in the appropriate direction:

```
fragment half4 fragment_cube_lookup(ProjectedVertex vert [[stage_in]],
    constant Uniforms &uniforms   [[buffer(0)]],
    texturecube<half> cubeTexture [[texture(0)]],
    sampler cubeSampler           [[sampler(0)]])
{
    float3 texCoords = float3(vert.texCoords.x, vert.texCoords.y,
         -vert.texCoords.z);
    return cubeTexture.sample(cubeSampler, texCoords);
}
```

We invert the z coordinate in this case because the coordinate system of the cube map is left-handed as viewed from the inside (borrowing OpenGL's convention), but we prefer a right-handed coordinate system. This is simply the convention adopted by the sample app; you may prefer to use left-handed coordinates. Consistency and correctness are more important than convention.

The Physics of Reflection

Reflection occurs when light bounces off of a reflective surface. Here, we will only consider perfect mirror reflection, the case where all light bounces off at an angle equal to its incident angle. To simulate reflection in a virtual scene, we run this process backward, tracing a ray from the virtual camera through the scene, reflecting it off a surface, and determining where it intersects our cube map.

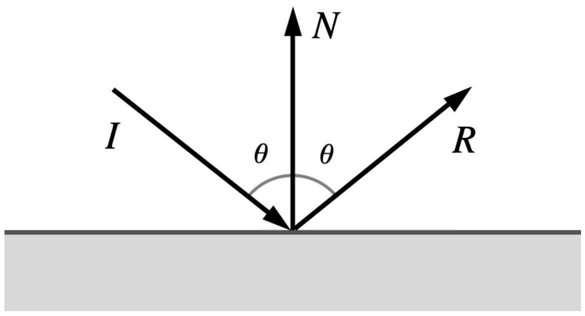


Figure 8.5: Light is reflected at an angle equal to its angle of incidence

The Reflection Vertex Shader

To compute the cube map texture coordinates for a given vector, we need to find the vector that points from the virtual camera to the current vertex, as well as its normal. These vectors must both be relative to the world coordinate system. The uniforms we pass into the vertex shader include the model matrix as well as the normal matrix (which is the inverse transpose of the model matrix). This allows us to compute the necessary vectors for finding the reflected vector according to the formula given above.

Fortunately, Metal includes a built-in function that does the vector math for us. `reflect` takes two parameters: the incoming vector, and the surface normal vector. It returns the reflected vector. In order for the math to work correctly, both input vectors should be normalized in advance.

```
vertex ProjectedVertex vertex_reflect(device Vertex *vertices     [[buffer(0)]],
                                      constant Uniforms &uniforms [[buffer(1)]],
                                      uint vid                    [[vertex_id]])
{
    float4 modelPosition = vertices[vid].position;
    float4 modelNormal = vertices[vid].normal;
    float4 worldCameraPosition = uniforms.worldCameraPosition;
    float4 worldPosition = uniforms.modelMatrix * modelPosition;
    float4 worldNormal = normalize(uniforms.normalMatrix * modelNormal);
    float4 worldEyeDirection = normalize(worldPosition - worldCameraPosition);

    ProjectedVertex outVert;
    outVert.position = uniforms.modelViewProjectionMatrix * modelPosition;
    outVert.texCoords = reflect(worldEyeDirection, worldNormal);

    return outVert;
}
```

The output vertex texture coordinates are interpolated by the hardware and passed to the fragment shader we already saw. This creates the effect of per-pixel reflection, since every execution of the fragment shader uses a different set of texture coordinates to sample the cube map.

The Physics of Refraction

The second phenomenon we will model is *refraction*. Refraction occurs when light passes from one medium to another. One everyday example is how a straw appears to bend in a glass of water.

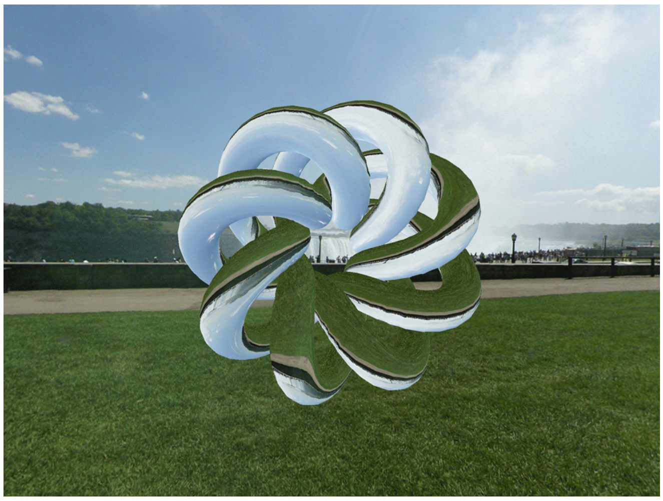


Figure 8.6: The environment map refracted through a torus knot

We won't discuss the mathematics of refraction in detail here, but the essential characteristic of refraction is that the amount the light bends is proportional to the ratio between the indices of refraction of the two media it is passing between. The *index of refraction* of a substance is a unitless quantity that characterizes how much it slows down the propagation of light, relative to a vacuum. Air has an index very near 1, while the index of water is approximately 1.333 and the index of glass is approximately 1.5.

Refraction follows a law called Snell's law, which states that the ratio of the sines of the incident and transmitted angles is equal to the inverse ratio of the indices of refraction of the media:

$\frac{\sin(\theta_I)}{\sin(\theta_T)} = \frac{\eta_1}{\eta_0}$

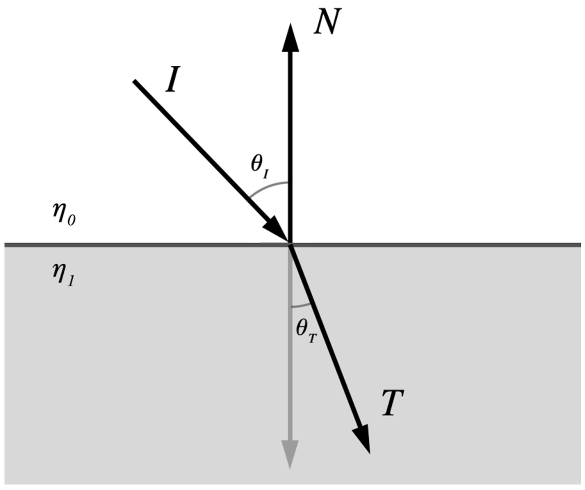


Figure 8.7: Light is bent when passing between substances with different indices of refraction

The Refraction Vertex Shader

The vertex shader for refraction is identical to that used for reflection, with the exception that we use a different build-in function, `refract`. We provide the incident vector, the surface normal, and the ratio between the two substances (air and glass, in the case of the sample code). The refracted vector is returned and used as the cube map texture coordinates as before.

```
vertex ProjectedVertex vertex_refract(device Vertex *vertices     [[buffer(0)]],
                                      constant Uniforms &uniforms [[buffer(1)]],
                                      uint vid                   [[vertex_id]])
{
    float4 modelPosition = vertices[vid].position;
    float4 modelNormal = vertices[vid].normal;

    float4 worldCameraPosition = uniforms.worldCameraPosition;
    float4 worldPosition = uniforms.modelMatrix * modelPosition;
    float4 worldNormal = normalize(uniforms.normalMatrix * modelNormal);
    float4 worldEyeDirection = normalize(worldPosition - worldCameraPosition);

    ProjectedVertex outVert;
    outVert.position = uniforms.modelViewProjectionMatrix * modelPosition;
    outVert.texCoords = refract(worldEyeDirection, worldNormal, kEtaRatio);

    return outVert;
}
```

Using Core Motion to Orient the Scene

The sample code for this chapter is in the 09-CubeMapping directory.

The scene in the sample app consists of a torus knot surrounded by a skybox. The torus knot reflects or refracts the surrounding scene; you can tap to alternate between the two.

The Motion Manager

The app uses the orientation of the device to rotate the scene so that the torus always appears at the center, with the environment revolving around it. This is achieved with the `CMMotionManager` class from the Core Motion framework.

To use a motion manager to track the device's movement, you should first check if device motion is available. If it is, you may proceed to set an update interval and request that the motion manager start tracking motion. The following code performs these steps, requesting an update sixty times per second.

```
CMMotionManager *motionManager = [[CMMotionManager alloc] init];
if (motionManager.deviceMotionAvailable)
{
    motionManager.deviceMotionUpdateInterval = 1 / 60.0;
    CMAttitudeReferenceFrame frame =
        CMAttitudeReferenceFrameXTrueNorthZVertical;
    [motionManager startDeviceMotionUpdatesUsingReferenceFrame:frame];
}
```

Reference Frames and Device Orientation

A *reference frame* is a set of axes with respect to which device orientation is specified. Several different reference frames are available. Each of them identifies Z as the vertical axis, while the alignment of the X axis can be chosen according to the application's need. We choose to align the X axis with magnetic north so that the orientation of the virtual scene aligns intuitively with the outside world and remains consistent between app runs.

The device orientation can be thought of as a rotation relative to the reference frame. The rotated basis has its axes aligned with the axes of the device, with +X pointing to the right of the device, +Y pointing to the top of the device, and +Z pointing out the front of the screen.

Reading the Current Orientation

Each frame, we want to get the updated device orientation, so we can align our virtual scene with the real world. We do this by asking for an instance of `CMDeviceMotion` from the motion manager.

```
CMDeviceMotion *motion = motionManager.deviceMotion;
```

The device motion object provides the device orientation in several equivalent forms: Euler angles, a rotation matrix, and a quaternion. Since we already use matrices to represent rotations in our Metal code, they are they best fit for incorporating Core Motion's data into our app.

Everyone has his or her own opinion about which view coordinate space is most intuitive, but we always choose a right-handed system with +Y pointing up, +X pointing right, and +Z pointing toward the viewer. This differs from Core Motion's conventions, so in order to use device orientation data, we interpret Core Motion's axes in the following way: Core Motion's X axis becomes our world's Z axis, Z becomes Y, and Y becomes X. We don't have to mirror any axes, because Core Motion's reference frames are all right-handed.

```
CMRotationMatrix m = motion.attitude.rotationMatrix;

vector_float4 X = { m.m12, m.m22, m.m32, 0 };
vector_float4 Y = { m.m13, m.m23, m.m33, 0 };
vector_float4 Z = { m.m11, m.m21, m.m31, 0 };
```

```
vector_float4 W = {     0,     0,     0, 1 };

matrix_float4x4 orientation = { X, Y, Z, W };
renderer.sceneOrientation = orientation;
```

The renderer incorporates the scene orientation matrix into the view matrices it constructs for the scene, thus keeping the scene in alignment with the physical world.

The Sample App

The sample code for this chapter is in the 08-CubeMapping directory.

Chapter 9

Compressed Textures

In this chapter, we will consider several GPU-friendly compressed texture formats. These formats allow us to trade some image quality for substantial improvements in disk usage and performance. In particular, we will look at the ETC2, PVRTC, and ASTC formats.

Why Use Compressed Textures?

In previous chapters, we created textures by loading an image into memory, drawing it into a bitmap context, and copying the bits into a Metal texture. These textures used uncompressed pixel formats such as `MTLPixelFormatRGBA8Unorm`.

Metal also supports several compressed formats. These formats allow us to copy image data directly into a texture without decompressing it. These formats are designed to be decompressed on-demand by the GPU, when the texture is sampled rather than when it is loaded.

The formats we will consider below are *lossy*, meaning they do not perfectly preserve the image contents, as a lossless format like PNG does. Instead, they trade a reduction in image quality for much smaller memory usage. Using compressed texture data therefore greatly reduces the memory bandwidth consumed when sampling textures and is more cache-friendly.

Figure 9.1: The sample app showcases a variety of compressed texture formats

A Brief Overview of Compressed Texture Formats

S3TC

S3TC, also known as DXT, was the first compressed format to gain broad adoption, due to its inclusion in DirectX 6.0 and OpenGL 1.3 in 1998 and 2001, respectively. Although S3TC is broadly supported on desktop GPUs, it is not available on iOS devices.

PVRTC

The PVRTC image format was introduced by Imagination Technologies, creators of the PowerVR series of GPUs at the heart of every iOS device. It was first described in a 2003 paper by Simon Fenney.

PVRTC operates by downsampling the source image into two smaller images, which are upscaled and blended to reconstruct an approximation of the original. It considers blocks of 4×4 or 4×8 pixels at a time, which are packed into one 64-bit quantity. Thus, each pixel occupies 4 bits or 2 bits, respectively.

One significant limitation of using PVRTC format on iOS is that textures must be square, and each dimension must be a power of two. Fortunately, game textures are most commonly produced in dimensions that are compatible with this limitation.

ETC

Ericsson Texture Compression (ETC) debuted in 2005. Similarly to PVRTC's 4-bit-per-pixel mode, it compresses each 4×4 block of pixels into a single 64-bit quantity, but lacks support for an alpha channel. A subsequent version, ETC2, adds support for 1-bit and 8-bit alpha channels. ETC2 is supported by all Metal-compatible hardware, but PVRTC often offers better quality at comparable file sizes.

ASTC

Advanced Scalable Texture Compression (ASTC) is the most recent compressed texture format supported by Metal. Developed by AMD and supported fairly widely on OpenGL ES 3.0-class hardware, this format incorporates a selectable block size (from 4×4 to 12×12) that determines the compression ratio of the image. This unprecedented flexibility makes ASTC a very attractive choice, but it requires an A8 processor at a minimum, making it unusable on devices such as the iPhone 5s.

Container Formats

The compression format used by a texture is only half of the picture. In order to load a texture from disk, we need to know its dimensions (width and height) and its pixel format. This metadata is often written into the file itself, in the form of a *header* that precedes the image data. The layout of the header is described by the *container format* of the image. In this chapter, we will use two container formats: PVR and ASTC.

The PVR Container Format

Not to be confused with the PVRTC compression format, the PVR container format describes how the data in a texture file should be interpreted. A PVR file can contain uncompressed data or compressed data in numerous formats (including S3TC, ETC, and PVRTC).

There are a few different, mutually-incompatible versions of the PVR format. This means that each version of PVR requires different code to parse correctly. For example, the header structure for the "legacy" format (PVRv2) looks like this:

```
struct PVRv2Header
{
    uint32_t headerLength;
    uint32_t height;
    uint32_t width;
    uint32_t mipmapCount;
    uint32_t flags;
    uint32_t dataLength;
    uint32_t bitsPerPixel;
    uint32_t redBitmask;
    uint32_t greenBitmask;
    uint32_t blueBitmask;
    uint32_t alphaBitmask;
    uint32_t pvrTag;
    uint32_t surfaceCount;
};
```

The header structure for the most recent version (PVRv3) looks like this:

```
struct PVRv3Header {
    uint32_t version;
    uint32_t flags;
    uint64_t pixelFormat;
    uint32_t colorSpace;
    uint32_t channelType;
    uint32_t height;
    uint32_t width;
    uint32_t depth;
    uint32_t surfaceCount;
    uint32_t faceCount;
    uint32_t mipmapCount;
    uint32_t metadataLength;
};
```

There are more similarities than differences between the headers. The various fields describe the contained data, including the texture's dimensions and whether the file contains a set of mipmaps or a single image. If the file contains mipmaps, they are simply concatenated together with no padding. The program reading the file is responsible for calculating the expected length of each image. The sample code for this chapter includes code that parses both legacy and PVRv3 containers.

Note that the header also has `surfaceCount` and (in the case of PVRv3) `faceCount` fields, which can be used when writing texture arrays or cube maps into a single file. We will not use these fields in this chapter.

The KTX Container Format

The KTX file format is a container devised by the Khronos Group, who are responsible for many open standards, including OpenGL. KTX is designed to make loading texture data into OpenGL as seamless as possible, but we can use it to load texture data into Metal quite easily.

The KTX file header looks like this:

```
struct KTXHeader
{
    uint8_t identifier[12];
    uint32_t endianness;
    uint32_t glType;
    uint32_t glTypeSize;
    uint32_t glFormat;
    uint32_t glInternalFormat;
    uint32_t glBaseInternalFormat;
    uint32_t width;
    uint32_t height;
    uint32_t depth;
    uint32_t arrayElementCount;
    uint32_t faceCount;
    uint32_t mipmapCount;
    uint32_t keyValueDataLength;
} MBEKTXHeader;
```

The `identifier` field is a sequence of bytes that identifies the file as a KTX container and helps detect encoding errors. The fields beginning with `gl` correspond to OpenGL enumerations. The one we care about is the `glInternalFormat` field, which tells us which encoder (PVRTC, ASTC, etc) was used to compress the texture data. Once we know this, we can map the GL internal format to a Metal pixel format and extract the data from the remainder of the file.

Refer to the KTX specification (Khronos Group 2013) for all of the details on this format.

Creating Compressed Textures

There are various tools for creating compressed textures, both on the command line and with a GUI. Below, we'll look at two tools, which allow us to create textures in the formats discussed above.

Creating Textures with texturetool

Apple includes a command-line utility with Xcode called `texturetool` that can convert images into compressed texture formats such as PVRTC and ASTC. At the time of this writing, it does not support the PVRv3 container format. Rather, it writes PVR files using the legacy (PVRv2) format, described above. With the `-e` flag, you specify your desired container format: PVR, KTX, or RAW (no header). With the `-f` format you specify the compression type: PVRTC or ASTC. The `-m` flag signifies that all mipmaps should be generated and written sequentially into the file.

Each compression format has its own flags for selecting the degree of compression. For example, to create a 4 bit-per-pixel PVRTC texture with a PVR legacy header, including all mipmap levels, use the following invocation of `texturetool`:

```
texturetool -m -e PVRTC -f PVR --bits-per-pixel-4 -o output.pvr input.png
```

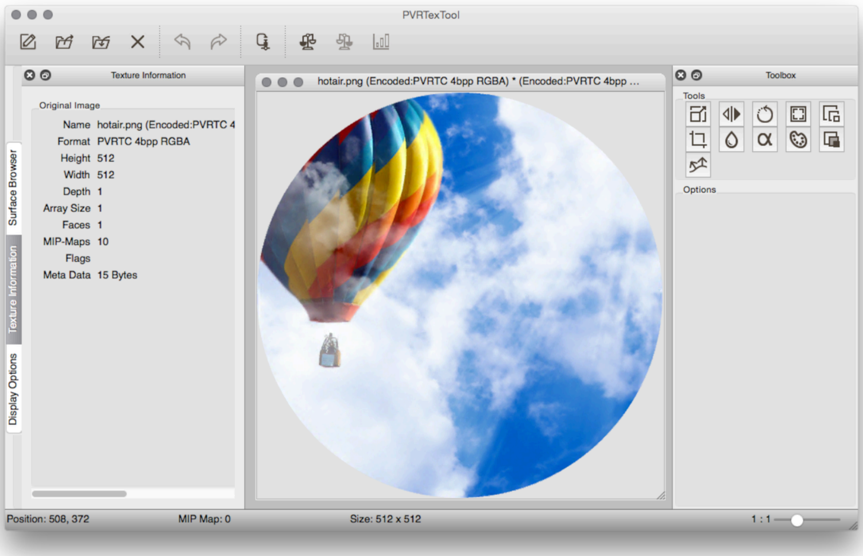
To create an ASTC-compressed texture in an KTX container with an 8×8 block size using the "thorough" compression mode:

```
texturetool -m -e ASTC -f KTX --block-width-8 --block-height-8 \
--compression-mode-thorough -o output.astc input.png
```

Creating Textures with PVRTexToolGUI

Imagination Technologies includes a tool in their PowerVR SDK that allows creation of compressed textures with a GUI. PVRTexToolGUI allows you to import an image and select a variety of compression formats, including ASTC, PVRTC, and ETC2. Unlike `texturetool`, this application uses the PVRv3 container format by default, so you should use code that expects the appropriate header format if you compress textures with this tool.

PVRTexToolGUI has a command-line counterpart, `PVRTexToolCLI`, that exposes all of the parameters available in the GUI and allows you to write scripts for rapidly batch-converting textures.



hotair.png (Encoded:PVRTC 4bpp RGBA) * (Encoded:PVRTC 4bpp ...

Original Image

Name	hotair.png (Encoded:PVRTC 4
Format	PVRTC 4bpp RGBA
Height	512
Width	512
Depth	1
Array Size	1
Faces	1
MIP-Maps	10
Flags	
Meta Data	15 Bytes

```
Position: 508, 372          MIP Map: 0          Size: 512 x 512          1 : 1
```

Figure 9.2: The PVRTexToolGUI from Imagination Technologies allows the user to compress images in a variety of formats

Loading Compressed Texture Data into Metal

Loading compressed textures is a two-fold process. First, we read the header of the container format. Then, we read the image data described by the header and hand it to Metal. Regardless of the compression algorithm used, we do not need to do any parsing of the image data ourselves, since Metal is able to understand and decode compressed textures in hardware.

Creating a texture with compressed data is exactly the same as with uncompressed data. First, we create a texture descriptor with the appropriate pixel format and dimensions. If the texture has mipmaps, we indicate so with the `mipmapped` parameter. For example, here is how we create a texture descriptor for a PVRTC texture with 4 bits per pixel for which we have mipmaps:

```
[MTLTextureDescriptor
    texture2DDescriptorWithPixelFormat:MTLPixelFormatPVRTC_RGBA_4BPP
   width:width
```

```
    height:height
    mipmapped:YES];
```

We can then ask a device to produce a texture that matches this descriptor:

```
id<MTLTexture> texture = [device newTextureWithDescriptor:descriptor];
```

For each mipmap level in the image, we need to make a separate call to the `MTLTexture` method `-replaceRegion:mipmapLevel:withBytes:bytesPerRow:`, as follows:

```
MTLRegion region = MTLRegionMake2D(0, 0, levelWidth, levelHeight);
[texture replaceRegion:region
           mipmapLevel:level
             withBytes:levelData
           bytesPerRow:levelBytesPerRow];
```

Where `levelWidth` and `levelHeight` are the dimensions of the current mip level, `level` is the index of the current mip level (the base level is index 0), and `levelData` is a pointer to the mip level's image data. `levelBytesPerRow` can be computed as the length of the image data divided by the level's height. When loading PVRTC data, `bytesPerRow` can be set to 0.

The Sample App

The sample code for this chapter is in the 09-CompressedTextures directory.

The sample app includes a demonstration of all of the texture formats mentioned in this chapter. Tapping the "Switch Textures" button brings up a menu that allows the user to select among the various textures. The menu options for the ASTC options will not be available if the device on which the app is running does not support ASTC compression.

Generally speaking, ASTC offers the best quality-to-size ratio on hardware where it is available. The PVRTC format should generally be preferred when targeting devices with the A7 processor.

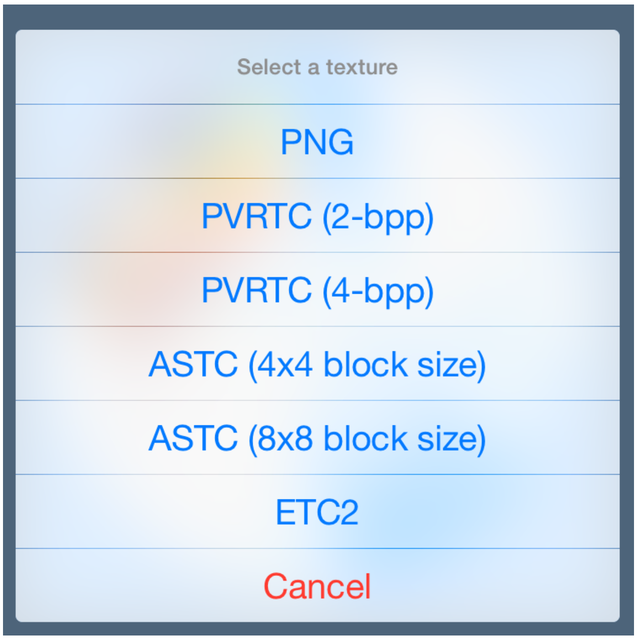


Figure 9.3: The sample app allows you to select among several compression formats supported by Metal.

Chapter 10

Translucency and Transparency

One of the topics we've handily avoided so far in our exploration of Metal is rendering of materials that are not opaque. In this chapter, we'll explore a couple of related techniques for achieving transparency and translucency: alpha testing and alpha blending.

The sample scene for this chapter is a desert containing many palm trees and a few pools of water. The leaves of the palm trees consist of a few polygons each, textured with a partially-transparent texture, and the water is rendered as a translucent surface via alpha blending, which we'll discuss in detail below.

What is Alpha, Anyway?

Alpha is the *opacity* (or *coverage*) of a color value. The higher the alpha value, the more opaque the color. An alpha value of 0 represents total transparency, while a value of 1 (or 100%) represents total opacity. When speaking in terms of fragment color, the alpha component indicates how much the scene behind the fragment shows through. See (Glassner 2015) for a rigorous discussion of the fundamentals.

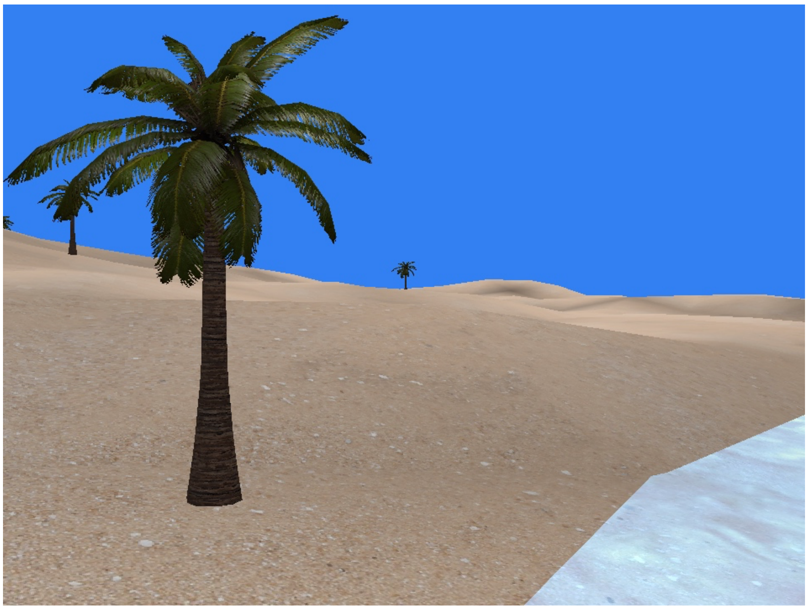


Figure 10.1: The sample app demonstrates alpha testing and alpha blending

Alpha Testing

The first technique we'll use to render partially-transparent surfaces is *alpha testing*.

Alpha testing allows us to determine whether a fragment should contribute to the color stored in the renderbuffer, by comparing its opacity against a threshold (called the *reference value*). Alpha testing is used to make surfaces *selectively* transparent.

One common application of alpha testing is the rendering of foliage, where relatively few polygons can be used to describe the shape of leaves, and the alpha channel of the leaf texture can be used to determine which pixels are drawn and which are not.

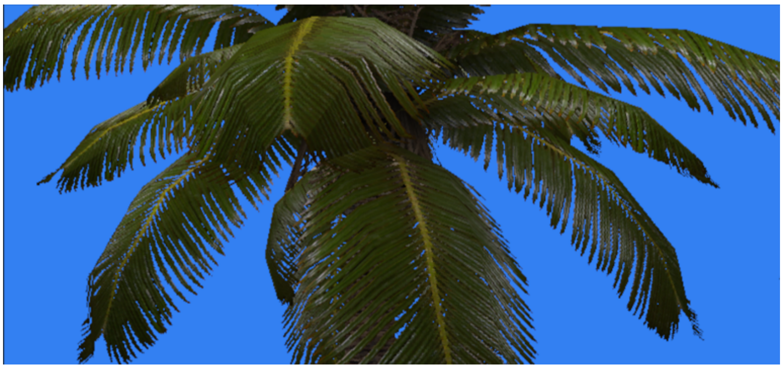


Figure 10.2: Alpha testing allows us to draw partially-transparent palm fronds

Alpha Testing in Metal

Alpha testing in Metal is implemented in the fragment function. The shader file contains a global reference value (0.5 in the sample app), against which our alpha values will be compared. The fragment function samples the diffuse texture of the palm tree, which has an alpha value of 0 for the texels that should be transparent. The sampled color is then used to decide whether the fragment should be visible.

The 'discard_fragment' function

In order to indicate to Metal that we don't want to provide a color for a fragment, we cannot simply set the returned alpha value to 0. If we did, the fragment depth would still be written into the depth buffer, causing any geometry behind the "transparent" point to be obscured.

Instead, we need to call a special function that avoids specifying a color value for the fragment entirely: `discard_fragment`. Calling this function prevents Metal from writing the computed depth and color values of the fragment into the framebuffer, which allows the scene behind the fragment to show through.

The Texture Alpha Channel

To perform the per-fragment alpha test, we need a specially-constructed texture that contains suitable coverage information in its alpha channel. The figure below shows the palm leaf texture used in the sample app.

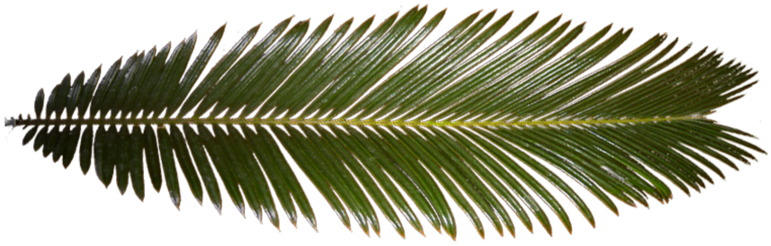


Figure 10.3: The texture used to draw the palm fronds in the sample app. Transparent portions of the image are not drawn.

Performing the Alpha Test

The implementation of alpha testing in the fragment shader is very straightforward. Bringing together the techniques we just discussed, we test the sampled alpha value against the threshold, and discard any fragments that fail the alpha test:

```
float4 textureColor = texture.sample(texSampler, vert.texCoords);

if (textureColor.a < kAlphaTestReferenceValue)
    discard_fragment();
```

A Note on Performance

Using `discard_fragment` has performance implications. In particular, it prevents the hardware from performing an optimization called *early depth test* (sometimes called *early-z*).

Oftentimes, the hardware can determine whether a fragment will contribute to the renderbuffer before calling the fragment function, since fragment depth is computed in the rasterizer. This means that it can avoid shading blocks of fragments that are known to be obscured by other geometry.

On the other hand, if a fragment is going to be shaded with a function that contains conditional `discard_fragment` calls, this optimization cannot be applied, and the hardware must invoke the shader for every potentially-visible fragment.

In the sample code for this chapter, we have separate fragment functions for geometry that is rendered with the alpha test and without. The alpha testing function should only be used on geometry that actually needs it, as overuse of `discard_fragment` can have a strongly negative performance impact.

A full explanation of early depth-testing is beyond the scope of this book, but Fabien Giesen's blog series (Giesen 2011) provides further details about both depth-testing and many other topics related to the modern 3D rendering pipeline.

Alpha Blending

Another useful technique for implementing transparency is *alpha blending*.

Alpha blending is achieved by interpolating between the color already in the renderbuffer (the destination) and the fragment currently being shaded (the source). The exact formula used depends on the desired effect, but for our current purposes we will use the following equation:

$c_f = \alpha_s \times c_s + (1 - \alpha_s) \times c_d$

where c_s and c_d are the RGB components of the source and destination colors, respectively; α_s is the source color's opacity; and c_f is the final blended color value for the fragment.

Expressed in words, this formula says: we multiply the source color's opacity by the source color's RGB components, producing the contribution of the fragment to the final color. This is added to the additive inverse of the opacity multiplied by the color already in the renderbuffer. This "blends" the two colors to create the new color that is written to the renderbuffer.

Alpha Blending in Metal

In the sample project for this chapter, we use alpha blending to make the surface of the water translucent. This effect is computationally cheap and not particularly convincing, but it could be supplemented with cube map reflection or an animation that slides the texture over the water surface or physically displaces it.

Enabling Blending in the Pipeline State

There are two basic approaches to blending in Metal: fixed-function and programmable. We will discuss fixed-function blending in this section. Fixed-function blending consists of setting properties on the color attachment descriptor of the render pipeline descriptor. These properties determine how the color returned by the fragment function is combined with the pixel's existing color to produce the final color.

To make the next several steps more concise, we save a reference to the color attachment:

```
MTLRenderPipelineColorAttachmentDescriptor *renderbufferAttachment =
    pipelineDescriptor.colorAttachments[0];
```

To enable blending, set `blendingEnabled` to `YES`:

```
renderbufferAttachment.blendingEnabled = YES;
```

Next, we select the operation that is used to combine the weighted color and alpha components. Here, we choose `Add`:

```
renderbufferAttachment.rgbBlendOperation = MTLBlendOperationAdd;
renderbufferAttachment.alphaBlendOperation = MTLBlendOperationAdd;
```

Now we need to specify the weighting factors for the source color and alpha. We select the `SourceAlpha` factor, to match with the formula given above.

```
renderbufferAttachment.sourceRGBBlendFactor = MTLBlendFactorSourceAlpha;
renderbufferAttachment.sourceAlphaBlendFactor = MTLBlendFactorSourceAlpha;
```

Finally, we choose the weighting factors for the destination color and alpha. These are the additive inverse of the factors for the source color, `OneMinusSourceAlpha`:

```
renderbufferAttachment.destinationRGBBlendFactor =
    MTLBlendFactorOneMinusSourceAlpha;
renderbufferAttachment.destinationAlphaBlendFactor =
    MTLBlendFactorOneMinusSourceAlpha;
```

The Alpha Blending Fragment Shader

The fragment shader returns the color it computes for its fragment. In previous chapters, the alpha component of this color was insignificant, because blending was not enabled. Now, we need to ensure that the alpha component returned by the shader represents the desired opacity of the fragment. This value will be used as the "source" alpha value in the blending equation configured on the rendering pipeline state.

In the sample code, we multiply the sampled diffuse texture color with the current vertex color to get a source color for the fragment. Since the texture is opaque, the vertex color's alpha component becomes the opacity of the water. Passing this value as the alpha of the fragment color's return value produces alpha blending according to how the color attachment was previously configured.

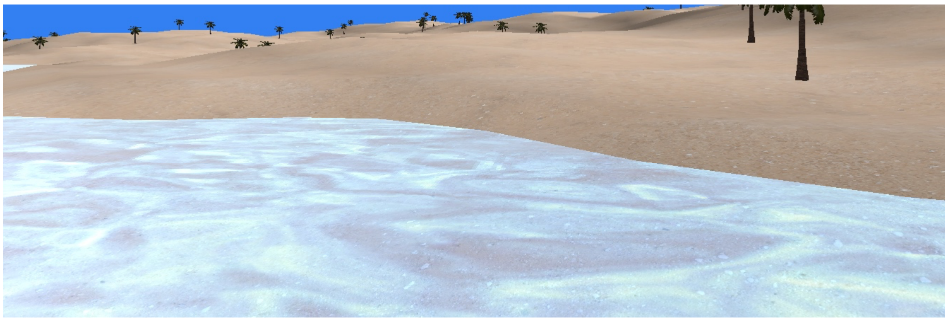


Figure 10.4: Alpha blending allows the sandy desert show through the clear water surface

Order-Dependent Blending versus Order-Independent Blending

When rendering translucent surfaces, order matters. In order for alpha blending to produce a correct image, translucent objects should be rendered after all opaque objects. Additionally, they should be drawn back-to-front, relative to the camera. Since order is dependent on the view position and orientation, objects need to be re-sorted every time their relative order changes, often whenever the camera moves.

We avoid this problem entirely in the sample app by having only a single, convex, translucent surface in the scene: the water. Other applications won't have it so easy.

There has been much research in the last several years on ways to avoid this expensive sorting step. One technique uses a so-called *A-buffer* (introduced by Carpenter in 1984 (Carpenter 1984)) to maintain a pair of color and depth values for the fragments that contribute to each pixel. These fragments are then sorted and blended in a second pass.

More recently, research has placed an emphasis on the mathematics of alpha compositing itself, in an attempt to derive an *order-independent transparency* solution. Refer to McGuire and Bavoil's paper (McGuire and Bavoil 2013) for advances in this technique.

The Sample App

The sample code for this chapter is in the 10-AlphaBlending directory. Pressing on the screen moves the camera forward along its current heading, while panning left or right changes the direction in which the camera heads.

Chapter 11

Instanced Rendering

In this chapter, we will discuss an important technique for efficiently drawing many objects with a single draw call: *instanced rendering*. This technique helps you get the most out of the GPU while keeping memory and CPU usage to a minimum.

The sample app for this chapter renders several dozen animated cows moving on top of a randomly-generated terrain patch. Each cow has its own position, orientation, and movement direction, all of which are updated every frame. We do all of this drawing with only *two* draw calls. The app consumes only a few percent of the CPU on an A8 processor, but maxes out the GPU, drawing over 240,000 triangles per frame. Even with this large load, such a device manages to render at an ideal 60 frames per second.

What is Instanced Rendering?

Virtual worlds frequently have many copies of certain elements in the scene: particles, foliage, enemies, and so on. These elements are represented in memory by a single piece of geometry (a mesh) and a set of attributes that are specific to the application. Instanced rendering draws the same geometry multiple times, with each instance's attributes used to control where and how it appears.

Instanced rendering is also called "geometry instancing", "instanced drawing", or sometimes just "instancing".

Setting the Scene

The virtual scene for this chapter is a pastoral hillside with numerous roving bovines. The terrain is uniquely generated each time the app runs, and every cow has its own randomized path of motion.

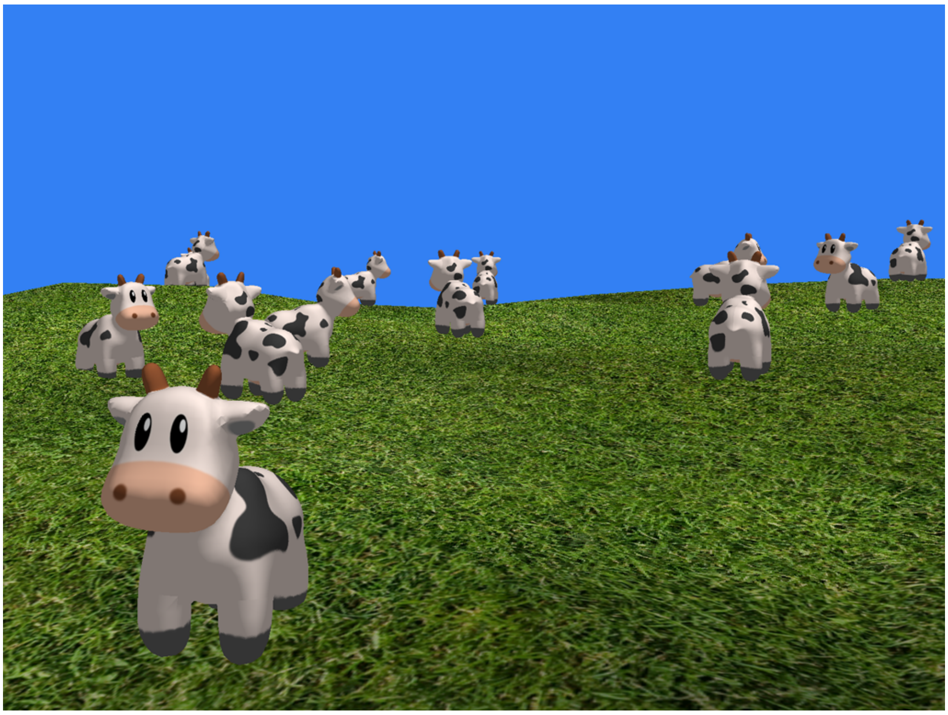


Figure 11.1: Dozens of animated characters can be drawn efficiently with instanced rendering. (Grass texture provided by Simon Murray of goodtextures.com)

Generating the Terrain

The features of the terrain are created by an algorithm called *midpoint displacement*, also called the "diamond-square" algorithm. It is a technique that recursively subdivides the edges of a patch of terrain and nudges them randomly up or down to produce natural-looking hills. Since the focus of this chapter is instanced drawing and not terrain generation, refer to the sample source code if you are curious about this technique (see the

`MBETerrainMesh` class). An interactive demonstration of the technique can be found online (Boxley 2011). The original paper on the technique is (Miller 1986).

Loading the Model

We use the OBJ model loader from previous chapters to load the cow model. Once we have an OBJ model in memory, we create an instance of `MBEOBJMesh` from the appropriate group from the OBJ file.

Instanced Rendering

Instanced rendering is performed by issuing a draw call that specifies how many times the geometry should be rendered. In order for each instance to have its own attributes, we set a buffer containing per-instance data as one of the buffer arguments in the vertex shader argument table. We also need to pass in a shared uniform buffer, which stores the uniforms that are shared across all instances. Here is the complete argument table configured for rendering the cow mesh:

```
[commandEncoder setVertexBuffer:cowMesh.vertexBuffer offset:0 atIndex:0];
[commandEncoder setVertexBuffer:sharedUniformBuffer offset:0 atIndex:1];
[commandEncoder setVertexBuffer:cowUniformBuffer offset:0 atIndex:2];
```

Now let's look specifically at how we lay out the uniforms in memory.

Storing Per-Instance Uniforms

For each instance, we need a unique model matrix, and a corresponding normal matrix. Recall that the normal matrix is used to transform the normals of the mesh into world space. We also want to store the projection matrix by itself in a shared uniform buffer. We split the `Uniforms` struct into two structs:

```
typedef struct
{
    matrix_float4x4 viewProjectionMatrix;
} Uniforms;

typedef struct
{
    matrix_float4x4 modelMatrix;
    matrix_float3x3 normalMatrix;
} PerInstanceUniforms;
```

The shared uniforms are stored in a Metal buffer that can accommodate a single instance of type `Uniforms`. The per-instance uniforms buffer has room for one instance of `PerInstanceUniforms` per cow we want to render:

```
cowUniformBuffer = [device newBufferWithLength:sizeof(PerInstanceUniforms) *
     MBECowCount
    options:MTLResourceOptionCPUCacheModeDefault];
```

Updating Per-Instance Uniforms

Because we want the cows to move, we store a few simple attributes in a data model called `MBECow`. Each frame, we update these values to move the cow to its new position and rotate it so it is aligned with its direction of travel.

Once a cow object is up-to-date, we can generate the appropriate matrices for each cow and write them into the per-instance buffer, for use with the next draw call:

```
PerInstanceUniforms uniforms;
uniforms.modelMatrix = matrix_multiply(translation, rotation);
uniforms.normalMatrix = matrix_upper_left3x3(uniforms.modelMatrix);
NSInteger instanceUniformOffset = sizeof(PerInstanceUniforms) * instanceIndex;
memcpy([self.cowUniformBuffer contents] + instanceUniformOffset, &uniforms,
     sizeof(PerInstanceUniforms));
```

Issuing the Draw Call

To issue the instanced drawing call, we use the `-drawIndexedPrimitive:` method on the render command encoder that has an `instanceCount:` parameter. Here, we pass the total number of instances:

```
[commandEncoder drawIndexedPrimitives:MTLPrimitiveTypeTriangle
                           indexCount:indexCount
                            indexType:MTLIndexTypeUInt16
                          indexBuffer:cowMesh.indexBuffer
                    indexBufferOffset:0
                        instanceCount:MBECowCount];
```

To execute this draw call, the GPU will draw the mesh many times, reusing the geometry each time. However, we need a way of getting the appropriate set of matrices inside the vertex shader so we can transform each cow to its position in the world. To do that, lets look at how to get the instance ID from inside the vertex shader.

Accessing Per-Instance Data in Shaders

To index into the per-instance uniform buffers, we add a vertex shader parameter with the `instance_id` attribute. This tells Metal that we want it to pass us a parameter that represents the index of the instance that is currently being drawn. We can then access the per-instance uniforms array at the correct offset and extract the appropriate matrices:

```
vertex ProjectedVertex vertex_project(device InVertex *vertices [[buffer(0)]],
    constant Uniforms &uniforms [[buffer(1)]],
    constant PerInstanceUniforms *perInstanceUniforms [[buffer(2)]],
    ushort vid [[vertex_id]],
    ushort iid [[instance_id]])
{
    float4x4 instanceModelMatrix = perInstanceUniforms[iid].modelMatrix;
    float3x3 instanceNormalMatrix = perInstanceUniforms[iid].normalMatrix;
```

The rest of the vertex shader is straightforward. It projects the vertex, transforms the normal, and passes through the texture coordinates.

Going Further

You can store any kind of data you want in the per-instance uniform structure. For example, you could pass in a color for each instance, and use it to tint each object uniquely. You could include a texture index, and index into a texture array to give a completely different visual appearance to certain instances. You can also multiply a scaling matrix into the model transformation to give each instance a different physical size. Essentially any characteristic (except the mesh topology itself) can be varied to create a unique appearance for each instance.

The Sample App

The sample code for this chapter is in the 11-InstancedDrawing directory.

You can move around in the sample app by holding your finger on the screen to move forward. Turn the camera by panning from left or right.

Chapter 12

Rendering Text

In this chapter, we will discuss a method for rendering high-fidelity text with Metal. It's easy to overlook text rendering when thinking about 3D graphics. However, very few games or applications can get by without displaying any text, so it's important to consider how we can best use the GPU to incorporate text into our Metal apps.

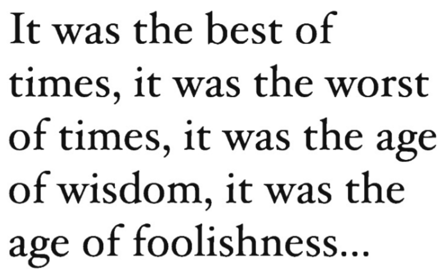

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness...

Figure 12.1: Text rendered by the sample app, using the signed-distance field technique

The Structure and Interpretation of Fonts

We begin the process of drawing text by selecting a *font*. Fonts are collections of glyphs, which are the graphical representation of characters or portions of characters. In modern font formats, glyphs are represented as piecewise curves, specifically line segments and quadratic Bezier curves.

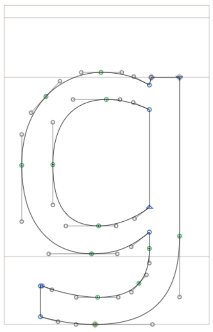


Figure 12.2: The 'g' glyph from Verdana, showing how glyphs are constructed of piecewise quadratic Beziér curves

Drawing a string of text consists of at least two distinct phases. First, the *text layout engine* determines which glyphs will be used to represent the string and how they'll be positioned relative to one another. Then, the *rendering engine* is responsible for turning the abstract description of the glyphs into text on the screen.

Approaches to Real-Time Text Rendering

There are numerous ways you might go about rendering text on iOS. You are probably familiar with UIKit controls such as `UILabel` and `UITextField`. These UI elements are backed by a powerful framework called Core Text. Core Text is a Unicode text layout engine that integrates tightly with Quartz 2D (Core Graphics) to lay out and render text.

Text layout is an enormously complicated subject that must account for different scripts, writing directions, and typographic conventions. We would never want to reinvent this functionality ourselves, so we'll let Core Text do a lot of the heavy lifting as we settle in on our solution for real-time text rendering.

Let's take a brief tour through common ways of drawing text in the context of real-time 3D graphics.

Dynamic Rasterization

One of the most flexible approaches to text rendering is dynamic rasterization, in which strings are rasterized on the CPU, and the resulting bitmap is uploaded as a texture to the GPU for drawing. This is the approach taken by libraries such as stb_truetype (Barrett 2014).

The disadvantage of dynamic rasterization is the computational cost of redrawing the glyphs whenever the text string changes. Even though much of the cost of text rendering occurs in the layout phase, rasterizing glyphs is nontrivial in its demand on the CPU, and there is no extant GPU implementation of font rasterization on iOS. This technique also requires an amount of texture memory proportional to the font size and the length of the string being rendered. Finally, when magnified, rasterized text tends to become blurry or blocky, depending on the magnification filter.

Font Atlases

Many applications that use the GPU to draw text prefer to render all possible glyphs up-front rather than drawing them dynamically. This approach trades off texture memory for the computational cost of rasterizing glyphs on-demand. In order to minimize the amount of texture memory required by a font, the glyphs are packed into a single rectangular texture, called an *atlas*. The figure below illustrates such a texture.

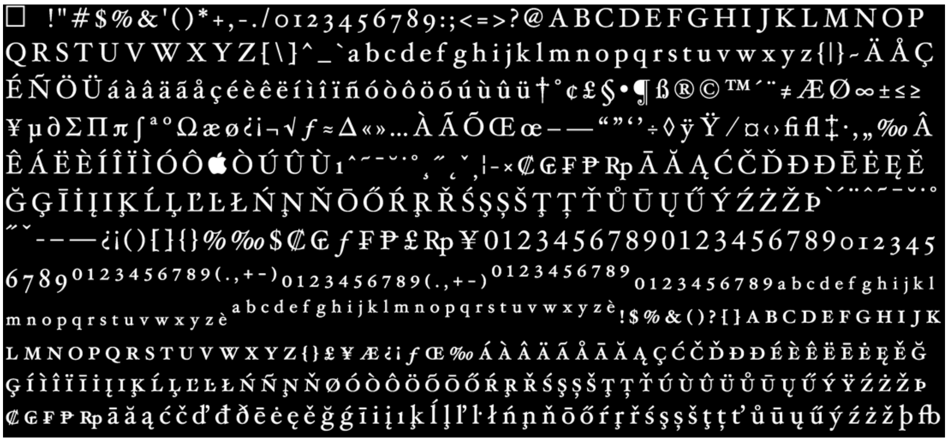


Figure 12.3: A font atlas

An ancillary data structure stores the texture coordinates describing the bounding rectangle of each glyph. When drawing a string, the application generates a mesh with the appropriate positions and texture coordinates of the string's constituent glyphs.

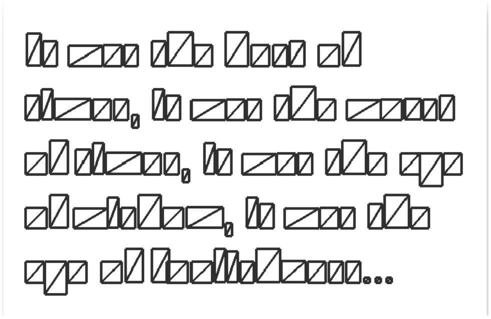


Figure 12.4: A pair of triangles are generated for each glyph in the text string

One disadvantage of font atlases is that they consume a sizable amount of memory even when many of the glyphs they contain are not used at runtime.

Like dynamic rasterization, text rendered with an atlas texture suffers from a loss of fidelity upon magnification. Here, the problem may be even worse, since the glyphs are often drawn smaller in order to pack an entire font into one texture.

In the next section, we will seek to rectify some of the issues with naïve atlas-based text rendering.

Signed-Distance Fields

The approach we will explore in depth uses a *signed-distance field*, which is a precomputed representation of a font atlas that stores the glyph outlines implicitly. Specifically, the texel values of a signed-distance field texture correspond to the distance of the texel to the nearest glyph edge, where texels outside the glyph take on negative values.

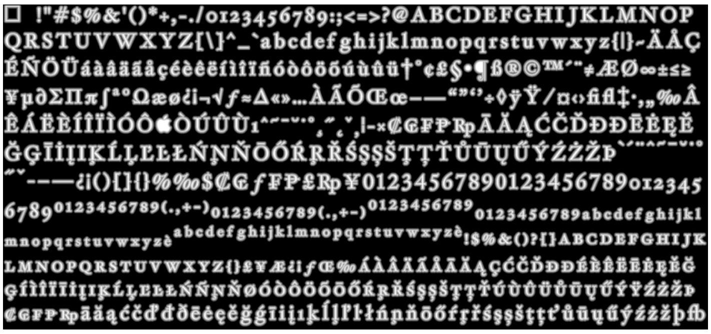


Figure 12.5: The signed-distance field generated from a font atlas. The brightest pixels are the farthest inside.

In order to store a signed-distance field in a texture, it must be scaled and quantized to match the pixel format. Our sample project uses a single-channel 8-bit texture, consuming one byte per pixel. By this construction, texels that fall exactly on the edge of a glyph have a value of 127, or 50%.

In its simplest incarnation, signed-distance field rendering can be done with fixed-function alpha testing. By discarding all fragments whose value is less than 50%, only those pixels inside the glyph will be rendered. Unfortunately, this has the effect of producing "wiggles" along the edges of the glyph, since the downsampled and quantized distance texture does not capture adequate detail to perfectly recreate the ideal outline.

An improvement to the alpha test technique is to use a pixel shader to interpolate between the inside and outside of the glyph, thus smoothing out the wiggly discontinuity. This is described in detail below.

Signed-distance field rendering was brought into the mainstream by Chris Green of Valve (Green 2007) describing the use of the technology in the hit game Team Fortress 2. Our implementation will closely follow the scheme laid out in Green's paper.

Signed-Distance Field Rendering in Metal

In this section we will describe in detail a method for achieving smooth text rendering on the GPU with Metal.

Generating a Font Atlas Texture

The first step is to render all of the available glyphs in our chosen font into an atlas. In the sample code, an optimal packing is not used; rather, the glyphs are simply laid out left to right, wrapping lines from top to bottom in a greedy fashion. This simplifies the implementation greatly, at the cost of some wasted space.

The sample code constructs a font atlas from a `UIFont` by determining the maximum size of the selected font whose glyphs will fit entirely within the bitmap used to construct the atlas (4096×4096 pixels). It then uses Core Text to retrieve the glyph outlines from the font and render them, without antialiasing, into the atlas image.

As the font atlas is being drawn, the implementation also stores the origin and extent (i.e., the texture coordinates) of each glyph in a separate array. This array is used during rendering to map from the laid-out glyphs to their respective area on the atlas texture.

Generating a Signed-Distance Field

The above procedure produces a binary image of the font at a fairly high resolution. That is, pixels that fall inside of a glyph are all the way "on" (255), and pixels that are outside a glyph are all the way "off" (0). We now need to perform a signed-distance transform on this bitmap to produce a signed-distance field representation of the font atlas, which we will use for rendering.

A Brute Force Approach

Generating a signed-distance field entails finding the distance from each texel to the closest glyph edge. Some implementations, such as GLyphy (Esfahbod 2014), perform this calculation directly against the piecewise curve representation of the glyphs. This approach can have spectacular fidelity, but the implementation is complicated and fraught with edge cases.

As we have chosen to generate a bitmap representation of the font, we could simply iterate over the neighborhood of each texel, performing a minimizing search as we encounter texels on the other side of the edge. To even approach tractability, this requires that we choose a reasonably-sized area in which to conduct our search.

A reasonable heuristic is half the average stroke width of the font. For example, in a font for which a typical stroke is 20 pixels wide, texels that are so far inside the glyph that they have a distance of greater than 10 are already going to be clamped to the maximum value of "insideness" during rendering. Similarly, texels at a distance of more than 10 outside the glyph are unlikely candidates for influencing the way the glyph is rendered. Therefore, we would conduct a search on the 10 x 10 neighborhood around each texel.

According to Green, the brute-force approach was suitably fast for creating distance fields for text and vector artwork on the class of workstations used in the production of TF2. However, since there has been so much research into signed-distance field generation, let's take a look at a slightly better approach that will allow us to generate them fairly quickly, even on mobile hardware.

A Better Approach: Dead Reckoning

Signed-distance fields have broad applicability, and have therefore been the subject of much research. G. J. Grevera, building on a venerable algorithm known as the chamfer distance algorithm, constructed a more precise heuristic known as "dead reckoning" (Grevera 2004). In essence, the algorithm performs two passes over the source image, first propagating the minimal distance down and to the right, then propagating the minimal distances found in the first pass back up and to the left. At each step, the distance value is determined as the minimum distance value over some mask surrounding the center texel, plus the distance along the vector to the nearest previously-discovered edge.

Without going into all the details of this algorithm, it is worth noting that it is drastically faster than the brute-force approach. Across both passes, dead reckoning consults a neighborhood of just 3x3 texels, far fewer than a brute-force algorithm of similar accuracy. Although we have not implemented it with a compute shader, we strongly suspect it could be made faster still with a GPU-based implementation.

Using Core Text for Layout

Once we have the signed-distance field representation of a font, we need a way to render its glyphs on the screen. The first part of this process is, again, using the Core Text layout engine to tell us which glyphs should be rendered, and how they should be positioned. We use a `CTFramesetter` object to lay out the text in a chosen rectangle. The framesetting process produces an array of `CTLine` objects, each containing series of glyphs. You should consult the Core Text reference for additional details on these classes.

To construct a text mesh for rendering with Metal, we enumerate the glyphs provided by the Core Text framesetter, which gives us their screen-space coordinates and an index into the table of texture coordinates we constructed earlier when building the font atlas texture. These two pieces of data allow us to create an indexed triangle mesh representing the text string. This mesh can then be rendered in the usual fashion, with a fragment shader doing the heavy lifting of transforming from the signed-distance field texture to an appropriate color for each pixel.

The Orthographic Projection

When drawing text onto the screen, we use an *orthographic*, or *parallel* projection. This kind of projection flattens the mesh to the plane of the screen without introducing the foreshortening inherent in a perspective projection.

When rendering UI elements, it is convenient to select an orthographic projection whose extents match the dimensions of the view. Therefore, the orthographic projection used by the sample application uses (0, 0) as the upper-left corner of the screen and the drawable width and height of the screen, in pixels, as the bottom-right corner, matching UIKit's convention.

Mathematically, this transformation is represented by the following matrix:

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{l-r} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{b-t} \\ 0 & 0 & \frac{1}{f-n} & \frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here l, r, t, b, n, and f are the left, right, top, bottom, near and far clipping plane values. In our implementation the near and far planes are assumed to sit at $z = 0$ and $z = 1$, respectively.

The Vertex and Fragment Functions

The vertex function for drawing text is very straightforward; it looks exactly like vertex functions we've used in the past. Each vertex of the text mesh is transformed by a model matrix (which can be used to position and scale the text) and a combined view-projection matrix, which is simply the orthographic projection matrix discussed above.

The fragment function, on the other hand, is a little more involved. Since we are using the signed-distance field as more of a look-up table than a texture, we need to transform the sampled texel from the field into a color value that varies based on the proximity of the pixel to the corresponding glyph's edge.

We apply antialiasing at the edges of the glyphs by interpolating from opaque to translucent in a narrow band around the edge. The width of this band is computed per-pixel by finding the length of the gradient of the distance field using the built-in `dfdx` and `dfdy` functions. We then use the `smoothstep` function, which transitions from 0 to 1 across the width of this smoothed edge, using the sampled distance value itself as the final parameter. This produces an edge band that is roughly one pixel wide, regardless of how much the text is scaled up or down. This refinement on Green’s original approach is due to Gustavson (Gustavson 2012).

Here is the complete fragment function for rendering a glyph from a signed-distance field representation:

```
float edgeDistance = 0.5;
float dist = texture.sample(samp, vert.texCoords).r;
float edgeWidth = 0.7 * length(float2(dfdx(dist), dfdy(dist)));
float opacity = smoothstep(edgeDistance - edgeWidth,
    edgeDistance + edgeWidth, dist);
return half4(textColor.r, textColor.g, textColor.b, opacity);
```

Note that we return a color that has an alpha component, so the pipeline state we use should have alpha blending enabled in order for the text to properly blend with the geometry behind it. This also implies that the text should be drawn after the rest of the scene geometry.

The Sample App

The sample code for this chapter is in the 12-TextRendering directory.

The sample app for this chapter renders a paragraph of text that can be zoomed and panned in real-time. Interactivity is achieved with a pair of `UIGestureRecognizer`s. Notice how the edges of the glyphs remain quite sharp even under extreme magnification, in contrast to the way a pre-rasterized bitmap texture would become jagged or blurry under magnification.

Chapter 13

Introduction to Data-Parallel Programming

This chapter is an introduction to topics in *data-parallel* programming, also called *compute* programming when done on the GPU (in contrast to graphics programming, which is what we normally do with GPUs). We will cover the basics of setting up the compute pipeline and executing kernel functions on large sets of data in parallel. This chapter serves as set-up for the following chapter, which demonstrates how to do image processing on the GPU with compute kernels.

Kernel Functions

Throughout this book, we have used vertex and fragment functions extensively. In this chapter, we introduce a new kind of shader function: the `kernel` function. Kernel functions allow us to build massively-parallel programs that operate on many pieces of data at once. We will use the terms "kernel function," "compute kernel," and "compute shader" interchangeably.

A kernel function is identified in shader source by prefixing it with the `kernel` qualifier, just as we prefix other types of functions with `vertex` or `fragment`. One difference between these types of functions is that kernel functions **must** return void. This is because kernel functions operate on buffers and textures rather than feeding data to additional pipeline stages, as vertex and fragment functions do.

The following snippet is an example of a kernel function signature. The new attributes introduced in this function signature will be discussed in detail later on.

```
kernel void kernel_function(
    texture2d<float, access::read> inTexture [[texture(0)]],
    texture2d<float, access::write> outTexture [[texture(1)]],
    uint2 gid [[thread_position_in_grid]]);
```

The Compute Pipeline

Building the compute pipeline is similar to building the render pipeline for our work with 3D rendering. Instead of a renderer class, we create a *context* object to hold onto various Metal objects.

The Context Class

The context wraps up the device, library, and command queue, since these are long-lived objects that will be referenced by the various kernel classes we create.

The context class has a very simple interface. Calling the `+newContext` factory method returns a context with the system default device.

```
@interface MBEContext : NSObject

@property (strong) id<MTLDevice> device;
@property (strong) id<MTLLibrary> library;
@property (strong) id<MTLCommandQueue> commandQueue;

+ (instancetype)newContext;

@end
```

There is one serial command queue per context. This allows us to serialize work items, such as image filters that may have data dependencies between them.

Creating a Pipeline State

Building a pipeline state for executing kernel functions is a little bit simpler than creating a rendering pipeline. There is no equivalent to the `MTLRenderPipelineDescriptor` for compute pipelines, because the only configurable part of the compute pipeline is its associated kernel function.

As with vertex and fragment functions, kernel functions are retrieved by name from a library:

```
[library newFunctionWithName:@"kernel_function"];
```

The compute pipeline is then created by asking for a compute pipeline state object from the device. If an error occurs when compiling the kernel function for the target hardware, the error will be assigned to the `error` parameter, and `nil` will be returned.

```
id<MTLComputePipelineState> pipeline = [device
    newComputePipelineStateWithFunction:kernelFunction error:&error];
```

Creating a Command Buffer and Command Encoder

Just as we use a render command encoder to encode draw calls into a command buffer, we use a new type of command encoder to execute kernel functions: the `MTLComputeCommandEncoder`.

```
id<MTLCommandBuffer> commandBuffer = [context.commandQueue commandBuffer];
id<MTLComputeCommandEncoder> commandEncoder =
    [commandBuffer computeCommandEncoder];
```

Before sending work to the GPU, though, we need to understand how to configure the compute command encoder's argument table and how to describe the work we want the GPU to do.

The Argument Table

We've used the argument table previously to specify which buffers, textures, and sampler states are bound to the parameters of our vertex and fragment functions.

For example, we configured the arguments of our fragment shaders by calling the `-setFragmentTexture:atIndex:` method on a render command encoder. Recall that the index parameter matches up the entry in the argument table to the parameter with the corresponding attribute (e.g., `[[texture(0)]]`) in the signature of the fragment function.

There is a similar method for setting up the argument table for compute encoders: `-setTexture:atIndex:`. We will use this method to set up the argument table when preparing to execute kernel functions.

Threadgroups

Threadgroups are a central concept in Metal kernel function programming. In order to execute in parallel, each workload must be broken apart into chunks, called threadgroups, that can be further divided and assigned to a pool of threads on the GPU.

In order to operate efficiently, the GPU does not schedule individual threads. Instead, they are scheduled in sets (sometimes called "warps" or "wavefronts", though the Metal documentation does not use these terms). The *thread execution width* represents the size of this unit of execution. This is the number of threads that are actually scheduled to run concurrently on the GPU. You can query this value from a command encoder using its `threadExecutionWidth` property. It is likely to be a small power of two, such as 32 or 64.

We can also determine the upper bound of our threadgroup size by quering the `maxTotalThreadsPerThreadgroup`. This number will always be a multiple of the thread execution width. For example, on iPhone 6 it is 512.

To make the most efficient use of the GPU, the total number of items in a threadgroup should be a multiple of the thread execution width, and must be lower than the maximum total threads per threadgroup. This informs how we choose to subdivide the input data for fastest and most convenient execution.

Threadgroups do not have to be one-dimensional. It is often convenient for the dimension of the threadgroup to match the dimension of the data being operated on. For example, when operating on a 2D image, each chunk will typically be a rectangular region of the source texture. The figure below shows how you might choose to subdivide a texture into threadgroups.

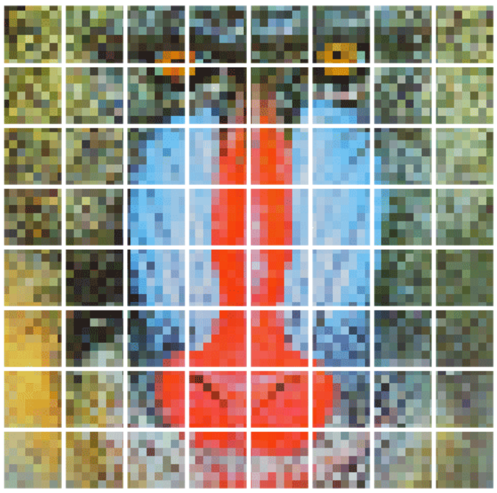


Figure 13.1: An image to be processed is divided up into a number of threadgroups, represented here by white squares.

In order to tell Metal the dimensions of each threadgroup and how many threadgroups should be executed in a given compute call, we create a pair of `MTLSize` structs:

```
MTLSize threadgroupCounts = MTLSizeMake(8, 8, 1);
MTLSize threadgroups = MTLSizeMake([texture width] / threadgroupCounts.width,
                                   [texture height] / threadgroupCounts.height,
                                   1);
```

Here, we somewhat arbitrarily choose a threadgroup size of eight rows by eight columns, or a total of 64 items per threadgroup. We make the assumption that textures being processed by this code will have dimensions that are multiples of 8, which is often a safe bet. The total threadgroup size is an even multiple of the target hardware's thread execution width, and is safely below the maximum total thread count. If the texture dimensions were not evenly divisible by our threadgroup size, we would need to take measures in our kernel function not to read or write outside the bounds of our textures.

Now that we have determined the size of our threadgroups and how many we need to execute, we are ready to put the GPU to work.

Dispatching Threadgroups for Execution

Encoding a command to execute a kernel function on a set of data is called *dispatching*. Once we have a reference to a compute command encoder, we can call its `-dispatchThreadgroups:threadsPerThreadgroup:` method to encode the request for execution, passing the `MTLSize` structs we computed previously.

```
[commandEncoder dispatchThreadgroups:threadgroups
    threadsPerThreadgroup:threadgroupCounts];
```

Once we are done dispatching, we tell the command encoder to `endEncoding`, then `commit` the corresponding command buffer. We can then use the `-waitUntilCompleted` method on the command buffer to block until the shader is done running on the GPU. The kernel function will be executed once per data item (e.g., once per texel in the source texture).

Conclusion

In this brief chapter, we have laid the groundwork for a discussion of high-performance image filtering in Metal, as well as other applications of data-parallel programming. In the next chapter, we will apply data-parallel programming in Metal to a couple of interesting problems in image processing.

Chapter 14

Fundamentals of Image Processing

In this chapter, we will start exploring the world of image processing with the Metal shading language. We will create a framework capable of representing chains of image filters, then write a pair of image filters that will allow us to adjust the saturation and blur of an image. The end result will be an interactive app that allows you to control the image filter parameters in real-time.

Image processing is one of the chief applications of data-parallel programming. In many cases, an image filter only needs to consult one pixel or a small neighborhood of pixels in a source image to compute the value of each output pixel. When this is the case, the work of the image filter can be done in a parallel fashion. This is a perfect fit for modern GPU architecture, which uses many small cores to work on many pieces of data at once.

A Look Ahead

To motivate this chapter a little further, here is a snippet from the sample project. It illustrates how to succinctly create and chain together the desaturation and blur filters controlled by a user interface that can be dynamically adjusted.

```
context = [MBEContext newContext];
imageProvider = [MBEMainBundleTextureProvider
     textureProviderWithImageNamed:@"mandrill"
    context:context];
desaturateFilter = [MBESaturationAdjustmentFilter
     filterWithSaturationFactor:0.75
    context:context];
```

```
desaturateFilter.provider = self.imageProvider;
blurFilter = [MBEGaussianBlur2DFilter filterWithRadius:0.0
    context:context];
blurFilter.provider = desaturateFilter;
imageView.image = [UIImage imageWithMTLTexture:blurFilter.texture];
```

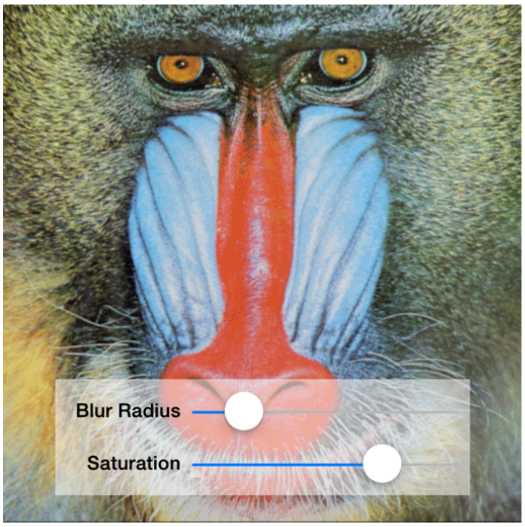


Figure 14.1: The sample application UI, allowing filter adjustment in real-time.

A Framework for Image Processing

Now that we have an idea of what we want the interface to our image processing framework to look like, we can talk about the practicalities of building such a framework. Much of the architecture of this framework was inspired by (Jargstorff 2004).

Texture Providers and Consumers

Each filter will the have the ability to configure its compute pipeline with its input and output textures and execute its kernel function.

Since we will be operating on images in the form of textures, we need an abstract way to refer to objects that produce and consume textures. Filters, for example, consume *and* produce textures, and we will also need a class for generating textures from images in the application bundle.

We abstract the idea of texture production with a protocol named `MBETextureProvider`:

```
@protocol MBETextureProvider <NSObject>
@property (nonatomic, readonly) id<MTLTexture> texture;
@end
```

This simple interface gives us a way to (synchronously) request a texture from a texture provider. It might cause an image filter to perform its filtering procedure, or load an image from disk. The important point is that we know we can retrieve a texture from any object that conforms to `MBETextureProvider`.

On the other hand, the `MBETextureConsumer` protocol allows us to tell an object which texture provider it should consume textures from:

```
@protocol MBETextureConsumer <NSObject>
@property (nonatomic, strong) id<MBETextureProvider> provider;
@end
```

When a texture consumer wants to operate on a texture, it requests the texture from its provider and operates on it.

An Image Filter Base Class

Abstractly, an image filter transforms one texture into another by performing an arbitrary operation on it. Our `MBEImageFilter` class does much of the work necessary to invoke a compute shader to produce one texture from another.

The image filter base class conforms to the texture provider and texture consumer protocols just discussed. Having filters behave as both texture providers and texture consumers allows us to chain filters together to perform multiple operations in sequence. Because of the serial nature of the command queue managed by the image context, it is guaranteed that each filter completes its work before its successor is allowed to execute.

Here is the relevant portion of the `MBEImageFilter` class' interface:

```
@interface MBEImageFilter : NSObject <MBETextureProvider, MBETextureConsumer>
@property (nonatomic, strong) MBEContext *context;
@property (nonatomic, strong) id<MTLComputePipelineState> pipeline;
@property (nonatomic, strong) id<MTLTexture> internalTexture;
@property (nonatomic, assign, getter=isDirty) BOOL dirty;
- (instancetype)initWithFunctionName:(NSString *)functionName
    context:(MBEContext *)context;
```

```
- (void)configureArgumentTableWithCommandEncoder:
    (id<MTLComputeCommandEncoder>)commandEncoder;
```

The filter must be instantiated with a function name and a context. These are used to create a compute pipeline state, which is then stored in the `pipeline` property.

The image filter maintains an internal texture that it uses as the output texture of its kernel function. This is so it can store the result of its computation and provide it to other filters. It can also be drawn to the screen or converted to an image.

Image filters may have any number of parameters that affect how they carry out their computation. When one of them changes, the internal texture is invalidated and the kernel function must be re-executed. The `dirty` flag allows filter subclasses to indicate when this is necessary. A filter will only be executed when its `dirty` flag is set to `YES`, which is done inside custom property setters.

The image filter base class contains an `-applyFilter` method that is invoked when it is asked to provide its output texture and it is currently dirty. This method creates a command buffer and command encoder, and dispatches its kernel function as discussed in the previous chapter.

Now that we have the necessary machinery in place, let's talk about a couple of example filters that will put it to use.

Building a Saturation Adjustment Filter

The first filter we will build is a saturation adjustment filter. The filter will have a configurable saturation factor that determines how much the filter should desaturate the input image. This factor can range from 0 to 1. When the saturation factor is 0, the output image will be a grayscale version of the input. For values between 0 and 1, the filter will produce an image in which the colors are more or less muted, by interpolating between the grayscale image and the input image.

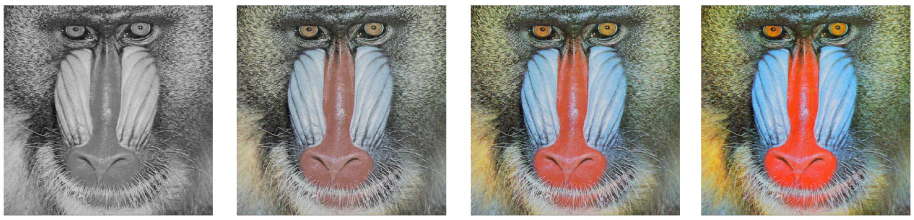


Figure 14.2: A sequence of images showing how increasing the saturation factor of the filter from 0 to 1 increases image saturation

Calculating Brightness from RGB

Every RGB color value has a corresponding brightness value, which we will represent by the symbol Y':

$$Y' = 0.299R + 0.587G + 0.114B$$

Note that the factors in the formula sum to 1. Their values are based on human perception of the intensity of different primary colors of light: the human eye is most sensitive to green light, followed by red light, and finally blue light. (These values are published as part of ITU-R Recommendation BT.601-7 (International Telecom Union 2011), section 2.5.1).

Replacing each color in an image with a gray pixel of its corresponding brightness results in a completely desaturated image that has the same perceptual brightness as the original image. This will be the task of our desaturation kernel function, which is presented below.

The Desaturation Kernel

To support passing the saturation factor to our kernel function, we create a single-membered struct called `AdjustSaturationUniforms`:

```
struct AdjustSaturationUniforms
{
    float saturationFactor;
};
```

The kernel function itself takes an input texture, an output texture, a reference to the uniform struct, and a 2D vector of integers with an attribute we didn't bother describing in detail in the previous chapter: `thread_position_in_grid`.

Recall from the previous chapter that we dispatch a two-dimensional set of threadgroups whose size is calculated from the dimensions of the source texture. The `thread_position_in_grid` attribute tells Metal to generate a coordinate vector that tells us where we are in the 2D grid that spans the entire dispatched set of work items, i.e. our current coordinates in the source texture.

We specify the intended access pattern for each texture parameter: `access::read` for the input texture, and `access::write` for the output texture. This restricts the set of functions we can call on these parameters.

```
kernel void adjust_saturation(texture2d<float, access::read> inTexture
     [[texture(0)]],
    texture2d<float, access::write> outTexture [[texture(1)]],
    constant AdjustSaturationUniforms &uniforms [[buffer(0)]],
    uint2 gid [[thread_position_in_grid]])
{
    float4 inColor = inTexture.read(gid);
    float value = dot(inColor.rgb, float3(0.299, 0.587, 0.114));
    float4 grayColor(value, value, value, 1.0);
    float4 outColor = mix(grayColor, inColor, uniforms.saturationFactor);
    outTexture.write(outColor, gid);
}
```

We read the source texel's color and calculate its brightness value based on the formula presented earlier. The `dot` function allows us to do this more succinctly than doing the three multiplications and two additions separately. We then generate a new color by duplicating the brightness into the RGB components, which makes a shade of gray.

To calculate the partially-desaturated output color, we use a function from the Metal standard library: `mix`, which takes two colors and a factor between 0 and 1. If the factor is 0, the first color is returned, and if it is 1, the second color is returned. In between, it blends them together using linear interpolation.

Finally, we write the resulting desaturated color into the output texture. Note that we previously assumed that the input and output textures had the same size, and that the dimensions of both were a multiple of our threadgroup size. If this were not the case, we would need to guard against reading outside the bounds of the input writing outside the bounds of the output texture.

The Saturation Adjustment Class

In order to drive the saturation adjustment kernel, we need to extend the image filter base class. This subclass is named `MBESaturationAdjustmentFilter`:

```
@interface MBESaturationAdjustmentFilter : MBEImageFilter
@property (nonatomic, assign) float saturationFactor;
+ (instancetype)filterWithSaturationFactor:(float)saturation
    context:(MBEContext *)context;
@end
```

This subclass calls the initializer with the name of the desaturation kernel function and passes through a reference to the context it should operate on.

Setting the `saturationFactor` property causes the filter to set its `dirty` property, which causes the desaturated image to be re-computed lazily when its `texture` property is requested.

The subclass' implementation of `-configureArgumentTableWithCommandEncoder:` consists of boilerplate to copy the saturation factor into a Metal buffer. It is not shown here.

We now have a complete filter class and kernel function for performing image desaturation.

Blur

The next type of image filter we will look at is the blur filter.

Blurring an image involves mixing the color of each texel with the colors of adjacent texels. Mathematically, a blur filter is a weighted average of a neighborhood of texels (such an operation is called a *convolution*). The size of the neighborhood is called the *radius* of the filter. A small radius will average fewer texels together and produce a less blurry image.

Box Blur

The simplest kind of blur is a *box blur*. The box blur gives equal weight to all the nearby texels, computing their average. Box blurs are easy to compute, but can produce ugly artifacts, since they give undue weight to noise.

Suppose we choose a box blur of radius 1. Then each texel in the output image will be the average of the input texel and its 8 nearest neighbors. Each of the 9 colors is given an equal weight of 1/9.

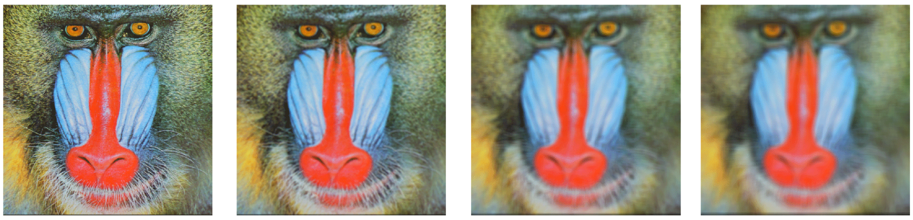


Figure 14.3: A box blur filter averages the neighborhood around each texel.

Box blurs are easy, but they don't produce very satisfying results. Instead, we will use a somewhat more sophisticated blur filter, the Gaussian blur.

Gaussian Blur

In contrast to the box blur, the Gaussian blur gives unequal weights to the neighboring texels, giving more weight to the texels that are closer, and less weight to those that are farther away. In fact, the weights are computed according to a 2D *normal distribution* about the current texel:

$$G_\sigma(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where x and y are the distances along the x and y axis, respectively, between the texel being processed and its neighbor. σ is the standard deviation of the distribution and is equal to half the radius by default.

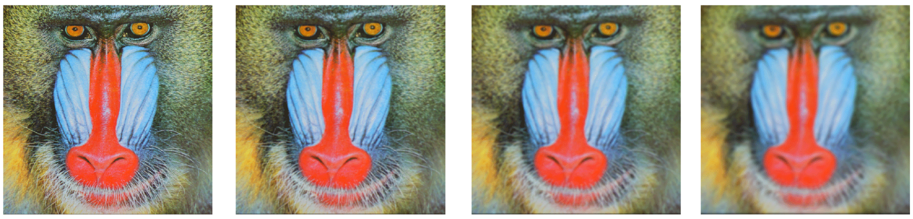


Figure 14.4: The Gaussian blur filter. Increasing the filter radius creates a smoother image. The maximum radius shown here is 7.

The Blur Shader

Computing the blur weights for a Gaussian filter is expensive to do in a kernel function, especially for a filter with a large radius. Because of this, we will precompute the table of weights and provide it to the Gaussian blur kernel function as a texture. This texture has a pixel format of `MTLPixelFormatR32Float`, which is a single channel 32-bit floating-point format. Each texel holds a weight between 0 and 1, and all of the weights sum up to 1.

Inside the kernel function, we iterate over the neighborhood of the current texel, reading each texel and its corresponding weight from the look-up table. We then add the weighted color to an accumulated value. Once we have finished adding up all the weighted color values, we write the final color to the output texture.

```
kernel void gaussian_blur_2d(texture2d<float, access::read> inTexture
     [[texture(0)]],
    texture2d<float, access::write> outTexture [[texture(1)]],
    texture2d<float, access::read> weights [[texture(2)]],
    uint2 gid [[thread_position_in_grid]])
{
    int size = blurKernel.get_width();
    int radius = size / 2;

    float4 accumColor(0, 0, 0, 0);
    for (int j = 0; j < size; ++j)
    {
        for (int i = 0; i < size; ++i)
        {
            uint2 kernelIndex(i, j);
            uint2 textureIndex(gid.x + (i - radius), gid.y + (j - radius));
            float4 color = inTexture.read(textureIndex).rgba;
            float4 weight = weights.read(kernelIndex).rrrr;
            accumColor += weight * color;
        }
    }

    outTexture.write(float4(accumColor.rgb, 1), gid);
}
```

The Filter Class

The blur filter class, `MBEGaussianBlur2DFilter` derives from the image filter base class. Its implementation of `-configureArgumentTableWithCommandEncoder:` lazily generates the blur weights and sets the look-up table texture on the command encoder as the third parameter (argument table index 2).

```
- (void)configureArgumentTableWithCommandEncoder:
     (id<MTLComputeCommandEncoder>)commandEncoder
{
    if (!self.blurWeightTexture)
    {
        [self generateBlurWeightTexture];
    }
    [commandEncoder setTexture:self.blurWeightTexture atIndex:2];
}
```

The `-generateBlurWeightTexture` method uses the 2D standard distribution formula above to calculate a weight matrix of the appropriate size and copy the values into a Metal texture.

This completes our implementation of the Gaussian blur filter class and shader. Now we need to discuss how we chain filters together and get the final image to the screen.

Chaining Image Filters

Consider again the code from the beginning of the chapter:

```
context = [MBEContext newContext];
imageProvider = [MBEMainBundleTextureProvider
     textureProviderWithImageNamed:@"mandrill"
    context:context];
desaturateFilter = [MBESaturationAdjustmentFilter
     filterWithSaturationFactor:0.75
    context:context];
desaturateFilter.provider = self.imageProvider;
blurFilter = [MBEGaussianBlur2DFilter filterWithRadius:0.0
    context:context];
blurFilter.provider = desaturateFilter;
imageView.image = [UIImage imageWithMTLTexture:blurFilter.texture];
```

The main bundle image provider is a utility for loading images into Metal textures and acts as the beginning of the chain. It is set as the texture provider for the desaturate filter, which in turn is set as the texture provider of the blur filter.

Requesting the blur filter's texture is actually what sets the image filtering process into motion. This causes the blur filter to request the desaturation filter's texture, which in turn causes the desaturation kernel to be dispatched synchronously. Once it is complete, the blur filter takes the desaturated texture as input and dispatches its own kernel function.

Now that we have a filtered image, we could use it to render a textured quad (or other surface) with Metal. Suppose we want to display it with UIKit, though? How do we do the opposite task of creating a Metal texture from a `UIImage` and instead create a `UIImage` *from* a Metal texture?

Creating a UIImage from a Texture

The most efficient way to do this is with the image utilities in Core Graphics. First, a temporary buffer is created, into which the pixel data of the Metal texture is read. This buffer can be wrapped with a `CGDataProviderRef`, which is then used to create a `CGImageRef`. We can then create a `UIImage` instance that wraps this `CGImageRef`.

The sample code implements a category on `UIImage` that does all of this, in a method named `+imageWithMTLTexture:`. It is not included here for the sake of brevity, but it makes for interesting reading.

It bears mentioning again that this is **not** the most efficient way to get the filtered image on the screen. Creating an image from a texture uses extra memory, takes CPU time, and requires the Core Animation compositor to copy the image data back into a texture for display. All of this is costly and can be avoided in many cases. Fortunately, throughout the course of this book, we've seen plenty of ways to use textures that don't involve displaying them with UIKit.

Driving Image Processing Asynchronously

Above, we mentioned that the filters dispatch their kernel functions *synchronously*. Since image processing is computationally intensive, we need a way to do the work on a background thread in order to keep the user interface responsive.

Committing a command buffer to a command queue is inherently thread-safe, but controlling other aspects of concurrency are the responsibility of the programmer.

Fortunately, Grand Central Dispatch makes our work easy. Since we will only be invoking image filters in response to user actions on the main thread, we can use a pair of `dispatch_async` calls to relocate our image processing work onto a background thread, asynchronously updating the image view on the main thread when all of the filters are done processing.

We will use a crude mutex in the form of an `atomic` 64-bit integer property on the view controller that increments every time an update is requested. The value of this counter is *captured* by the block that is enqueued. If another user event has not occurred by the time the block executes on the background queue, the image filters are allowed to execute, and the UI is refreshed.

```
- (void)updateImage
{
    ++self.jobIndex;
    uint64_t currentJobIndex = self.jobIndex;

    float blurRadius = self.blurRadiusSlider.value;
    float saturation = self.saturationSlider.value;

    dispatch_async(self.renderingQueue, ^{
        if (currentJobIndex != self.jobIndex)
            return;

        self.blurFilter.radius = blurRadius;
        self.desaturateFilter.saturationFactor = saturation;

        id<MTLTexture> texture = self.blurFilter.texture;
        UIImage *image = [UIImage imageWithMTLTexture:texture];

        dispatch_async(dispatch_get_main_queue(), ^{
            self.imageView.image = image;
        });
    });
}
```

The Sample Project

The sample code for this chapter is in the 14-ImageProcessing directory.

In this chapter, we took our next steps in parallel computation with Metal and saw a couple of examples of image filters that can run efficiently on the GPU. You can now use the framework presented here to create your own effects and use the power of Metal to run them efficiently on the GPU.

Bibliography

Barrett, Sean. 2014. "stb Single-File Public Domain Libraries for C/C++." https://github.com/nothings/stb.

Blinn, James F. 1977. "Models of Light Reflection for Computer Synthesized Pictures." In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, 192–98. SIGGRAPH '77. New York, NY, USA: ACM. doi:10.1145/563858.563893.

Boxley, Paul. 2011. "Terrain Generation with the Diamond Square Algorithm." http://www.paulboxley.com/blog/2011/03/terrain-generation-mark-one.

Carpenter, Loren. 1984. "The A-Buffer, an Antialiased Hidden Surface Method." In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, 103–8. SIGGRAPH '84. New York, NY, USA: ACM. doi:10.1145/800031.808585.

Esfahbod, Behdad. 2014. "GLyphy." https://github.com/behdad/glyphy.

Giesen, Fabien. 2011. "A Trip Through the Graphics Pipeline 2011, Part 7," July. https://fgiesen.wordpress.com/2011/07/08/a-trip-through-the-graphics-pipeline-2011-part-7/.

Glassner, Andrew. 2015. "Interpreting Alpha." *Journal of Computer Graphics Techniques (JCGT)* 4 (2): 30–44. http://jcgt.org/published/0004/02/03/.

Gortler, Steven J. 2012. *Foundations of 3D Computer Graphics*. The MIT Press.

Green, Chris. 2007. "Improved Alpha-Tested Magnification for Vector Textures and Special Effects." In *ACM SIGGRAPH 2007 Courses*, 9–18. SIGGRAPH '07. New York, NY, USA: ACM. doi:10.1145/1281500.1281665.

Grevera, George J. 2004. "The 'Dead Reckoning' Signed Distance Transform." *Comput. Vis. Image Underst.* 95 (3). New York, NY, USA: Elsevier Science Inc.: 317–33. doi:10.1016/j.cviu.2004.05.002.

Gustavson, Stefan. 2012. "2D Shape Rendering by Distance Fields." In *OpenGL Insights*, edited by Patrick Cozzi and Christophe Riccio, 173–82. CRC Press.

Hughes, J.F., A. Van Dam, J.D. Foley, and S.K. Feiner. 2013. *Computer Graphics: Principles and Practice*. 3rd ed. Addison-Wesley.

International Telecom Union. 2011. *RECOMMENDATION ITU-R BT.601-7.* https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf.

Jargstorff, Frank. 2004. "A Framework for Image Processing." In *GPU Gems*, edited by Randima Fernando. Addison-Wesley Professional. http://http.developer.nvidia.com/GPUGems/gpugems_ch27.html.

Khronos Group. 2013. "KTX File Format Specification." Edited by Mark Callow, Georg Kolling, and Jacob Ström. https://www.khronos.org/opengles/sdk/tools/KTX/file_format_spec/.

Lengyel, Eric. 2011. *Mathematics for 3D Game Programming and Computer Graphics*. 3rd ed. Boston, MA, United States: Course Technology Press.

McGuire, Morgan, and Louis Bavoil. 2013. "Weighted Blended Order-Independent Transparency." *Journal of Computer Graphics Techniques (JCGT)* 2 (2): 122–41. http://jcgt.org/published/0002/02/09/.

Miller, Gavin S P. 1986. "The Definition and Rendering of Terrain Maps." In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, 39–48. SIGGRAPH '86. New York, NY, USA: ACM. doi:10.1145/15922.15890.

Wavefront Technologies. 1991. "Appendix B1. Object Files." In *Programmer's Reference Manual for the Advanced Visualizer*. http://www.cs.utah.edu/~boulos/cs3505/obj_spec.pdf.