Metal Programming Guide

Comprehensive Tutorial and Reference via Swift

Janie Clayton

Contents

Preface

In 2014, Apple announced a game-changing innovation that shook up 30 years of established paradigms and caused general excitement in the field of computing. Then, 10 minutes later, it introduced Swift.

If you ask most Cocoa developers what the most exciting development was of 2014, they will overwhelmingly mention Swift. But Swift was not the only innovation Apple announced that year—the game-changer it announced was Metal, a new GPU programming framework. For decades, the default low-level graphics framework on iOS was OpenGL ES. OpenGL had grown very limited and crufty over the years. It did not take full advantage of the GPU. It did not allow general-purpose GPU programming as the newer APIs and frameworks did. Rather than adopt of one those frameworks, Apple opted to do what it usually does and roll its own implementation.

Over the years, I have seen many people attempt to pick up Metal and fail. They have different reasons for picking up Metal. Many want to prove they can learn something difficult. Some are interested in how low-level graphics work. Some want to do scientific computing. Graphics programming is an incredibly large and diverse topic, and it's easy to get overwhelmed by the vast amount of information out there to learn. This book is the product of my years-long attempts at learning graphics programming and gaining an understanding of where people tend to get lost and give up.

Who Should Read This Book

This book is targeted primarily at iOS programmers who are interested in graphics programming but don't know quite where to begin. The book's format was born of many years of frustration in my attempt to learn OpenGL. There were a multitude of books that went over the framework in loving detail but never made clear what someone could do with the framework. They assumed a knowledge of terminology such as textures and projection matrices that led to a lot of confusion for someone just getting started.

This book assumes a base knowledge of Swift. If your intention is simply to understand the concepts, you can get by without being a Swift expert. However, if you're new to iOS and are counting on this book to explain Swift and Xcode, then this book is not the best place to start. There are a number of great resources out there for learning Swift and iOS concepts.

This book is not targeted at people who are pushing the boundaries of what Metal can do. Metal is a very broad topic, and this book focuses on an overview of the topic to give newcomers enough information to get started. It does not delve into the nuances of building a Unity-level gaming engine or highly complex recurrent neural networks. If you have such goals, this book is a good first step and should provide you with a solid foundation to build your knowledge on.

How This Book Is Organized

In this book, we discuss the Metal programming framework. There are two components to this framework: graphics and compute. This book provides a conceptual overview of the concepts necessary to create something useful along with specific code implementations in Metal.

At the beginning of the book are foundational chapters that explain concepts you need to understand in order to work with later chapters. It can be tempting to jump directly into the final chapters on neural networking, but those chapters assume that you have read most of the ones before them and will not make sense if you don’t have a cursory understanding of the concepts introduced earlier.

Part I, “Metal Basics,” covers introductory Metal and graphics concepts. Chapters 1, 2, and 3 correlate to much of what you would find in an introductory Metal tutorial on a website or at a conference.

In Part II, “Rendering and Graphics,” you move into more graphics-specific libraries and concepts. Chapters 4 through 14 cover specific Metal classes, mathematics, and an introduction to the Metal Shading Language. Some of these chapters are applicable to both graphics and data processing, which is discussed in Part III, “Data Parallel Programming.”

Part III introduces you to the Metal compute pipeline. Chapters 15 through 20 detail the Metal compute pipeline and give an overview of practical applications of GPGPU programming.

Following is a brief description of each chapter. Part I focuses on the history and foundation of Metal graphics programming:

• Chapter 1, “What Is Metal?,” gives a general introduction of Metal programming and a history of graphics programming. It also explains the scope of this book.

• Chapter 2, “Overview of Rendering and Raster Graphics,” details how the Metal render pipeline works under the hood. This chapter gives you an idea about how all the parts connect together.

• Chapter 3, “Your First Metal Application (Hello, Triangle!),” gives a bare-bones project to introduce you to all the Metal objects necessary to get the most basic Metal project up and running.

Part II introduces the Metal render pipeline and concepts necessary to implement 3D graphics on the iPhone.

• Chapter 4, “Essential Mathematics for Graphics,” gives a refresher of mathematical concepts used in shader programming. This chapter is meant as a gentle refresher for readers who have not used a TI-83 in at least 10 years, but it should be useful to anyone.

• Chapter 5, “Introduction to Shaders,” walks you through implementing a slightly more complex shader. This chapter helps you to learn how to adapt algorithms and shaders in other languages to your applications.

• Chapter 6, “Metal Resources and Memory Management,” goes over MTLResource objects and how they play into memory management.

• Chapter 7, “Libraries, Functions, and Pipeline States,” explains how Metal shaders are compiled and accessed by the CPU.

• Chapter 8, “2D Drawing,” goes a step beyond Chapter 3 and gives a more in-depth explanation of the objects necessary to create a 2D application.

• Chapter 9, “Introduction to 3D Drawing,” gives an overview of 3D graphics programming concepts.

• Chapter 10, “Advanced 3D Drawing,” explains how to apply these 3D graphics concepts to your Metal applications.

• Chapter 11, “Interfacing with Model I/O,” explores Apple’s framework for importing 3D models and assets from programs such as Blender and Maya.

• Chapter 12, “Texturing and Sampling,” explains how to import a texture and apply it to a model object.

• Chapter 13, “Multipass Rendering Techniques,” gives details about how to implement more computationally expensive rendering techniques without destroying your performance.

• Chapter 14, “Geometry Unleashed: Tessellation in Metal,” discusses how to take advantage of tessellation to utilize less detailed meshes.

Part III introduces how to use the GPU for general-purpose computing, specifically image processing and neural networks.

• Chapter 15, “The Metal Compute Pipeline,” gives the details on how the compute encoder differs from the render encoder and how to set up a data parallel Metal application.

• Chapter 16, “Image Processing in Metal,” goes over some foundational image processing concepts.

• Chapter 17, “Machine Vision,” builds on the foundational concepts introduced in the previous chapter to create computer vision applications.

• Chapter 18, “Metal Performance Shaders Framework,” gives the details on what objects exist in the Metal Performance Shaders framework to implement image processing and linear algebra.

• Chapter 19, “Neural Network Concepts,” explains what a neural network is and gives an overview of the components necessary to build one.

• Chapter 20, “Convolutional Neural Networks,” teaches you how to take advantage of the Metal Performance Shaders to build convolutional neural network graphs.

Example Code

Throughout the book, many code examples are available to test the contents in each chapter. It is recommended that you download the sample code while you read this book. It can give you good hands-on experience and provide you with valuable insight to understand the contents of each chapter.

The sample code was written in Swift 4 with Xcode 9. It is the author’s intention to maintain the code for future versions of Swift and Xcode. The sample code is intended to supplement the material in this book and for each project to build on the previous ones.

The source code in this book can be found on GitHub at https://github.com/RedQueenCoder/Metal-Programming-Guide.

Conventions Used in This Book

The following typographical conventions are used in this book:

• *Italic* indicates new terms and menu options.

• Constant width is used for program listings, as well as in the text to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Register your copy of *Metal Programming Guide* at informit.com for convenient access to downloads, updates, and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134668949) and click Submit. Once the process is complete, you will find any available bonus content under “Registered Products.”

I: Metal Basics

1. What Is Metal?

It is a point where our old models must be discarded and a new reality rules.

—Vernor Vinge

In 2014, Apple introduced a new low-level GPU programming framework for iOS: Metal. A year later, Metal came to macOS, followed by watchOS and tvOS. Apple devices have two "brains" that can be programmed to create applications: a central processing unit (CPU) and a graphics processing unit (GPU). The GPU is a specialized processor that does floating-point math in parallel very quickly and efficiently. These tasks are expensive on the CPU because they can't be done in parallel, so various frameworks and APIs have been created to offload these expensive tasks to the processor that is best equipped to do them.

Floating-point math is integral to graphics programming, but that's not the only application for it. Nongraphics GPU programming is known as General Purpose GPU (GPGPU) programming. Older graphics paradigms, such as OpenGL, were developed before GPGPU programming was feasible. OpenGL was sufficient for the first 5 years of the iPhone, but as it became more powerful and more tasks became possible, it evolved.

History of Graphics APIs

Three-dimensional computer graphics date back to the 1960s and 1970s. Work by pioneers such as James Blinn and Edwin Catmull paved the way for many of today's innovations. But computer graphics didn't become ubiquitous until the home computer revolution in the 1980s and 1990s.

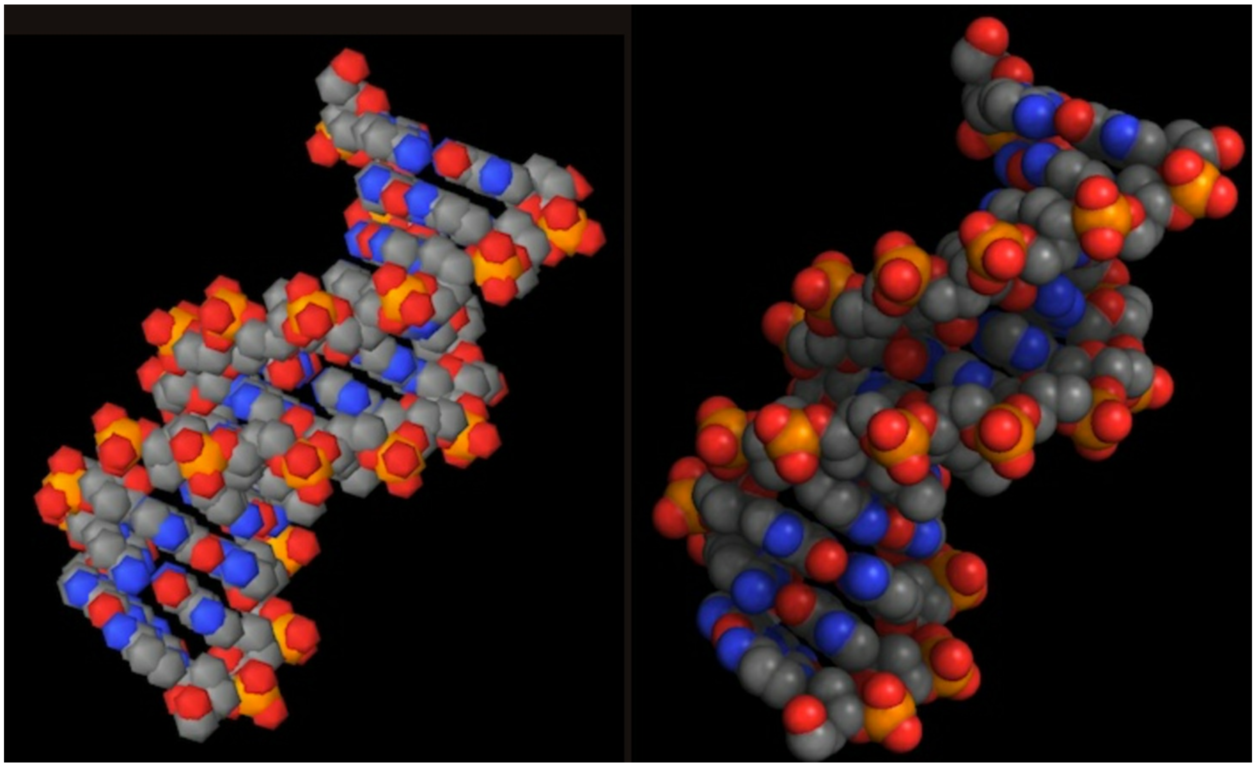
During the 1980s, writing cross-platform software was incredibly difficult, especially for games. Every computer and video-gaming system had its own proprietary hardware and software drivers. Every time you wanted to release your software on a new device, you had to write device-specific drivers, software, and user interfaces. This work was repetitive and felt largely unnecessary.

In 1981, the company Silicon Graphics (SGI) was founded. It manufactured high-performance 3D workstations. Like all other manufacturers of 3D workstations, it had its own proprietary 3D graphics API called IRIS GL. It dominated the market for much of the 1980s, but by the early 1990s, its market share had eroded. In 1992, in an attempt to prevent further erosion of market share, it opened the IRIS GL API and renamed it OpenGL.

OpenGL didn't take off as the de facto 3D graphics API until 1997. The video-game company idSoftware, led by lead programmer John Carmack, developed a follow-up to its best-selling video game, *Doom*. It was called *Quake*. Every game platform was eager to have a port of *Quake* added to its platform. Carmack didn't want to have to write unique drivers for every variation of hardware and OS on the market, so he chose OpenGL as a standard. He told the manufacturers that if they didn't support OpenGL, they could not port *Quake* to their platform. The manufacturers complied and wrote drivers to support OpenGL. Since every manufacturer had done the work to support OpenGL, many game developers began to exclusively support OpenGL because they knew it would work on every platform. Voila! An industry standard was born.

In 2004, OpenGL 2.0 was released. This was a significant advance in OpenGL because it introduced the OpenGL Shading Language. This C-like language allowed programmers to write programs for the GPU and to modify the transformation and fragment-shading stages of the programmable pipeline (see Figure 1.1).

Figure 1.1. *The application Molecules. On the left is the app with a fixed function pipeline. On the right is the version enhanced with shaders.*



By the time shaders were added to OpenGL, the API was starting to get rather crufty. When OpenGL was initially released, GPUs were not particularly powerful. They couldn’t handle much work. The original OpenGL API was designed to push as much work as possible to the CPU because the GPU was a bottleneck. By the time shaders were announced, this limitation had changed dramatically. GPUs had gotten progressively more powerful, and many of the original ways of performing tasks were incredibly inefficient. New methods were introduced over time to accommodate for this new reality, but the old methods were never removed from the API. There were multiple ways of to perform the same tasks, and some were more efficient than others.

In 2003, a streamlined version of OpenGL was released: OpenGL ES. OpenGL ES is a subset of OpenGL. It removed all of the old, crufty implementations that were no longer the optimal way of allocating work to the GPU. It was developed for embedded devices, such as video-game consoles and smart phones, but any program written in OpenGL ES can work on a program that uses OpenGL, with some small modifications to the shaders. In 2007, OpenGL ES 2.0 was released. This version incorporated shaders and a programmable pipeline.

It was also in 2007 that the first iPhone was released. The graphics frameworks and APIs available in the first iOS SDK were incredibly limited, but the iPhone did support OpenGL ES. Most developers who wanted to adopt sophisticated graphics for games and applications had to program everything in OpenGL ES. Many programmers who had years of experience working with OpenGL were able to create sophisticated graphics for the iPhone at a time when the SDK was in its infancy. In 2010, Apple began to support OpenGL ES 2.0, along with programmable shaders, on the iPhone.

Over the years, Apple has developed higher-level abstractions for graphics on the iPhone: Sprite Kit, Scene Kit, and so on. But if you wanted to program the GPU directly, you had to use OpenGL ES. You also were constrained to programming the GPU for graphics. You couldn’t utilize the GPU for general-purpose programming, nor could you take advantage of the tight hardware/software integration that Apple has with its other frameworks.

Metal: The New Way to Do Graphics on Apple Platforms

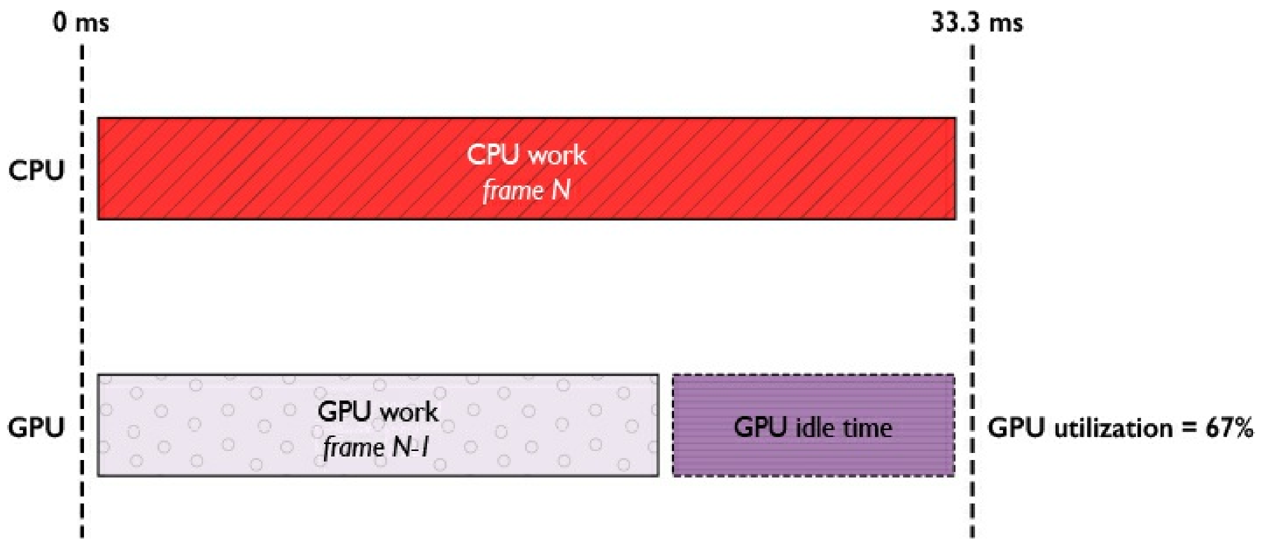
Today, we carry supercomputers in our pockets. The iPhone is approaching the computing capability of many of our laptops. Even with all this extra power, we’re still being rate limited by the OpenGL API. Because OpenGL must accommodate all combinations and variations of GPU and CPU, it is never able to fully take advantage of the deep integration that Apple has for all of its products. Apple knows exactly which chip it will use for its GPU programming and can optimize the framework to work specifically and directly with that chip’s idiosyncrasies.

Beyond the deep integration that is now possible by controlling the entire manufacturing chain, there were some structural issues with OpenGL that prevented it from being as efficient as it could be. There are many expensive operations that are scheduled to occur on every draw call that don’t need to be. By changing the order of operations and moving expensive work outside the draw call, Metal frees up more processor bandwidth. Bandwidth can be freed up even further by giving programmers absolute control over how they want to schedule work with the GPU.

Balancing Work between the CPU and the GPU

The CPU and the GPU work in offset. The CPU prepares commands for the GPU and then sends them to the GPU. While the GPU is processing the commands it has received, the CPU is preparing the next batch of commands. In graphics programming, you have a responsibility to balance the workloads between the CPU and the GPU. Batching and sending work from the CPU to the GPU is expensive in terms of time. The more work you send the GPU, the more time it takes. In traditional OpenGL applications, many times, this offset wasn’t perfectly balanced. Often, the CPU is unable to send enough work to the GPU to keep it busy the entire frame, so the GPU sits idle, as shown in Figure 1.2. Metal increases the efficiency of this offset so that the GPU can be used to its full capacity. Part of this increased efficiency is accomplished by eliminating the need to copy memory between the CPU and GPU.

Figure 1.2. *In older graphics APIs, the CPU cannot batch enough work to keep the GPU fully utilized.*

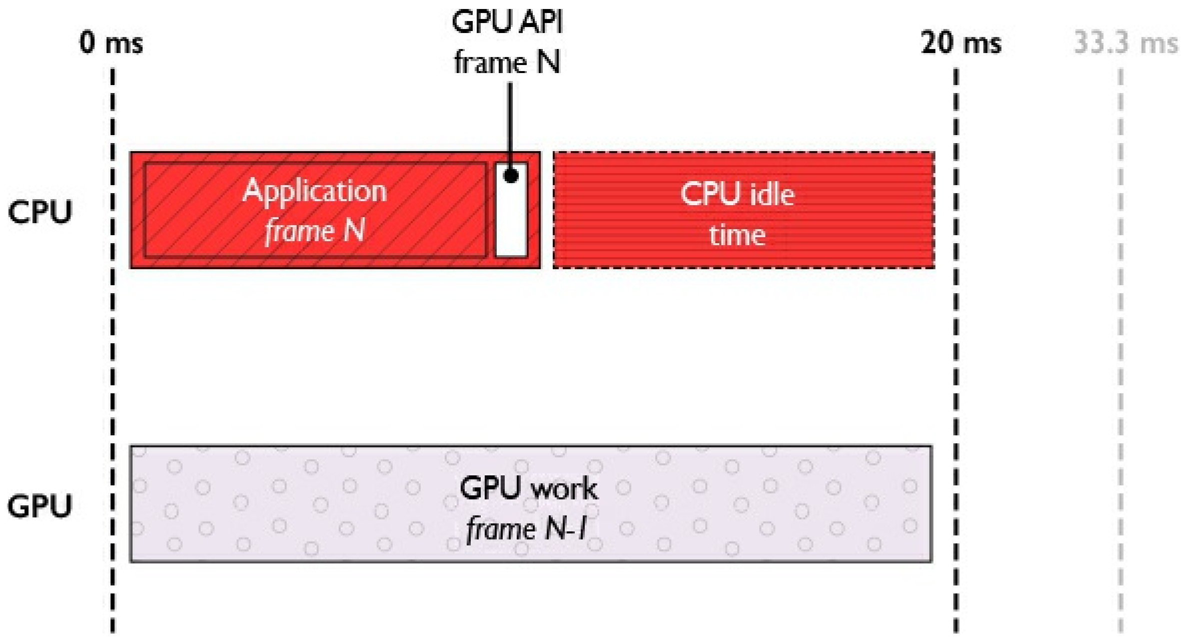


Copying Memory

OpenGL is set up the way it is because, until recently, the CPU and the GPU were always on separate chips. It required that data be copied from one chip to another, which entails expensive thread locks to ensure that race conditions don't occur and overwrite data.

On every iteration of the iPhone, the CPU and GPU occupy the same chip. This negates the need to copy data back and forth, eliminating an expensive operation that is vestigial from prior computer architectures (see Figure 1.3). The CPU simply grants access to the GPU to modify the data directly. Starting with the A7 chip, Apple took full advantage of this design by supporting Metal.

Figure 1.3. *By eliminating the need to copy memory, the expensive batching of commands to be copied is eliminated.*



Direct Control of the Command Buffer

The command buffer is the central control object for a Metal application. Metal differs from OpenGL because it gives the programmer access to this control rather than abstracting it away in the name of simplification. Exposing the command buffer to the programmer gives more flexibility in when and how render buffers are executed. In OpenGL, the OpenGL driver—not the developer—determines the order of work. By exposing the buffer's complexity to developers and allowing them to queue work in their preferred order, Metal provides far more control over the performance of the application. Multiple threads can be created, and buffer execution can be delayed.

Precompiled Shaders

In OpenGL, shaders are compiled at runtime. Compiling shaders is expensive. That work, before Metal, had to be done on each draw call. It doesn't make a lot of sense to have to compile these shaders at load time. This work can be moved outside the load time to application build time. Precompiled shaders can be stored in a library where they can be referenced at runtime, which moves long and expensive work to a place where it doesn't affect the user experience.

Prevalidated State

In other graphics APIs, on every draw call, the application has to check to ensure that the state on the API is valid in order to avoid sending bad commands to the GPU. The state in the application changes, but not on every draw call. You, as the developer, encode those state changes in the command encoder. The command encoder is committed to the command buffer, and the expensive state validation occurs only when the state actually changes.

The Apple engineers made the Metal API as thin as possible. A thin design means that less work needs to run when batching and sending work from the CPU to the GPU. It eliminates unnecessary work that is done silently by the GPU because the GPU's work is abstracted away from the programmers to make their job easier. Making this work less expensive to send means that programmers are no longer rate limited by the CPU. They can utilize the GPU to its full potential.

Metal in Context: How Metal Complements and Supports Other Platform Frameworks

The Metal framework is an extraordinary accomplishment, but it doesn't exist on its own. It's part of the vast Cocoa framework ecosystem. It plays an important role, but it is not a solo actor. It is integrated into many other frameworks, and it is dependent on other frameworks.

Several frameworks were updated to utilize Metal, including SceneKit and SpriteKit. If you want to do something that is not highly specialized, you can take advantage of Metal by using one of these frameworks. They are also set up to integrate easily with Metal. If you want to write a custom shader for a SceneKit project, it's easy to drop down to Metal for the parts you need while still taking advantage of the higher-level abstraction that SceneKit provides.

Metal follows the law of conservation of data. It doesn't create or destroy data; it merely modifies it. Therefore, Metal needs a data source. One framework in Cocoa that is specifically set up to act as a data source is the Model I/O framework. Model I/O provides an easy way to bring in vertex data from 3D modeling programs. It was designed to integrate tightly with Metal.

You can let Model I/O do all the heavy lifting of parsing data files and loading vertex buffers so that you can focus on what you want to do with the data rather than how to get it into your application.

Summary

Apple’s revolutionary new GPU programming framework resolves many longstanding issues with traditional graphics programming APIs. By taking advantage of tight hardware and software integration, Apple squeezes every last bit of functionality from the GPU. By moving expensive tasks outside of draw calls, Apple can make sending commands to the GPU less expensive, freeing up more CPU time to improve other aspects of the application. Metal has also been designed to integrate nicely with other Apple frameworks, making it easier than ever to take advantage of these frameworks for the common tasks you don’t need to drop down to the Metal for.

2. Overview of Rendering and Raster Graphics

The basic tool for the manipulation of reality is the manipulation of words. If you can control the meaning of words, you can control the people who must use them.

—Philip K. Dick

The end result of your Metal code is that an image is rendered to the screen. But how does it get there? How does Metal take a bunch of data and translate it to something you can see?

Many people who start out with an interest in 3D graphics programming find themselves drowning in a sea of unknowns. What are *shaders*? What are *buffers*? What do all these numbers represent?

The way we think about how 3D graphics works and how the computer thinks about it are vastly different. We can conceptualize picking up a crayon and drawing a stick figure on a piece of paper, but that’s not how the computer does it. It’s our responsibility as programmers to format our directions in a way that the computer can understand.

This chapter gives you a better understanding of how the computer takes your data and turns it into something you can see on screen.

Representing the GPU

The keystone of this book and the bedrock of all the concepts you need to understand all comes back to the graphics processing unit (GPU). This entire book revolves around how to program the GPU, so it makes sense to give a clear explanation of how Metal represents the GPU and how programmers interact with it.

The GPU is represented in Metal by an object that conforms to the MTLDevice protocol. The MTLDevice is responsible for creating and managing a variety of persistent and transient objects used to process data and render it to the screen. There is a single method call to create a default system device, which is the only way that the GPU can be represented in software. This ensures safety when sending data to and from the GPU.

The MTLDevice allows you to create command queues. The queues hold command buffers, and the buffers in turn contain encoders that attach commands for the GPU. Don’t worry if this is slightly confusing; it’s described in detail throughout this chapter.

Command buffers store the instructions necessary to render a frame. The instructions include commands to set state, which controls how drawing is done, and the draw calls that cause the GPU to do the actual work of rendering. These are transient objects that are single use. Once the buffer’s work has been executed, the buffer is deallocated.

Commands are encoded in the command buffer. Each command is executed in the order that it is enqueued. Once all the commands are enqueued, the command buffer is committed and submitted to the command queue.

A command queue accepts an ordered list of command buffers that the GPU will execute.

Command buffers are executed in the order in which they were committed to the command queue. Command queues are expensive to create, so they are reused rather than destroyed when they have finished executing their current instructions.

Now that the groundwork has been laid, let’s explore how this works.

Render State Configuration

We can conceptualize the work done by Metal as a series of transformations that are linked together in a sort of rendering pipeline. The pipeline performs a series of steps to prepare data to be processed by the GPU (shown in Figure 2.1):

1. Data preparation for the GPU

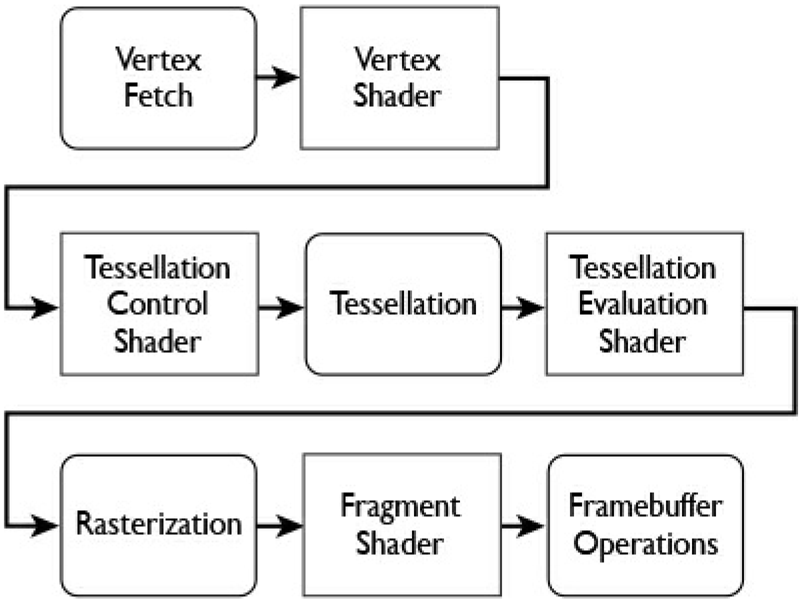
2. Vertex processing

3. Primitive assembly

4. Fragment shading

5. Raster output

Figure 2.1. *The path your data travels on its way to the screen*



One of the motivations associated with the creation of Metal was to move CPU-expensive work to the beginning of the render process rather than forcing the program to unnecessarily pay that cost. One of those tasks is the process of state validation.

Metal uses the MTLRenderPipelineState protocol. A render pipeline state object contains a collection of state that can be quickly set on a render command encoder, allowing you to avoid calling many state-setting methods, additionally ensuring that the state you set is internally consistent, because it was validated at the time the pipeline was created. The MTLRenderPipelineDescriptor has properties that allow you to configure how a render pipeline state object is created.

The MTLRenderPipelineDescriptor sets up a prevalidated state. Rather than checking that the state is valid on every draw call, this work is moved to the creation of the descriptor.

The pipeline state that can be prevalidated includes the following settings:

• Specifying shader functions

• Attaching color, depth, and stencil data

• Raster and visibility state

• Tessellation state

The MTLRenderPipelineState can then be passed to the command encoder. Since Metal knows that the render state is valid, it doesn't need to keep referring to it on each draw call, thus eliminating an expensive and unnecessary step.

Preparing Data for the GPU

Most assets that you will use to create 3D Metal applications will come from a program such as Blender or Maya. These digital content creation programs create formatted files that contain vertex information describing whatever model you are trying to create.

It describes the position and color of each vertex in your shape. It describes how those vertices are connected to generate meshes that create the appearance of a 3D model.

That vertex information is brought into your Metal program as an array of values. This can be done in a few different ways. If the file format is supported by Apple's Model/IO framework, you can import the model using that framework. You can write your own file parser if it's not. You could even manually enter all the vertex data into the application, but that approach quickly gets overwhelming.

The application now needs a way to understand that these arrays of values are the data that it will send to the GPU to generate images to the screen, so you need to encode commands to a command buffer.

There are four ways to encode data for the command buffer:

• MTLRenderCommandEncoder

• MTLComputeCommandEncoder

• MTLBlitCommandEncoder

• MTLParallelRenderCommandEncoder

You choose what type of command encoder you need depending on what kind of work you want the GPU to perform. In this chapter, we're focusing on the graphics side of things, so we will focus on the MTLRenderCommandEncoder. The other encoders are covered in the chapters related to their functionality.

The render encoder needs to prepare data sources for the vertex and fragment shaders. This can be done in the form of buffers. To prepare those array of vertices in a way that the GPU can understand, you need to create vertex and fragment buffers that use these arrays as their data source. The buffer also needs to allocate enough memory to hold the arrays of vertex values.

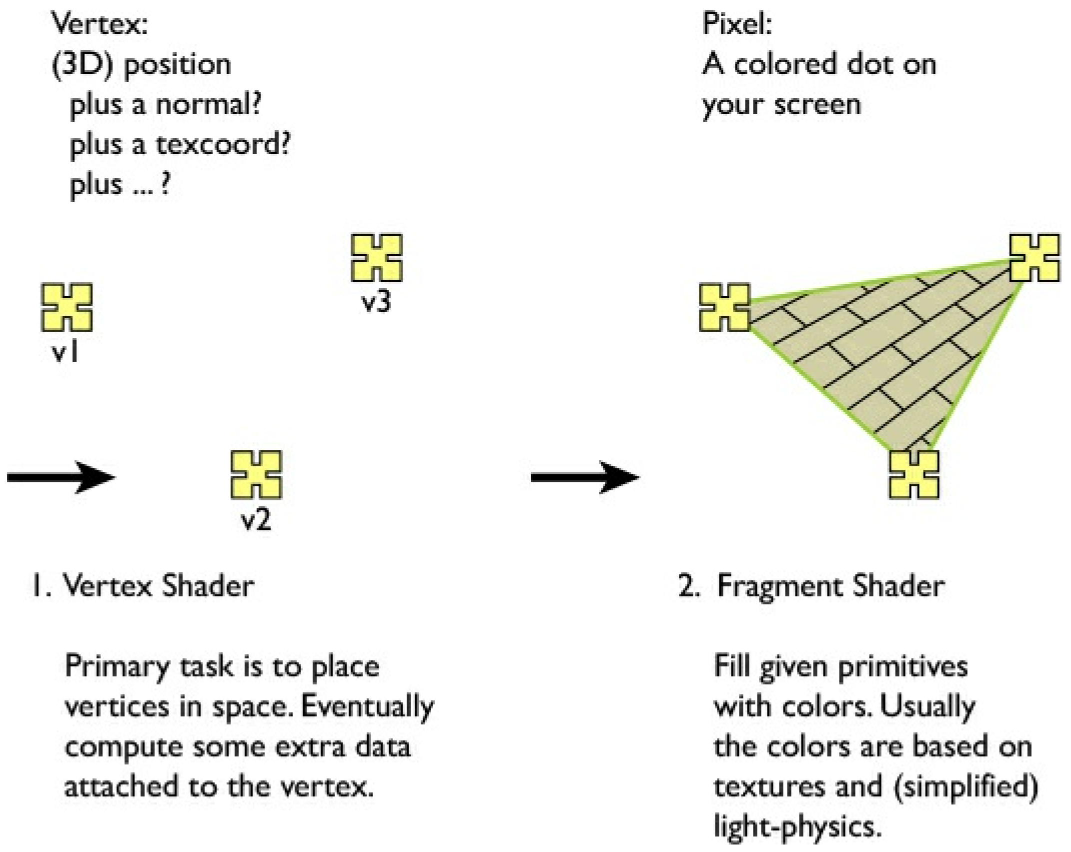
For more information about how this is done, see Chapter 8, "2D Drawing."

Vertex Shading

In order to perform rendering, your application must create a pair of functions, each with its own distinct responsibility. The vertex and fragment shaders are two parts of the same program; they are not two separate things.

The vertex shader is responsible for calculating the position of each vertex (see Figure 2.2). It determines whether the vertex is visible within the frame. At the very least, the vertex shader will receive positional data for each vertex. It may also receive other information, such as texture coordinates. It can receive this data either per vertex or per object primitive. Object primitives are used in instanced drawing.

Figure 2.2. *The job of each shader type in the graphics rendering pipeline*


Vertex:
(3D) position
plus a normal?
plus a texcoord?
plus ... ?
Pixel:
A colored dot on
your screen
v1
v3
v2
1. Vertex Shader
Primary task is to place vertices in space. Eventually compute some extra data attached to the vertex.
2. Fragment Shader
Fill given primitives with colors. Usually the colors are based on textures and (simplified) light-physics.

Graphics functions, like vertex and fragment functions, have an argument list that enumerates the resources they operate on. These arguments are bound to buffer and texture objects created by the application through an abstraction called the *argument table*. The arguments are passed into the shader, so when you write your shader application, your passed-in arguments must correlate to arguments present in the argument table.

Vertex shaders receive data that has been prepared for the GPU. These vertices are set up in a stream that is steadily fed to the shader program, where each vertex is processed one at a time. The shader program then processes that data and passes it along to be assembled into primitives. The vertex shader gets called for every single vertex you have referenced in a draw call, so one way to optimize your programs is to use models with as few vertices as possible.

Shaders are small programs written in the Metal Shading Language (MSL). MSL is based on a subset of C++14. It includes some standard data types, such as ints and floats, but it also has some specialized built-in data types and functions, such as dot, which are useful for matrix operations.

Metal has a number of built-in classes to determine how vertex data is stored in memory and how it is mapped to arguments for a vertex shader function.

Clipping

In Metal, you need to know two conceptual worlds: world space and camera space.

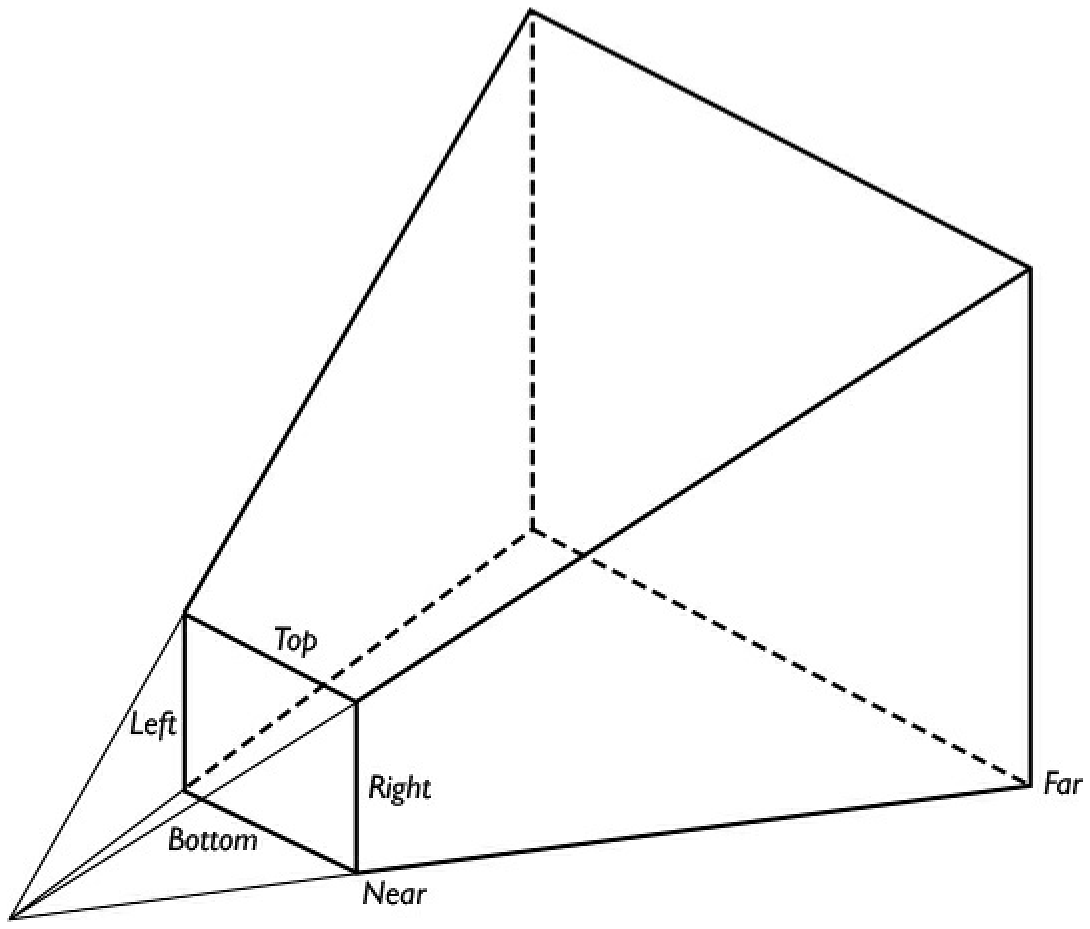
Think about when you watch a movie. You see actors and furniture, but these are not the only things in that space. Outside of the camera range are lights, a studio audience, and a film crew.

It doesn’t make sense for the GPU to render any shapes that don’t appear on the screen. It’s possible to create models and shapes that exist outside of the frame of view. These objects may move in and out of the frame of view.

One optimization that your Metal application will make is to determine whether a shape appears within the frame of view. It will also determine whether a shape is partially obstructed and only partially appears on the screen.

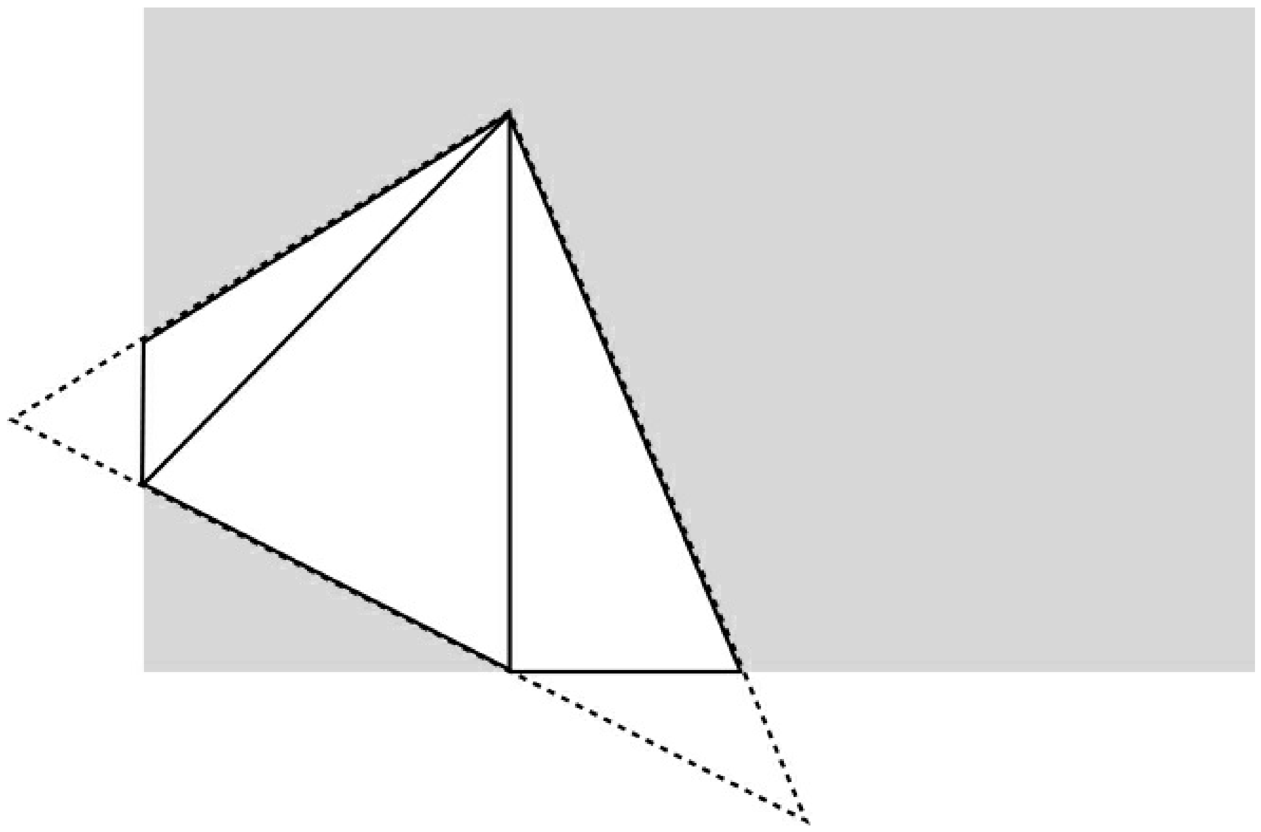
If a primitive is not present at all within the camera space, the primitive is culled. As shown in Figure 2.3, for example, it’s common to cull the face of the vertex that is not facing the camera if it will never be seen. It is not rendered at all, and the program moves on.

Figure 2.3. *The “stage” upon which visibility is determined*



However, if the primitive is partially visible, the primitive is clipped. The triangle is modified to a polygon that conforms to the clip space, and the polygon is broken down into new triangles, as shown in Figure 2.4.

Figure 2.4. *Anything that doesn't appear on the screen is culled. This process sometimes results in retriangulation.*



After the program determines what is and what is not visible, the data is passed to the rasterizer for primitive assembly.

Primitive Assembly

After the vertex shader is done processing the vertex data, that data needs to be assembled into primitives.

Metal supports three basic types of primitives:

• **Points**: Consist of one vertex

• **Lines**: Consist of two vertices

• **Triangles**: Composed of three vertices

In your draw call, you tell the GPU what type of primitive you want to draw. It creates sets of these vertices and constructs them into each primitive type. These shapes are translated into a series of pixels that need to be colored and output to the screen.

This set of pixels is then rasterized.

Rasterization

Rendering revolves around modeling how images appear on a screen. It involves more than just the shapes and the primitives. It also has to do with modeling how the light is scattered around the scene and how it interacts with the objects in the scene.

Traditionally, there are two ways of rendering this data to the screen: ray tracing and rasterization. These methods have similar methods, but they approach the problem from the opposite direction.

Ray tracing scans each pixel in the frame until it intersects with something. The color of whatever object the ray intersects is the color that pixel will be recorded as. If primitives are behind other objects, they won't be rendered on the screen; the ray will stop and return a color before it can reach the obstructed object.

Rasterization takes the opposite approach. Like those old-school overhead projectors teachers used in the 1990s, rasterization projects the primitives to the screen instead of scanning the object from behind. It then loops over the pixels and checks to see whether an object is present in the pixel. If it is, the pixel is filled with the object's color.

Fragment Shading

The fragment shader is responsible for determining the color of every pixel in the frame. Every pixel is a combination of red, green, and blue. Most colors that humans can see can be represented as some combination of red, green, and blue. If each color is at full intensity, the color seen on the screen will be white. If each color has no intensity, the color will be black.

There is a final value, the *alpha* value, which represents transparency. If your object is completely opaque, the alpha value is 1. If your object is a balloon that you can partially see through, the alpha value would be something like 0.8. If you had a completely transparent object, the alpha value would be 0.

If there are no lighting or filters or other effects applied to the object, then the fragment shader requires a simple pass through program that receives and returns the color passed to the shader by the rasterizer.

If, as is more common, you want to add lighting or other effects, the fragment shader calculates how to determine the color of the pixel. Additionally, if you are doing texture mapping to an object, the fragment shader manages that task as well.

Pretend you have a tennis ball that is uniformly green. If you shine a light on the tennis ball, the light introduces gradient shades of the original green. To accurately model the effect of the light, the fragment shader calculates how far away the pixel is from the light source, how intense the light is, and how shiny the surface is. These calculations enable the program to model light surface interaction.

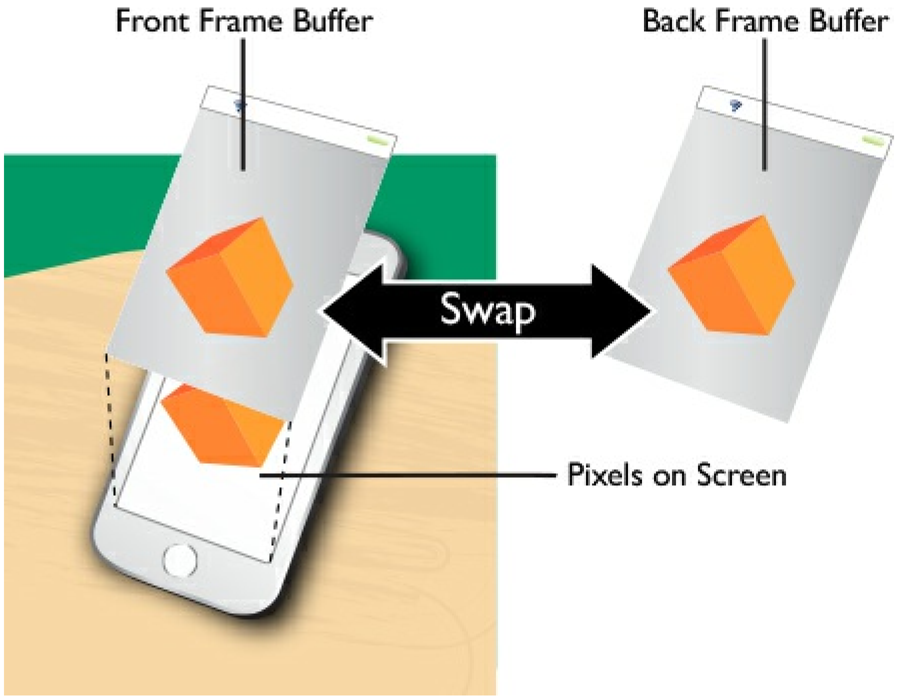
The fragment shader returns a data representation for the red, green, blue, and alpha values of each pixel and passes it along the pipeline. The fragment shader is the last stop for modifying image data before it is sent to the frame buffer to output.

Raster Output

Before the image can be output to the screen, it first goes through the frame buffer. No matter how fast you optimize your rendering calls, it will still take a certain amount of time for the GPU to draw the data for rasterization. If you were to draw directly to the screen, there would be flickering and the quality would be poor, which you clearly don't want.

Rather than trying to draw directly to the screen, the image is composed and stored in the frame buffer. There are at least two frame buffers at any given point in time. One is actively being presented to the screen while the other is being drawn to, as shown in Figure 2.5.

Figure 2.5. *The frame buffers swap out. While one is being drawn to, the other is being presented.*



The frame buffers tag team so that one is always being presented to the screen. This ensures a seamless user experience. It also frees up the GPU to continue rendering while the previous frame is being shown to the user. If you weren't double- or triple-buffering, you wouldn't actually be able to do any work on the GPU, since you wouldn't have anywhere to put the rendered data.

Summary

The way we humans conceptualize graphics is purely visual. A computer, however, must numerically represent images. For a computer to understand what the image is supposed to look like, we must format information in a way that a computer can understand.

3D images are broken into triangles. These triangles are described using vertices. These vertices are written to a vertex buffer and fed to a vertex shader. Then, the computer determines whether the triangle appears on the screen. It culls any objects that don't appear. These objects are then refined by the fragment shader, where effects can be applied. Lastly, the data describing each pixel is sent to a frame buffer, where the image is drawn before it is presented to the screen.

3. Your First Metal Application (Hello, Triangle!)

Even the largest avalanche is triggered by small things.

—Vernor Vinge

To get a good feel for how Metal works, we visit an old chestnut that acts as the Hello, World! of graphics programming: rendering a triangle to the screen. This chapter is a high-level overview that touches on the various players in the Metal ecosystem. A more in-depth exploration of drawing 2D and 3D graphics is presented later in this book. This chapter helps you build a solid foundation upon which to build more useful and complex programs.

Creating a Metal Application in Xcode (without Using a Template)

This chapter begins by setting up Metal without using the built-in Apple template. You might be wondering why you would want to reinvent the wheel and not just use what Apple gives you.

First, setting up the pieces yourself gives you a good feel for each of the moving parts in Metal. You are responsible for managing memory in a way that you are not used to, so it's good to become intimately familiar with all of the objects you will be configuring in the future. Second, Apple's templates and sample code tend to include some scaffolding that you don't necessarily need in your projects. When you're first starting out, it can be hard to differentiate between what is absolutely necessary and what is superfluous. Starting fresh lets you know what the bare-bones necessary objects are. With that, let's get started.

Creating a Project in Xcode

Xcode is Apple's integrated design environment (IDE) in which programmers can develop applications for Apple devices. It's a free application that can be downloaded from the Mac App Store. Yes, that means that Xcode is a Mac-only IDE. To develop iOS applications, you must own (or have access to) both a Mac and an iOS device that runs Metal. Xcode is a large and complex application. Going into depth on Xcode is beyond the scope of this book. There are many resources available to you if you want or need to become more familiar with it. If you're an experienced iOS developer, you can skip ahead to the next section.

After the application launches, you are given a few options to either create a new project or open one. Choose *Create a new project*. This opens another window where you are given a selection of templates. The Metal template is neatly hidden within the Game template. Since we're not planning to generate a Metal project from a template, select *Single View Application*. This is usually a good choice as a starter template for any project you have in the future.

After you choose the single view application, you are asked to name your application. Make sure that you have a company identifier. If you're not an Apple developer and are simply exploring, just enter your name. You don't need to choose Core Data or any of the other options.

Xcode Simulator and Metal

Xcode comes with a Simulator that allows you to test your applications without having to build them to a device. If you look in the upper-left corner, you see a Play button. It launches the Simulator and allows you to see how your code works.

Currently, the Simulator does not work with Metal. Therefore, to check your application, you need to build it on a device. To do that, you need to have an Apple Developer Account (register at https://developer.apple.com). Unless you plan to sell an app on the App Store, sign up for a free account.

The first time you try to build to your phone, you are prompted to authorize your developer account on the phone. Additionally, because Xcode can't build a Metal project in the Simulator, if you have a Simulator set on the Scheme Editor next to the Run button, you will get some errors that make no sense. One error you will commonly see related to this issue says, "CAMetalLayer does not exist." If you see strange errors in code that you haven't touched, check to make sure the build target is not the Simulator.

Your new project has numerous files in it. You can ignore most of these for now. You'll be working primarily in ViewController.swift. At the top of the view controller, you need to import Metal. You should do this directly under the import of UIKit:

```
import Metal
import UIKit
```

Xcode requires you to import every framework that you are going to refer to in your project. This is not the only framework you will import, but it will suffice for now.

Creating a MTLDevice

The foundation upon which Metal is built is the GPU. To interact with the GPU, you need to create a software interface for it in your code. This interface is the MTLDevice protocol. An object that conforms to the MTLDevice protocol represents a GPU in your Mac or iOS device. Apple does not want you to try to subclass the MTLDevice because it's so close to the metal that overriding built-in functions on this protocol could cause serious problems with your application. At the top of the view controller class, you need to create an instance of MTLDevice:

```
var device:MTLDevice! = nil
```

This device cannot be initialized when you create the property at the top of the class. MTLDevice has one—and only one—safe initialization method. This method should be called in the function viewDidLoad(). Any settings you need to make when the view loads but before the user can interact with the application should be placed in viewDidLoad().

```
device = MTLCreateSystemDefaultDevice()
```

That's all you need to do to set up a software interface with the GPU. The rest of your application will be referring back to this device object.

Creating a CAMetalLayer

Metal can render images to the screen, but it is not equipped to allow you to interact with them. In iOS, interactions between the screen and the user are controlled by Core Animation layers.

Both Metal and OpenGL before it have special Core Animation layers that you need to create and attach to the view. First, you need to import another framework:

```
import QuartzCore
```

Next, you need to create an instance of your special Core Animation–backed Metal layer:

```
var metalLayer: CAMetalLayer
```

Again, as you did when initializing your MTLDevice, you will initialize your CAMetalLayer in viewDidLoad():

```
metalLayer = CAMetalLayer()
metalLayer.device = device
metalLayer.pixelFormat = .bgra8Unorm
metalLayer.framebufferOnly = true
metalLayer.frame = view.layer.frame
view.layer.addSublayer(metalLayer)
```

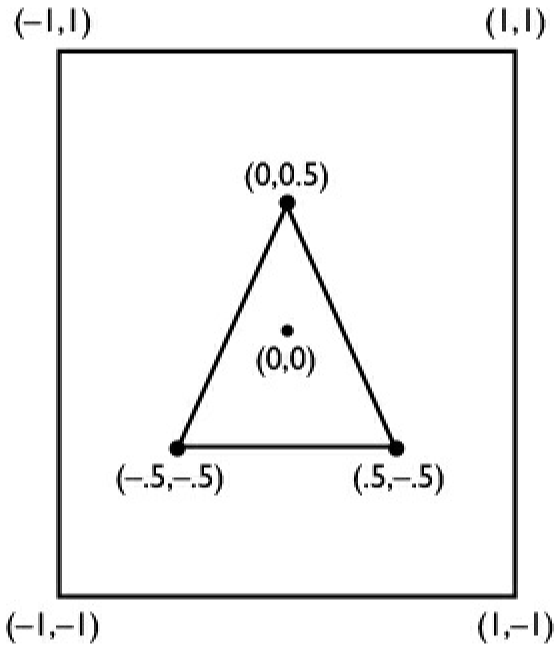
First, you're initializing the CAMetalLayer. Then, you tell it that the layer it's going to be interfacing with is the MTLDevice you created earlier. You're setting a few other properties on the layer. These defaults are fine, and if you want to look into them further on your own, go ahead. Lastly, the view needs to associate this layer with itself, so you need to add it as a sublayer once all of its properties are set.

Creating a Vertex Buffer

All images in Metal are composed of primitive shapes. The most common primitive shape in Metal is a triangle. All shapes can be composed of triangles. You can't just feed a bunch of triangles into a software program. You need some way to represent these triangles in code. This is done by using *vertices*. A vertex is a data structure that describes attributes in graphics programming. These can be color or texture position data. In this simple example, the vertices describe 2D positions in space at the tips of the triangles.

Metal utilizes a *normalized coordinate system*. Rather than having a discrete unit of measure such as an inch or a Kropog, the coordinates are expressed in terms of proportion to a whole. If you look at an iPhone, it is taller than it is wide. Even though it's height is greater than its width, it's considered to be one unit across by one unit high. The proportions of those units represent different amounts on differently shaped screens. Metal's normalized coordinate system is two units by two units by one unit. This means that the upper-left front corner's coordinate is (1, 1, 1). The lower-right corner's coordinate is (−1, −1, 0). The center of your coordinate system is represented by (0, 0, 0.5) as shown in Figure 3.1.

Figure 3.1. *Metal's normalized coordinate system with a triangle mapped out*



Metal uses the normalized coordinated system no matter what the aspect ratio of your screen is. This holds true whether you're looking at an iPhone, which is twice as tall as it is wide, or an Apple Watch, which is perfectly square.

You need to create an array to hold all the positional data necessary to draw a triangle. The triangle you will create is about half the size of the screen, so your triangle data will look something like this:

```
let vertexData:[Float] = [
    0.0, 0.5, 0.0,
    -0.5, -0.5, 0.0,
    0.5, -0.5, 0.0]
```

The GPU can't access this data in its present form. The data must be encoded into a buffer, so you need to create a *vertex buffer* to hold it:

```
var vertexBuffer: MTLBuffer! = nil
```

As you did for the instance of MTLDevice, you must set the vertex buffer's data in the viewDidLoad() method:

```
let dataSize = vertexData.count *
    MemoryLayout.size(ofValue: vertexData[0])
vertexBuffer = device.makeBuffer(bytes: vertexData,
                                 length: dataSize,
                                options: .storageModePrivate)
```

To create a buffer large enough to hold the data, you need to determine the size of the vertex array. Next, based on the array size, you specify how many bytes you need and point the buffer at the vertex data. This vertex buffer is primed with data ready to be processed by the GPU. This job is done by shaders, which we cover next.

A First Look at Shaders

We've spent a lot of time talking about preparing data for the GPU. The actual code that you write that instructs the GPU how to process data comes in the form of *shaders*.

Shader functions in graphics programs have two components: *vertex* and *fragment*. The vertex shader receives the buffer data and return information about each vertex, such as position and color. The fragment shader receives information about the scene and determines the color of each pixel in the scene.

Both shader types are written in the *Metal Shading Language* (MSL), which is based on C++14. The principles of shaders are fairly consistent among the various shading languages, so if you're familiar with another language, such as OpenGL Shading Language, it should be easy to pick up MSL.

You need to create a new file to hold your shaders. You can put multiple shaders in a single file, or you can split them up into different files. It's your choice. As long as the file has the .metal extension and is included in the Compile Sources phase of the target, it will be compiled into the default library, which we discuss shortly.

Create a new file. Choose *Metal* from your template options, and name your new file *Shaders.metal*. Make sure the file is saved to your project directory.

Creating a Vertex Shader

All vertex shaders must begin with the keyword *vertex*. They must also at least return position data for the vertex in the form of a float4 object. Add the following code to the bottom of the shader file:

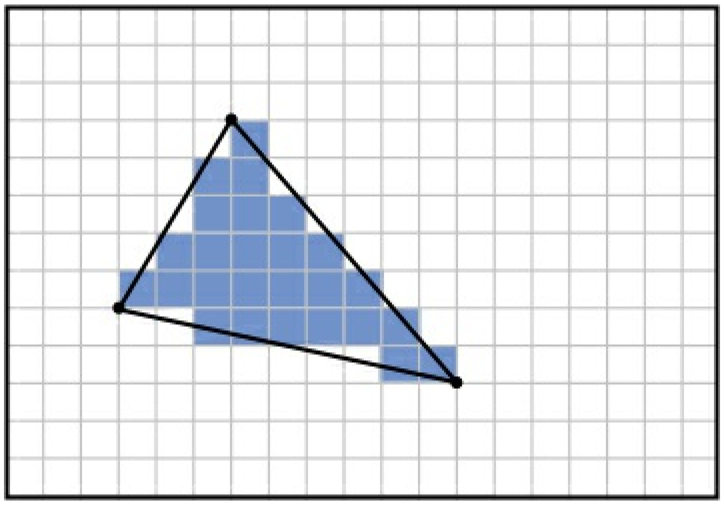
```
vertex float4 basic_vertex(
    const device packed_float3* vertex_array [[ buffer(0) ]],
    unsigned int vid [[ vertex_id ]]) {
        return float4(vertex_array[vid], 1.0);
}
```

This vertex shader receives two parameters. The first parameter is the position of each vertex. The [[buffer(0)]] code specifies to the vertex shader to pull its data from the first vertex buffer you sent to the shader. Because you've created only one vertex buffer, it's easy to figure out which one comes first. The second parameter is the index of the vertex within the vertex array.

Creating a Fragment Shader

The fragment shader is responsible for calculating the color of each pixel on the screen, as shown in Figure 3.2. It calculates the color according to the color data it receives from the vertices. If one vertex is red and another one is green, each pixel between those vertices will be colored on a gradient. The amount of red and green is calculated on the basis of how close or far away each vertex is.

Figure 3.2. *The work for which the vertex and fragment shaders are responsible*



Each fragment shader program begins with the keyword *fragment*. Add this code to the bottom of your shader file:

```
fragment half4 basic_fragment() {
    return half4(1.0);
}
```

At a minimum, each fragment shader must return the color of the pixel it is shading. This is represented by the half4 type. This shader simply sets the color to white.

This is just a small taste of what shaders can do. Although we provide further information about them throughout this book, a comprehensive discussion of shaders can and does fill several books but is beyond the scope of this book.

Libraries, Functions, and Pipeline States

Shaders are examples of MTLFunction objects. MTLFunction objects are written in MSL and executed on the GPU. There are three different types of MTLFunction: vertex, fragment, and kernel. We already saw vertex and fragment functions.

MTLFunctions are collected in MTLLibrary objects. A MTLLibrary can have one or more MTLFunction in it. These functions can be created from a compiled binary library that is created during the app-building process or from a text string that is compiled at runtime.

You need to create a library to hold your shader programs:

```
let defaultLibrary = device.newDefaultLibrary()
    let fragmentProgram = defaultLibrary!.makeFunction(name: "basic_fragment")
    let vertexProgram = defaultLibrary!.makeFunction(name: "basic_vertex")
```

In this snippet of code, you’re creating a new default MTLLibrary to hold your shader program. You’re then creating two MTLFunctions to hold your fragment and vertex shaders. Those function names should look familiar, as they are the shaders you wrote in the previous section. Since those functions are in a file with the extension .metal, the library will know where to look for them.

Now that you have a default library to contain your shaders, you can add them to your render pipeline. The render pipeline holds all of the directions necessary to render your triangle to the screen. First, you need to set up an object to hold all of the state the render pipeline needs to refer to. Add this beneath all of your other properties:

```
var pipelineState: MTLRenderPipelineState! = nil
```

Next, add the following code to the end of viewDidLoad():

```
let pipelineStateDescriptor = MTLRenderPipelineDescriptor()
pipelineStateDescriptor.vertexFunction = vertexProgram
pipelineStateDescriptor.fragmentFunction = fragmentProgram
pipelineStateDescriptor.colorAttachments[0].pixelFormat = .bgra8Unorm

do {
try pipelineState = device.makeRenderPipelineState(descriptor:
    pipelineStateDescriptor)
} catch let error {
    print("Failed to create pipeline state, error \(error)")
}
```

With this code, you’re first initializing your render pipeline descriptor. Next, you’re adding the vertex and fragment programs you created when you created your library. Since the pipeline state is fallible, you need to do error handling to ensure that you’re initializing a valid pipeline state. This is really important for Metal. One of the optimizations available with Metal is prevalidated render pipeline state. If your pipeline state is not set up properly, you want this to fail immediately and not after you start sending data to the GPU.

Introducing Render Passes

Now that the render pipeline is set up, it’s time to look at setting up render passes. Render passes are the mechanism for composing a scene. Multiple passes can be made to add additional objects and details. For example, you can render different objects or add lighting and highlighting effects in separate passes.

The first thing you need to create is your render pass descriptor. It makes sense to put this code in a separate method that can be run every time a new frame needs to be rendered. Create a new method in the view class called render():

```
func render() {
    let renderPassDescriptor = MTLRenderPassDescriptor()
    guard let drawable = metalLayer.nextDrawable() else {return}
    renderPassDescriptor.colorAttachments[0].texture = drawable.texture
    renderPassDescriptor.colorAttachments[0].loadAction = .clear
    renderPassDescriptor.colorAttachments[0].clearColor =
        MTLClearColor(red: 221.0/255.0, green: 160.0/255.0,
        blue: 221.0/255.0, alpha: 1.0)
}
```

The render() method sets up the render pass descriptor. The descriptor contains a collection of attachments that describe color, depth, and stencil data. In our simple program, we’re only setting up color data. Our descriptor is targeting the first color attachment in the array. It is setting the render pass to clear whatever color was there before and to fill the screen with a nice plum color.

Command Queue and Command Encoder

Now that you have your render pass descriptor set up, you need a way to submit work to the GPU for execution. This is done with the command queue. The command queue is the object responsible for scheduling time with the GPU to complete work.

Add this property to the top of the class with your other properties:

```
var commandQueue: MTLCommandQueue! = nil
```

Again, initialize the command queue in viewDidLoad():

```
commandQueue = device.makeCommandQueue()
```

The command queue is an expensive object to create, so you want to create only one and reuse it. The command queue doesn't take a render pass directly. Instead, it takes command buffer objects. Command buffer objects are cheap to create. They are used and disposed of when you're done with them. Add this line to the bottom of your render() method:

```
let commandBuffer = commandQueue.makeCommandBuffer()
```

Issuing Draw Calls

The command buffer needs to encode render commands in order to know what work to send to the GPU. This is where all of the setup you've been doing throughout this project comes together. It takes all of the pipeline state you've set and all of the vertex data you've processed and encodes the rendering pass. Complete the render() function by adding the following code to the end of the method:

```
let renderEncoder = commandBuffer.makeRenderCommandEncoder(descriptor: renderPassDescriptor)
renderEncoder.setRenderPipelineState(pipelineState)
renderEncoder.setVertexBuffer(vertexBuffer, offset: 0, at: 0)
renderEncoder.drawPrimitives(type: .triangle, vertexStart: 0, vertexCount: 3)
renderEncoder.endEncoding()

commandBuffer.present(drawable)
commandBuffer.commit()
```

The renderEncoder object is encoded with the pipeline state and the vertex buffer. This is where your draw call is as well. In drawPrimitives(), you're telling the GPU what type of primitive it needs to draw so that it knows how many vertices constitute a shape. This is explained in more detail in Chapter 8, "2D Drawing."

The render() method is now complete. You need to call it from somewhere, so set up a game loop:

```
func gameloop() {
    autoreleasepool {
        self.render()
    }
}
```

All of your data is primed and ready to be sent to the GPU. The last step in this process is to actually present your triangle to the screen.

Presenting Metal Content to the Screen

Every time your rendering passes create a frame, you need the view controller class to know it needs to refresh the screen. This is done using a CADisplayLink. CADisplayLink is a special timer that is used to coordinate draw calls with the refresh rate of the screen. Create your timer property at the top of the view controller class:

```
var timer: CADisplayLink! = nil
```

At the bottom of viewDidLoad(), create the timer. The timer calls the game loop every time the screen refreshes, which signals the GPU that it has work to do:

```
timer = CADisplayLink(target: self, selector: #selector(ViewController.gameloop))
timer.add(to: RunLoop.main, forMode: RunLoopMode.defaultRunLoopMode)
```

Every time the screen refreshes, a few things happen. The screen is cleared and replaced with a purple background. Then a draw call kicks off, telling the GPU to take the prepared vertex data and construct your triangle. This is then sent to the fragment shader, which colors each pixel within the triangle white. The render pass takes this data and draws it to the screen.

Introducing MetalKit Features and MTKView

Apple, in its infinite wisdom, realized that it isn’t really optimal to do this vast amount of setup every time you use Metal. Some of the steps you took to set up Metal are basically the same for all applications. Because it makes no sense to have to write that boilerplate code constantly, a new framework was introduced in iOS 9.

MetalKit was designed to reduce efforts in three key areas:

• Texture loading

• Model handling

• View management

Texture loading and model handling are covered in more depth later in this book. Here, we focus on view management. You had to do a lot of setup to create the CAMetalLayer and attach it to the view. You also had to set up a render loop and connect it to a CADisplayLink. These tasks must be done for every Metal application you create, and it doesn’t make sense to rewrite that boilerplate for every application. Therefore, the boilerplate code is taken care of by MTKView and MTKViewDelegate.

MTKView negates the need to set up a CAMetalLayer or a CADisplayLink. MTKView has three different modes you can use to draw to the screen. The default mode draws according to an internal timer and is similar to what you set up in your Hello, Triangle! application. The second mode uses notifications whereby the view redraws only when it is told that something has changed and requires redrawing. The final mode uses a MTKView method, draw(). The view will draw only when draw() is called. This mode is explored in more depth in Chapter 8.

You can choose to either subclass MTKView or make the view conform to MTKViewDelegate. MTKViewDelegate also has a draw() method that can be implemented in lieu of MTKView’s draw method. Do whichever you prefer; the choice is up to you.

Summary

Preparing data and programs to be processed by the GPU requires many steps. You need to implement a special Core Animation layer to allow users to interact with Metal-backed views. All objects on the screen are composed of primitives. These primitives are represented by arrays of vertex data. Vertex data is prepared and sent to the GPU using vertex buffers. Programs that run on the GPU are known as shaders and are written in MSL. Work is scheduled by the command queue in the form of command buffers. These buffers are encoded with commands and state information. Those kick off the draw calls, which signal the GPU to do scheduled work. After the work is completed, it is presented to the screen.

II: Rendering and Graphics

4. Essential Mathematics for Graphics

The difference between the poet and the mathematician is that the poet tries to get his head into the heavens while the mathematician tries to get the heavens into his head.

—G.K. Chesterton

In graphics programming, there are two components you need to understand: the programming framework and what problems it is capable of solving. The language of graphics is mathematics, which is the purest language we have to describe the world around us. Everything we see and everything that moves can be expressed mathematically, and the equations can be easily expressed in code. This chapter covers some important mathematical concepts that enable the framework to turn lifeless-looking code into lively, detailed graphics.

Language of Mathematics

An algorithm is a process or set of rules to be followed in calculations or other problem-solving operations. The meat of an algorithm is the idea or process it describes. This description can be expressed in many “languages”—English, JavaScript, or linear algebra, for example. The ability to mentally articulate how to solve a problem and what a process does is an invaluable skill that will help you on your journey as a graphics programmer.

One advantage of expressing things mathematically is that mathematical algorithms are platform agnostic. There are not, as of this writing, a multitude of books on Metal programming, but the concepts around graphics math have been around for hundreds of years. Being able to “speak” this language allows you to use these resources to do incredible effects—you’re not constrained to materials that are written in the Metal language.

Graphics math is a vast topic. It includes aspects of linear algebra, vector calculus, and statistics. This information could—and does—fill several books. This chapter serves as a high-level overview of the concepts you should be familiar with and how they fit in this process. There are many wonderful resources out there for shader effects, but many of them are either platform specific or express their ideas mathematically. Having a basic understanding of these concepts allows you to decode those resources even though they may have been intended for a piece of software that no longer exists.

Coordinate Spaces and Moving among Them

Graphics programming boils down to figuring out how to express how an object appears and moves around in space. Before you can begin to move an object in space, you need to orient where it is and how it moves around—you need a coordinate system. A coordinate system requires

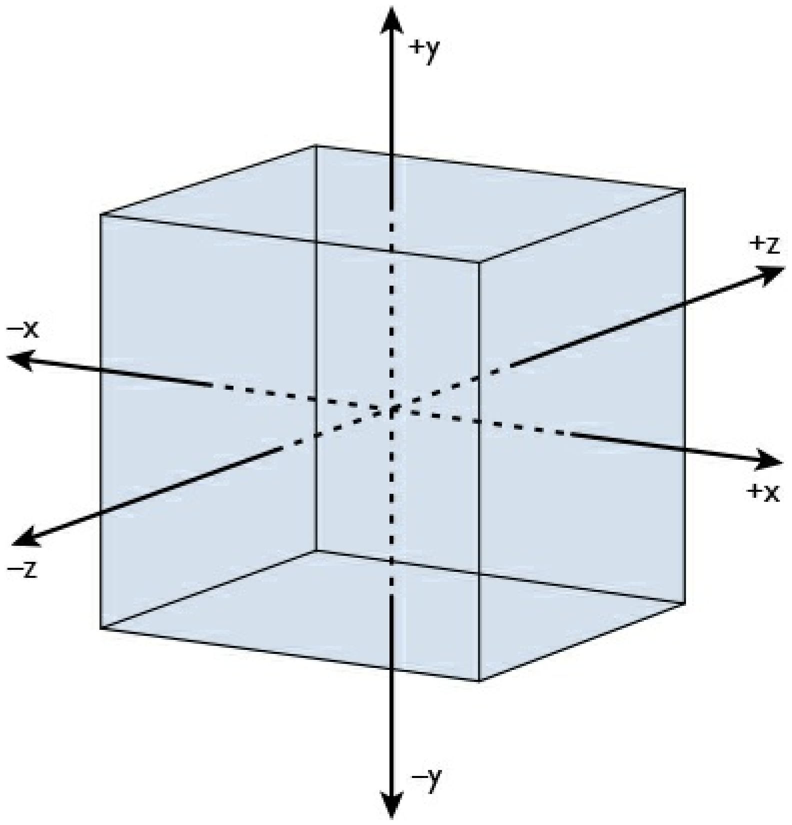
• A point of origin

• A consistent unit of measurement

• A convention to orient the positive and negative directions

The earliest coordinate system most of us remember is the Cartesian coordinate system. The 2D Cartesian system is represented by two perpendicular lines, one horizontal (x-axis) and one vertical (y-axis). Each line contains equally spaced ticks measured in the same unit of length (when you learned this system, you probably used graph paper, with each square representing equal length). The place where the x- and y-axes intersect is the *origin*. Each tick on the horizontal line to the right of the origin and on the vertical line above the origin is considered positive. Each tick on the horizontal line to the left of the origin and on the vertical line below the origin is considered negative. In the 3D Cartesian coordinate system, a third line, the z-axis, is added, perpendicular to the x- and y-axes (see Figure 4.1). Objects moving up and to the right are moving in a positive direction, and objects moving down and to the left are moving in a negative direction.

Figure 4.1. *Metal's 3D coordinate space*



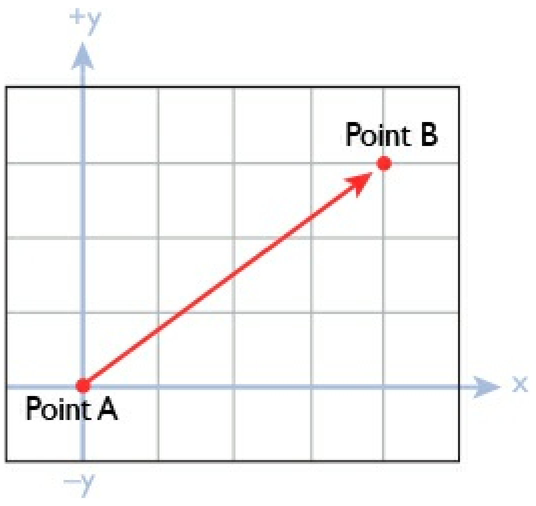
Even though this system is familiar, it's not the only coordinate system. The axis pointing up could be the z-axis instead of the y-axis, for example. Or up on the y-axis could be negative instead of positive. Another important aspect of coordinate systems, especially applicable later in this chapter, is to know whether you are dealing with a left- or a right-handed coordinate system. This has to do with positive and negative orientation in relation to the axis of rotation. If the coordinate system is right handed, then the positive axis of rotation moves counterclockwise. If the coordinate system is left handed, the positive axis of rotation is clockwise.

It's important to know the rules governing your coordinate system before you begin working with it. Metal actually has two coordinate systems. One applies to the 3D world space and the other applies to the rasterized 2D projection space. In Metal's 3D coordinate system, both the x- and y-axes run from −1 to 1, making those coordinates two units long. The z-axis is different and runs only from 0 to 1. Metal's 2D coordinate system is the same as the one used by Core Graphics but opposite the OpenGL coordinate system. Its origin is in the upper-left corner of the projection space. The x value is the distance across the space that an object appears, and the y value is the distance down the space that the object appears. If you are not familiar with the coordinate system you will be working with, you won't be able to accurately describe how you want your objects to appear in space.

Points, Vectors, and Vector Operations

The next concept we cover is the *point*. A point is a coordinate in space. Let's use a point as an example: (3, 4, 0). The first number is the point's *x* coordinate, the second number is the *y* coordinate, and the third is the *z* coordinate. Point B shown in Figure 4.2 describes a location that is three units to the right and four units up from the origin.

Figure 4.2. *A vector between the origin at Point A and another location at Point B*



Vectors are expressed similarly to points but are different. A vector is defined as having both a magnitude and a direction. Let's take our previous point in Figure 4.2 but change it to a vector: (3, 4, 0). It's still expressed the same way, but there is a difference. Instead of describing a specific point in space, (3, 4, 0) describes the movement between one point and another. For this, you use the Pythagorean theorem. The first point in this vector is five units away from the second point in the vector because the square root of three squared plus four squared is four. This is covered in more detail later in this chapter. Because a square root can only be positive as it is the addition of two squared numbers, the vector is moving in a positive direction three units over and four units up. This vector can originate anywhere; it doesn't matter. All it is describing is movement from one point to another.

Both vectors and points are contained in a float4 vector type. At first, this doesn't seem to make sense, because points and vectors have only three coordinates. So what is the fourth component for? The fourth component differentiates whether the float4 is a point or a vector. Points have the value 1 and vectors have the value 0 as the last value in a float4.

Several operations can be performed on both points and vectors. Points can be subtracted to create vectors. Points cannot be added together. Vectors can be multiplied by scalars to scale them up and down. In the previous paragraph, you learned that points have a final value of 1. Any legal operation done on points and vectors will result in the final coordinate value equaling either 1 or 0. In this overview, we focus on two special vector operations used extensively in computer graphics: dot product and cross product.

The dot product of two vectors is found by multiplying the x component of each vector and adding it to the product of the y component of each vector.

$$\vec{a} \cdot \vec{b} = a_x \cdot b_x + a_y \cdot b_y = |\vec{a}||\vec{b}|\cos(\theta)$$

The dot product also gives the cosine of an angle between two vectors that both have a length of 1. This simple explanation doesn’t explain what you can do with the dot product or why it’s relevant to your journey as a graphics programmer. At its heart, the dot product is a way of measuring the impact one vector has on another.

A good example of a vector effect is in the video game Mario Kart. In Mario Kart, there are areas of the track that give you a speed boost. If your cart isn’t moving, the speed boost doesn’t do anything because the dot product augments what you’re already doing. The speed boost only helps with forward momentum, so if you’re primarily moving sideways, it won’t help you very much.

Let’s say you get a speed boost of two in the forward direction, but you’re not moving forward. Your forward vector is (0,0). The speed boost vector is (0,2). The dot product of these vectors is (0 * 0) + (0 * 2), which is still zero. However, if you’re moving forward at one unit but sideways at three units, the dot product has no effect on your sideways movement. In this case, the dot product would be (3 * 0) + (1 * 2), which equals 2. It doesn’t matter how fast you are moving sideways—it will never affect the final result because the speed boost targets only one axis of movement.

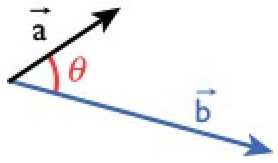
The dot product in graphics programming is used, among other things, to determine how light interacts with an object. For lighting, it finds the cosine of the angle created by the surface normal and the direction the ray of light is coming from. This is explained further as you read through this chapter.

The cross product is related to the dot product, but it is different. The dot product is concerned with the degree of similarity between magnitudes within the same dimension that are perpendicular to one another. It compares x-values to x-values. It doesn’t care what the y-value or the z-value is. The cross product does. The cross product is used to find the axis of rotation between two vectors, as shown in Figure 4.3.

Figure 4.3. *How to calculate the cross product*

$$\text{LENGTH}\left(\vec{a} \times \vec{b}\right) = \sin\theta \bullet \text{LENGTH}\left(\vec{a}\right) \bullet \text{LENGTH}\left(\vec{b}\right)$$

$$\vec{a} \leftrightarrow \left(a_x, a_y, a_z\right)$$
$$\times$$
$$\vec{b} \leftrightarrow \left(b_x, b_y, b_z\right)$$



$$\vec{a} \times \vec{b} = \left(a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x\right)$$

Normalization and Unit Vectors

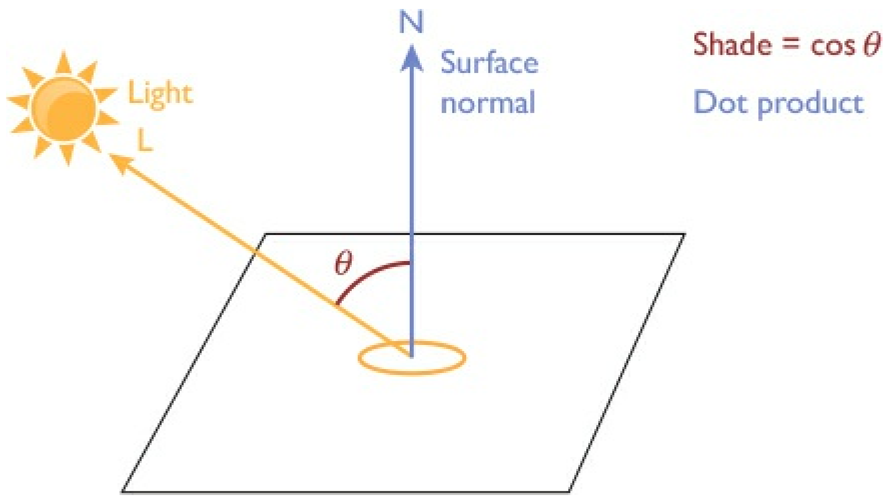
A lot of graphics-related math is easier to do when you deal with percentages. If you think of an iPhone screen as being one unit long by one unit wide with the center of the screen being (0.5, 0.5), you don't have to think about what the screen's pixel resolution is or what iteration of the iPhone you're on. This philosophy applies to many other equations in graphics programming, which is why an important mathematical concept you should be aware of is the process of vector normalization.

Much of graphics programming works under the assumption that you are working with percentages and that you will not have a value larger than 1. If you have a right triangle with unit values of 3, 4, and 5, in order to perform algorithms on this triangle, you need to determine proportionally how these values scale. The hypotenuse has a value of 1, while the other two sides are some percentage of that hypotenuse. To find that ratio, you divide each side by the value of the hypotenuse. So instead of the triangle being (3, 4, 5), the triangle is normalized to (3/5, 4/5, and 1).

Vector normalization is useful for determining direction. One common use of the dot product, for example, is to calculate the cosine of an angle between a light ray and the surface normal, as shown in Figure 4.4. This calculation is done only after the vectors are normalized.

To normalize a vector, you perform the Pythagorean theorem on the components of the vector. That is, you square each component, add them together, and take the square root of that number. In our (3, 4, 0) vector, $3^2 = 9$, $4^2 = 16$, and $0^2 = 0$, so $9 + 16 + 0 = 25$. The square root of 25 is 5, which is the length of the vector. To get the normalized coordinates, you divide each component by the length. This tells you what percentage of the overall vector each component makes up.

Figure 4.4. *Normalizing a vector*



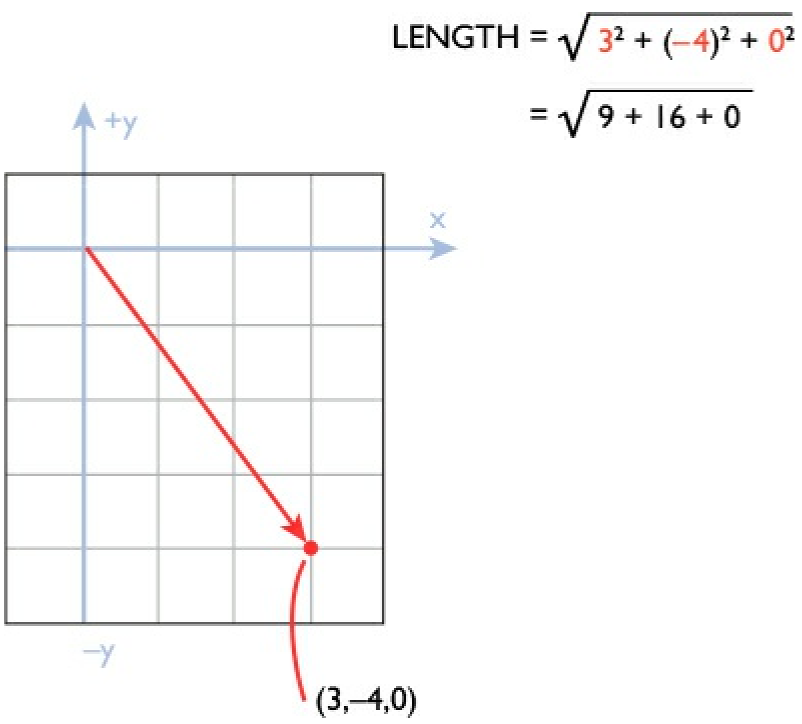
Normalizing a vector doesn't change the shape or the angle of the vector. It simply scales the vector up or down. Picture a 45-45-90 degree right triangle. There are certain rules about that triangle that hold true no matter what. Two of the three sides will always be the same length regardless of whether that length is 10 units or 42 units. Making the hypotenuse one unit and every other side a percentage of that unit makes the hypotenuse easier to do mathematical processes on. A lot of units associated with your coordinate system are arbitrary. The important thing is to make sure that the proportions of the sides and the angles of the triangle are consistent.

Pythagorean Theorem

Many of the previous sections dealing with dot product and normalization rely on orthogonality. Orthogonality deals with the relation of two lines at right angles to one another. If you have a set of lines that are orthogonal—or perpendicular to one another—there is a powerful equation that allows you to calculate a lot of useful measures in graphics programming: the Pythagorean theorem.

The Pythagorean theorem enables you to find the shortest distance between orthogonal directions, as shown in Figure 4.5. It's not really about right triangles; it's about comparing "things" moving at right angles. When we looked at the dot product, we were looking at how combining two vectors would impact their growth in the x and y directions.

Figure 4.5. *The Pythagorean theorem*

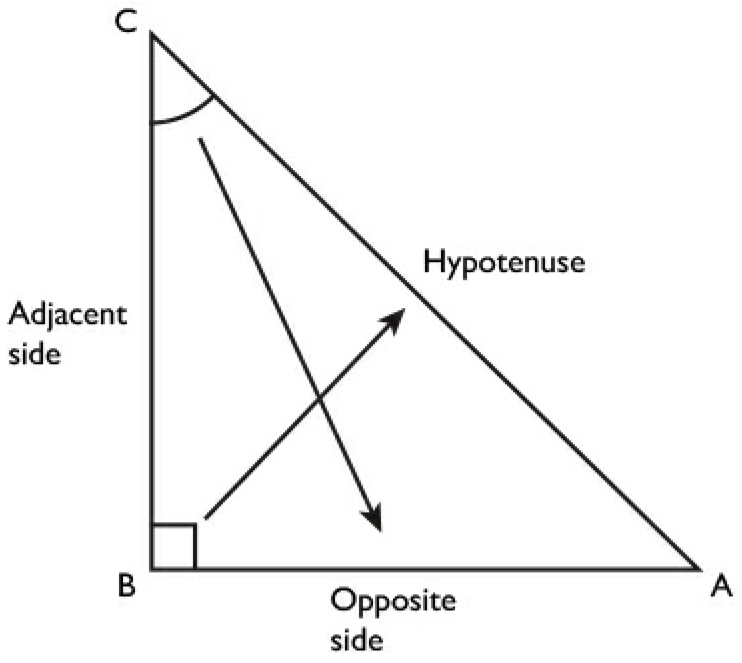


Even though our vectors and not triangles, per se, we can think of them as such for the purposes of being able to use the Pythagorean theorem. Additionally, the shape that we use for our meshes are triangles, which do fit within the Pythagorean theorem. This theorem is the basis for most of the vector operations that you will use to determine how vectors are affected by one another. It's important to grasp that this seemingly simple formula is the key to unlocking much of the power of 3D graphics.

Sine, Cosine, and Tangent

Another concept to dig around in the dredges of your memory from high school trigonometry class is the three triangle operations: sine, cosine, and tangent. If you haven't thought about these concepts since you took the SATs, you may be a little foggy on them. Figure 4.6 illustrates an example of the triangle operations. In this section, we discuss what they mean and how they can help you to do neat things in graphics programming.

Figure 4.6. *Sine, cosine, and tangent operations*



$$\sin c = \frac{\text{opposite}}{\text{hypotenuse}} = \frac{AB}{AC}$$

$$\cos c = \frac{\text{adjacent}}{\text{hypotenuse}} = \frac{BC}{AC}$$

$$\tan c = \frac{\text{opposite}}{\text{adjacent}} = \frac{AB}{BC}$$

$$\cot c = \frac{\text{adjacent}}{\text{opposite}} = \frac{BC}{AB}$$

$$\sec c = \frac{\text{hypotenuse}}{\text{adjacent}} = \frac{AC}{BC}$$

$$\csc c = \frac{\text{hypotenuse}}{\text{opposite}} = \frac{AC}{AB}$$

Every triangle that is not an equilateral triangle or a scalene triangle has one side that is longer than the others. This side is the hypotenuse, which is important because many of the operations performed in graphics programming are based in trigonometric functions involving the hypotenuse. In most of your vector operations, you will be working under the assumption that the vector is the hypotenuse of a triangle.

A mnemonic device may help you remember how to find the sine, cosine, and tangent: SOH CAH TOA. It translates to

• **Sine:** Opposite divided by hypotenuse

• **Cosine:** Adjacent divided by hypotenuse

• **Tangent:** Opposite divided by adjacent

To get the normalized coordinates, you divide each component by the length. Think back to the description of how to normalize the coordinates of a vector: You divide each side by the length of the hypotenuse, which results in the hypotenuse equaling 1. An added bonus is that this operation also gives you both the sine and cosine of the triangle. This knowledge will be especially useful when you get to the section “Transformations: Scale, Translation, Rotation, Projection.”

Matrices and Matrix Operations

Matrices are scary-looking structures, but once you understand them, they are incredibly powerful. They are used extensively in 3D graphics programming and are unavoidable. So let’s dive into matrices and take advantage of their usefulness.

The main type of matrix that you will deal with in 3D graphics programming is a 4×4 matrix. One use of this matrix is known as a *transform matrix*. The transform matrix is used to transform points and vectors from one shape and space to another. There are a few moving parts associated with this process, which are detailed throughout this section.

The foundation of working with matrices rests on the dot product, covered earlier in this chapter. The dot product is concerned with figuring out how much weight is given to like coordinate spaces. Think about an elevator. The elevator goes up and down, but it doesn't matter how fast or high the elevator goes because it doesn't move from side to side. The speed and height of the elevator will never affect its movement back and forth or side to side because the speed and height of the elevator are constrained to the y-axis and never bleed over into the x- or z-axes. Matrices are set up with null values in places where the coordinate is not impacted by changes happening in other coordinate spaces. With that in mind, let's talk about how this applies to 3D graphics programming.

Identity Matrix

The "Hello, World" for matrices is the *identity matrix*. The identity matrix is set up so that if you perform the dot product on a coordinate with the identity matrix, you get the same coordinate. To understand how the matrix affects a vector, it's important to see an example of one that has no side effects. That way you can see how the nulls affect the matrix and where you need to add values in order to change only what you want to change. This means that the identity matrix has a diagonal line of 1s from the upper-left coordinate to the lower-right coordinate. Everything else is a 0.

```
[
 1 0 0 0
 0 1 0 0
 0 0 1 0
 0 0 0 1
]
```

This is your default starting point because it doesn't change or affect anything. Why not give it a try with our (3, 4, 0) coordinate?

```
[
 3
 4
 0
 1
]
```

If you perform a matrix operation on this point using the identity matrix, you get the following:

```
C1 = (3 * 1) + (4 * 0) + (0 * 0) + (1 * 0) = 3
C2 = (3 * 0) + (4 * 1) + (0 * 0) + (1 * 0) = 4
C3 = (3 * 0) + (4 * 0) + (0 * 1) + (1 * 0) = 0
C4 = (3 * 0) + (4 * 0) + (0 * 0) + (1 * 1) = 1
```

You will modify the identity matrix to perform certain tasks, but it's important to know what you need to do to get the same result out as you put in so you can isolate the side effects that occur on each coordinate.

Transformations: Scale, Translation, Rotation, Projection

Now that you have a basic feel for how matrix operations work, it's time to explain how you use them in the context of graphics programming. One large part of graphics programming—and one of the reasons it's so fascinating and powerful—is its ability to implement change. You can change the size, location, spin, and so on, of objects in a scene. This is how you get interactive computer graphics and not just a pretty picture to put on your wall. All of these operations are known as *transformations*, and all of them can be expressed as matrices.

Scale Matrix

Now that you have a grasp on how to set up your matrices, let’s move on to actually changing some of the values and see a change to your matrix. The first matrix we talk about is the scale matrix. The scale matrix isn’t much different from the identity matrix.

The scale matrix has all the same zeros as the identity matrix, but it doesn’t necessarily keep using the ones across the diagonal. You are trying to decide how to scale your coordinate, and you don’t want the default scale value to be 1. Here is the scale matrix:

```
[
 Sx 0  0  0
 0  Sy 0  0
 0  0  Sz 0
 0  0  0  1
]
```

For Sx, Sy, and Sz, you determine how much you want to scale that coordinate by and you enter that value into the matrix. Nothing else changes or is affected. This is the easiest matrix to deal with besides the identity matrix because, in a sense, the identity matrix can be a scale matrix. It just has a scale of 1.

Translation Matrix

The next matrix we talk about is the translation matrix. The translation matrix tweaks the identity matrix somewhat. We already established that the identity matrix returns the same coordinate that you started with. The translation matrix goes a little further and applies a translation value to the coordinate.

The translation matrix looks the same as the identity matrix, but the last column is a little different. The last column applies an amount of change for the *x*, *y*, and *z* coordinates:

```
[
 1 0 0 Tx
 0 1 0 Ty
 0 0 1 Tz
 0 0 0 1
]
```

Let’s look back at our (3,4,0) coordinate. This coordinate would be written out as:

```
[
 3
 4
 0
 1
]
```

Let’s say you want to adjust the x value by 3. You don’t want anything else in the coordinate to change; you just want the *x* value to increase by 3. The translation matrix would look like this:

```
[
 1 0 0 3
 0 1 0 0
 0 0 1 0
 0 0 0 1
]
```

The bottom three rows of the matrix are the same as the identity matrix, so don't worry about them for now. Just look at how this translation affects the first coordinate:

```
(3 * 1) + (4 * 0) + (0 * 0) + (1 * 3) = 6
```

What happens if you try to apply a translation to a vector? Nothing. Vectors don't represent a specific point in space and consequently cannot be affected. Let's make our test coordinate a vector:

```
[
 3
 4
 0
 0
]
```

Now let's apply a crazy translation to it:

```
[
 1 0 0 42
 0 1 0 108
 0 0 1 23
 0 0 0 1
]

C1 = (3 * 1) + (4 * 0) + (0 * 0) + (42 * 0) = 3
C2 = (3 * 0) + (4 * 1) + (0 * 0) + (108 * 0) = 4
C3 = (3 * 0) + (4 * 0) + (0 * 0) + (23 * 0) = 0
C4 = (3 * 0) + (4 * 0) + (0 * 0) + (0 * 0) = 0
```

Because that last value in the vector is 0, it doesn't matter how radically you try to translate a vector—it's not going to change. This is all well and good, but why would we want to do it? What does understanding the translation matrix allow us to do?

Rotation Matrix

Movement is an important part of interactive 3D graphics. Sometimes, movement is unfettered, like a ball, and moves in all directions, but there are many subsets of movement that revolve around rotation. If you are animating a door swinging open, there is a limited range of motion available for that action as the door rotates around the edge where the hinges are. This movement can be calculated in a matrix operation.

If you read the section "Sine, Cosine, and Tangent," you learned that you use sine and cosine to determine angles of a triangle. If you think of the initial position of the vector as one side of a triangle and the desired final position as another, you can take advantage of the triangle operations to figure out how to describe the rotation of the vector in your matrix.

An example of a rotation matrix would look something like this:

```
[
 1 0    0     0
 0 cosθ -sinθ 0
 0 sinθ cosθ  0
 0 0    0     1
]
```

This matrix describes an angle of rotation around the x-axis. Because the x-axis is acting as the hinge on the door, it does not change. You choose the angle you want to rotate the vector by and the new *y* and *z* coordinates are calculated by applying the sine or cosine of the angle of rotation.

Projection Matrix

The last matrix we discuss is an important one that you need to understand, and that is the projection matrix. In graphics programming, there are two spaces that you use: camera space and world space. World space encompasses every object in a scene. Camera space determines how many of these objects are within the field of view. It's possible and common for there to be areas of a scene that are not always visible at any given moment. Think about any first-person shooter game. If your character is moving down a hallway, the areas your character passed are no longer in your field of view and should no longer be rendered.

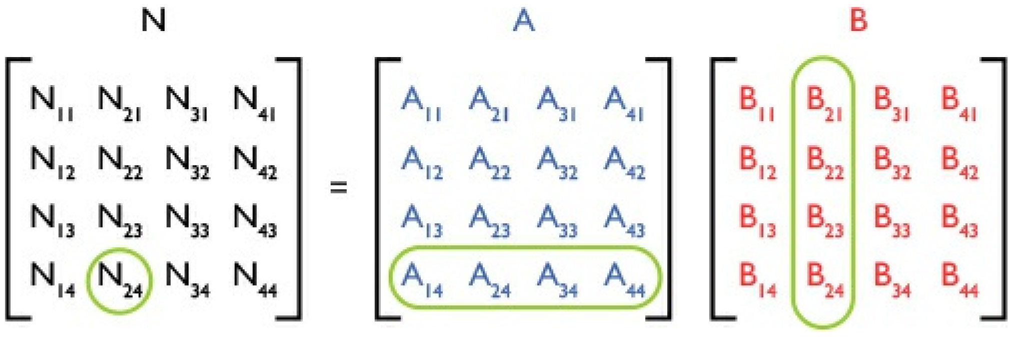
The projection matrix determines the camera space, which is the visible area in a scene, so that the renderer knows to check for objects only in places that will be seen. It also helps determine the clipping area by figuring out if objects are partially off screen and need to be retriangulated. You are making the transition away from thinking about everything in relation to the origin of the model to thinking about the model in relation to the origin of the world space.

Concatenation

So far, we've been talking about applying a matrix to a coordinate. But, is it possible to apply one matrix to another matrix? Absolutely. The process of multiplying two matrices together is known as *concatenation*. Concatenation isn't limited to just two matrices. In fact, in graphics programming, you will chain many matrix operations, and these can be concatenated into a single matrix.

To apply one matrix to another, you take the dot product of the corresponding rows and columns from the two matrices you are multiplying together, as shown in Figure 4.7. For example, if you wanted to find the second value in the first row of the new matrix, you would take the dot product of the first row of the first matrix and the second column of the second matrix.

Figure 4.7. *How matrix concatenation is calculated*



$$\overset{N}{\begin{bmatrix} N_{11} & N_{21} & N_{31} & N_{41} \\ N_{12} & N_{22} & N_{32} & N_{42} \\ N_{13} & N_{23} & N_{33} & N_{43} \\ N_{14} & N_{24} & N_{34} & N_{44} \end{bmatrix}} = \overset{A}{\begin{bmatrix} A_{11} & A_{21} & A_{31} & A_{41} \\ A_{12} & A_{22} & A_{32} & A_{42} \\ A_{13} & A_{23} & A_{33} & A_{43} \\ A_{14} & A_{24} & A_{34} & A_{44} \end{bmatrix}} \overset{B}{\begin{bmatrix} B_{11} & B_{21} & B_{31} & B_{41} \\ B_{12} & B_{22} & B_{32} & B_{42} \\ B_{13} & B_{23} & B_{33} & B_{43} \\ B_{14} & B_{24} & B_{34} & B_{44} \end{bmatrix}}$$

If you have more than two matrices, you can still perform this operation. The dot product of the first two matrices creates a temporary matrix that can be applied to the next matrix, and so on. These matrices can then be applied to transform a coordinate. It's important to note that matrix multiplication usually proceeds from right to left, rather than from left to right, which is what happens when treating vectors as column matrices.

Summary

The basis of all the operations you use to create images on the screen are expressed mathematically. This chapter gave a refresher of trigonometry, linear algebra, and vector calculus. You learned about how to use the Pythagorean theorem and normalized vectors to calculate angles of rotation. You learned that triangulation is at the heart of most graphics operations and that everything you do relies on an understanding of those fundamentals.

5. Introduction to Shaders

Color is my day-long obsession, joy and torment.

—Claude Monet

Even though the white triangle from Chapter 3 is cool and all, it doesn't do that much. It doesn't have any shadows, it doesn't move, and it doesn't change color. Additionally, most of the work is being done on the CPU, which negates the whole point of having GPU-accelerated image processing. This is severely limiting—what can you do about that?

Metal Shading Language Overview

Shaders are small programs that are run on the GPU. Even though you can set up most of the work in your main program and not the shaders, it is silly to force the CPU to do this work when the GPU is uniquely suited to doing this work efficiently. Metal shaders are written in the Metal Shading Language (MSL), which is based on C++14. It has specialized data types and functions specifically suited to graphics programming. If you read Chapter 4, "Essential Mathematics for Graphics," a lot of these data types should look familiar.

This chapter focuses on shaders for 3D graphics rendering. Later, we talk more about using shaders for GPGPU programming. In graphics programming, you are required to use two types of shaders: vertex and fragment.

If your only exposure to shaders is using a tool such as ShaderToy, you may think that you only need a fragment shader, and that is not the case. Both the vertex and fragment shaders are separate and vital parts of the programmable graphics pipeline. Thus far, you have used pass-through shaders that don't really change any of the vertex or fragment information that is passed to them. This chapter helps you understand what functions the vertex and fragment shaders are primarily responsible for and gives you a basic understanding of how to make your own shaders.

Although MSL is a C++14-based language, fluency in C++14 is not necessary for writing MSL, but having a passing familiarity helps. Apple's programming guide is more than 100 pages, so this is a large and broad language. This section is an introduction to a few bits you need to know now. More in-depth references to MSL appear in relevant chapters.

MSL has data types for vectors and matrices. The easy way to decode how to create a vector or a matrix is to figure out what data type you want to use for the components and how many of those components you need. For example, a three-item vector of integers would be declared as int3. For MSL, you're usually going to use floats, so you'll primarily be using a lot of float3 and float4 objects.

Matrices follow a similar scheme. For any given matrix, your format will be floatNxM where N is the number of columns and M is the number of rows. The most common matrix type in graphics programming is the old, reliable 4×4 matrix. In MSL, this is written as float4x4.

In addition to MSL is the Metal Standard Library. Most of the math functions that exist in other shading languages, such as GLSL, are directly mappable to the standard library. This includes dot(), normalize(), and so on. You use a few of these for your shader.

It’s impractical to try to memorize MSL and the standard library—particularly the dozens of math concepts, which can quickly become overwhelming if you try to digest them without any context in which to base them. A better approach it is to have a reference to MSL so you can look up specific information based on what you’re currently trying to accomplish.

Setting Up Shaders

One thing that’s helpful when considering how to write a shader is to figure out what your inputs and outputs will be. This goes a long way toward helping you approach shaders. The method signature for all your shaders will look something like this:

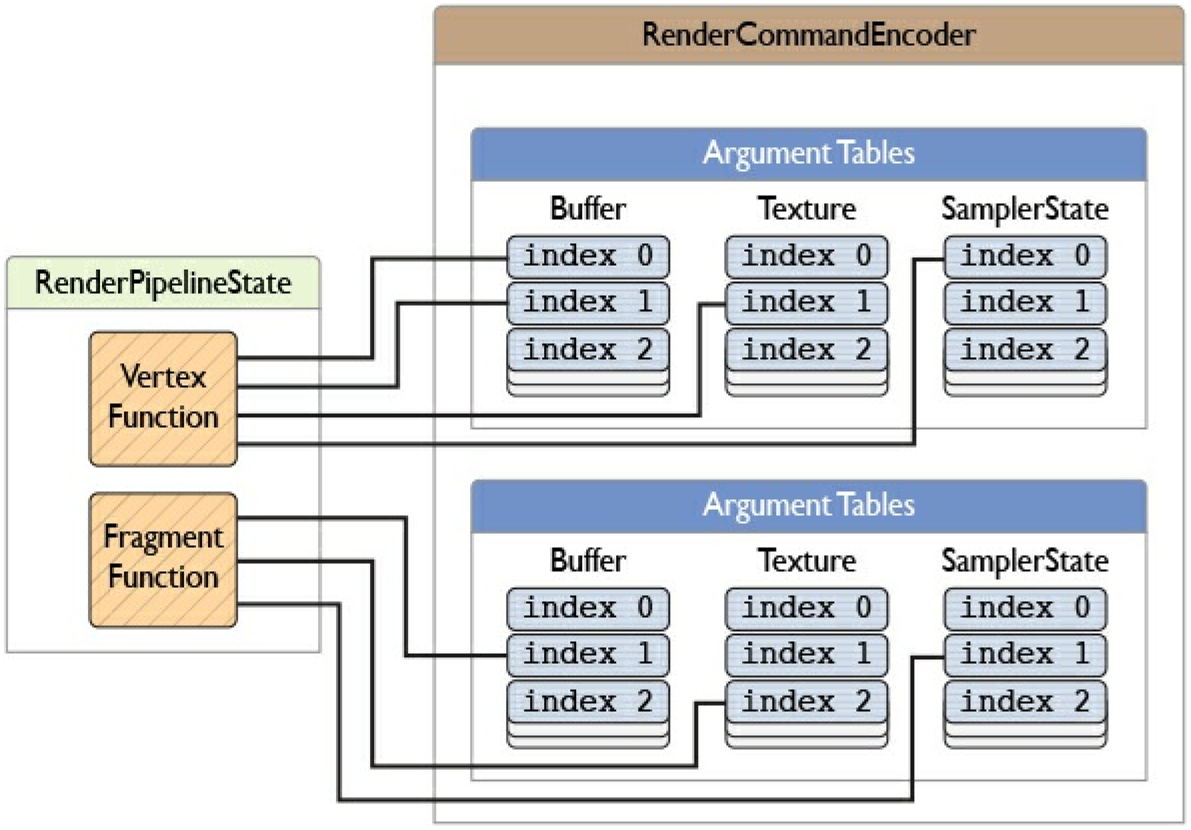
```
shaderType returnType shaderName(parameter1,
                                 parameter2,
                                 etc....)
{
}
```

Each shader must begin with a declaration of type. It can be a vertex, a fragment, or a kernel shader. Next, you need to declare your return type. If you have to return multiple parameters, it’s common to create structs for all the return values and to make the return type an instance of that struct. Next, the shader needs a name. The name is accessed when you’re setting up your render pipeline in the main program. As with other functions, you have parameters to pass in. For vertex shaders, these parameters could be position, color, or other bits of data you need to compose the vertex shader.

Your First Shader: Pass Through

Shaders don’t exist in a vacuum. They need to be connected to the main program so they can share an argument table. The simplest shader programs you can create receive all the data they need from the main program. This data passes straight through the shaders without modification. To understand how these shaders work, you need to have some understanding of how to hook up the shaders using an argument table, as shown in Figure 5.1.

Figure 5.1. *The argument table*



The shaders need to receive information from the vertex buffers. The vertex buffers need to have memory allocated and the vertex data bound to it. The buffers also have to be added to the argument table so they can be accessed by the shaders. The buffers need to be created as an instance variable within the MTKView class because they will be affected by several different methods:

```
var vertexBuffer: MTLBuffer! = nil
var vertexColorBuffer: MTLBuffer! = nil
```

First, you need to specify the names of the shaders that this application will use. Then, you need to generate a pipeline state descriptor that incorporates the Metal functions you have created:

```
guard let defaultLibrary = device.newDefaultLibrary() else {return}
guard let fragmentProgram = defaultLibrary.makeFunction(
    name: "passThroughFragment") else {return}
guard let vertexProgram = defaultLibrary.makeFunction(
    name: "passThroughVertex") else {return}

let pipelineStateDescriptor = MTLRenderPipelineDescriptor()
pipelineStateDescriptor.vertexFunction = vertexProgram
pipelineStateDescriptor.fragmentFunction = fragmentProgram
```

Next, you need to allocate the requisite memory for both of the buffers and give them a label.

```
// generate a large enough buffer to allow streaming vertices for 3
// semaphore controlled frames
vertexBuffer = device.makeBuffer(length: ConstantBufferSize,
                                 options: [])
vertexBuffer.label = "vertices"

let vertexColorSize = vertexData.count * MemoryLayout<Float>.size
vertexColorBuffer = device.makeBuffer(
    bytes: vertexColorData, length: vertexColorSize, options: [])
vertexColorBuffer.label = "colors"
```

Last, you need to set the vertex buffers. This step adds these buffers to the argument table that is shared between the main program and the shaders. You generally create buffers up front as your application starts, but you'll set them in the argument table each frame. You do not need to create buffers every frame.

```
renderEncoder.setVertexBuffer(vertexBuffer,
                              offset: 256*bufferIndex, at: 0)
renderEncoder.setVertexBuffer(vertexColorBuffer, offset:0 , at: 1)
```

Let's look at the default vertex and fragment shaders you get if you start a new project from the Metal template:

```
#include <metal_stdlib>

using namespace metal;

struct VertexInOut
{
   float4  position [[position]];
   float4  color;
};

vertex VertexInOut passThroughVertex(
    uint vid [[ vertex_id ]],
    constant packed_float4* position [[ buffer(0) ]],
    constant packed_float4* color    [[ buffer(1) ]])
{
   VertexInOut outVertex;

   outVertex.position = position[vid];
   outVertex.color    = color[vid];

return outVertex;
};

fragment half4 passThroughFragment(VertexInOut inFrag [[stage_in]])
{
   return half4(inFrag.color);
};
```

This shader file has three components. The first is a struct to contain the output parameters for the vertex shader. The minimal amount that you need to get something to render to the screen is a set of position data, but for this simple example, you are also passing a set of color data. This data can be created in the main program and passed into the shader, as is done here, but that is uncommon. The point of this shader set is to be as bare bones as possible, so the programmers created a struct to hold this position and color data so it can be packaged and passed between the vertex and fragment shaders.

The second component of the shader file is the vertex shader. The output of the vertex shader is the struct previously created. It also takes in three parameters:

• vid**:** This parameter is the vertex ID, which connects to the vertex buffer to bring in the current vertex.

• position**:** This parameter connects to the attribute table. In the default Metal template, there are two attributes in the attribute table: position and color. The position buffer is the first attribute. Here, you are passing into the vertex shader what the current position is for the current vertex.

• color**:** This is the second attribute from the attribute table.

This vertex shader creates an instance of the struct that needs to be returned. It then sets each member of the struct to the parameter that was passed in and returns the struct. No processing takes place—the shader just grabs the current position and color and passes the struct to the rasterizer.

The third structure in this pass-through shader example is the fragment shader. The fragment shader takes in the struct that was returned by the vertex shader. The fragment shader is concerned only with the color of each pixel, so it cherry picks the color component of the struct as its return value.

Packed versus Unpacked

You may have noticed that the vertex shader takes in a weird data type: constant packed_float4. What does that mean? How is it different from a plain vanilla float4?

The color and the position data are both stored in float4s, but they have only three components. Color data has RGB, and position data has XYZ. The last float is padding to get the alignment right with the vectors.

The layout for a position buffer that is unpacked (i.e., consists of float4s) looks like this:

```
XYZ_XYZ_XYZ_XYZ_XYZ_XYZ_XYZ_XYZ_
```

and the layout for a position buffer that is packed (i.e., consists of packed_float4s) looks like this:

```
XYZXYZXYZXYZXYZXYZXYZXYZ
```

where every X, Y, or Z, represents a float comprising 4 bytes. The underscore characters represent space that is consumed by the vector but not actually used for any purpose other than padding. The packed data is about 20 percent smaller due to not having wasted 4 bytes for the "missing" fourth component that is implied in the layout of float4 so that it can be 16-byte aligned.

Generally speaking, when possible, use packed data types. Every little byte helps.

The pass-through shader model is painfully limited. If you want to do anything interesting, you need to dive into shader programming.

Writing Your First Shader

The remainder of this chapter focuses on a simple project that imports a model file and applies a per-vertex lighting model. Importing the data model for this project is detailed in Chapter 11, "Interfacing with Model I/O." For now, be aware that the Model I/O framework has brought in vertex position and normal data that is being stored in buffers that will be sent to the shaders. This section details how to set up the remaining arguments necessary to produce effects using the vertex and fragment shaders.

Per-Vertex Lighting

Lighting is one of the most essential aspects of 3D graphics, and it's covered in Chapter 9, "Introduction to 3D Drawing" For now, this section walks you through implementing a common effect in Metal and helps familiarize you with setting up uniforms, projection matrices, and buffers, using a teapot model.

The shading technique you'll implement is a diffuse, per-vertex shading model with a single-point light source. This means that for every vertex in our model, a determination will be made about how much light it is exposed to. Vertices closer to the light source will have more light exposure than ones that are inside the teapot.

Here is the equation to calculate the light intensity of each vertex:

$$L = K_d L_d \mathrm{s} \cdot \mathrm{n}$$

Now don't freak out! You're going to walk through what this equation means and how to create all the objects and code necessary to translate it to a physical effect. This equation requires several objects to calculate the per-vertex value of the illumination:

• Vertex position

• Vertex normal

• Light position

• Color

• Reflectivity

• Light source intensity

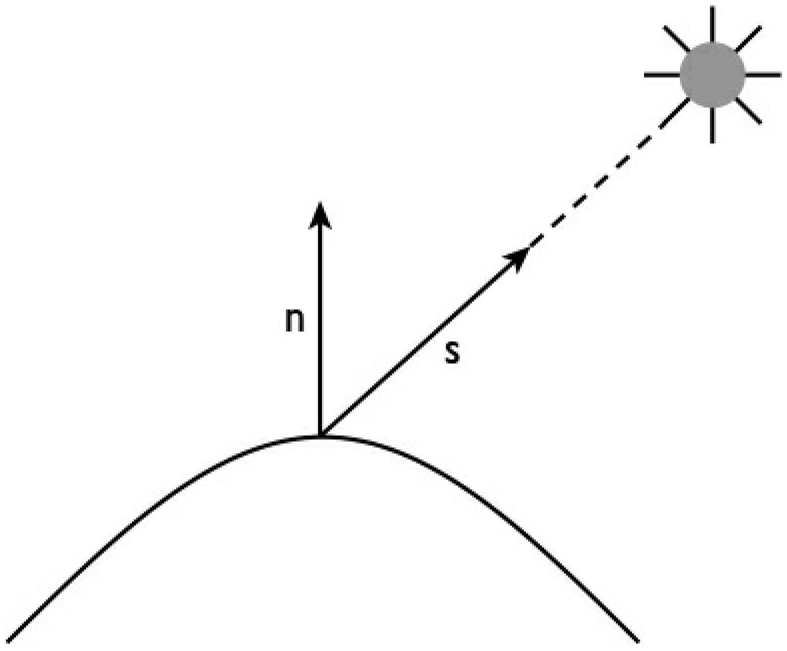
• Model view matrix

• Projection matrix

The foundation for this shader is two vectors: direction from the light source in relation to the surface normal vector. You need to determine the cosine of the angle between these two vectors. As you saw in Chapter 4, there is an easy way to calculate the cosine, which is to use the dot product on the two normalized vectors.

The cosine will be a value between −1 and 1. It determines how close the surface normal is to being in line with the light source. The closer the surface normal is to being in line with the light source, the more light the surface will reflect back, as shown in Figure 5.2.

Figure 5.2. *How the diffuse lighting model works*



Additionally, not all of the light is going to be directly reflected back. All objects, even mirrors, absorb some amount of light. This shader specifically is meant to be diffuse and not shiny. So one last bit of the equation is setting the level of reflectivity to determine how much of the light is absorbed and how much is emitted.

So, looking back at the equation, what you're calculating is the light intensity of each vertex. You take the product of the diffuse reflectivity (K_d), multiplied by the diffuse light intensity (L_d), multiplied by the cosine of the light direction in relation to the surface normal ($s \bullet n$).

Model I/O Vertex Buffer

MSL is designed to be easy to learn if you are familiar with other shading languages, such as GLSL. Some common data types, such as vectors and matrices, and some common functions, such as dot and cross, are necessary in all shading languages. However, because of Metal's design, you can't just copy and paste a GLSL shader directly into Metal.

Here is the original GLSL shader that this project is based on:

```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 LightIntensity;

uniform vec4 LightPosition; // Light position in eye coords
uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ld;            // Diffuse light intensity

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 MVP;

void main()
{
   vec3 tnorm = normalize( NormalMatrix * VertexNormal);
   vec4 eyeCoords = ModelViewMatrix * vec4(VertexPosition,1.0);
   vec3 s = normalize(vec3(LightPosition - eyeCoords));

   LightIntensity = Ld * Kd * max( dot( s, tnorm ), 0.0 );
   gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

Looking at this code and being unfamiliar with MSL, you might think that translating this shader entails creating a bunch of constants in your shader file that will be accessed by both the vertex and fragment shader. This assumption is incorrect. All data used by the shader functions must be passed in as arguments from the argument table or hardcoded into the shader itself.

There are two sets of arguments that need to be passed into the vertex shader. One set of arguments (the vertex position and the vertex normal) is extracted from the model file brought into the project using Model I/O. The other arguments are those that you, the programmer, are responsible for deciding. These include the light position and reflectivity of the surface. If you can't declare those as global uniforms in the shader program, then how does the vertex shader access them?

They get accessed the same way the position and normal data are accessed: by a buffer. First, you're going to look at some code in the view controller. When the model file was brought into the program using Model I/O, the position and normals were bound to a vertex descriptor.

```
let desc = MTKModelIOVertexDescriptorFromMetal(vertexDescriptor)
var attribute = desc.attributes[0] as! MDLVertexAttribute
attribute.name = MDLVertexAttributePosition
attribute = desc.attributes[1] as! MDLVertexAttribute
attribute.name = MDLVertexAttributeNormal
guard let mtkBufferAllocator = MTKMeshBufferAllocator(device: device)
    else {return}
```

This descriptor is used to set up a vertex buffer that is then added to the argument table:

```
let vertexBuffer = mesh.vertexBuffers[0]
renderEncoder.setVertexBuffer(vertexBuffer.buffer,
    offset: vertexBuffer.offset, at: 0)
```

Over in the vertex shader, you need to set up a lock that will fit the key you just created. Both the position and the normal data are interleaved in the first buffer argument in the argument table. You need to specify that argument in the vertex shader declaration:

```
vertex VertexOut lightingVertex(uint vid [[ vertex_id ]],
                                VertexIn *position [[ buffer(0) ]]
{
}
```

Notice that the data type you're telling the shader to expect in the arguments is a VertexIn type. You're also telling the shader that this VertexIn data type is what it expects to find in the first buffer argument in the argument table.

At this point, you might be wondering why you're passing in only one argument when you have two different pieces of information and need both the position and the normal data in order to perform the calculation. How do you access both attributes of the argument?

For every buffer of data that you pass to the shaders, you need to create a correlating structure for that data. You're generally not going to create a new buffer for every single piece of data you pass in. You're going to "stack" and interleave the data, and the shader needs a way of knowing what sets of bytes correlate to what kind of information. Right now, you're passing in a 24-byte chunk of data, and the shader needs a way to understand how these bytes are grouped and how to apply them.

```
struct VertexIn
{
   float3  position [[attribute(0)]];
   float3  normal [[attribute(1)]];
};
```

Here, you are creating a correlating data structure telling the vertex shader how those 16 bytes are divvied up. It's telling the shader there are two attributes in the buffer argument. One is called position, and it uses the first 12 bytes of the argument. The second attribute is called normal, and it occupies the last 12 bytes of the buffer.

Uniform Buffer

The vertex position and vertex normals are in place, but there are still several pieces you need to assemble to implement this shader:

• Light position

• Color

• Reflectivity

• Light source intensity

• Model view matrix

• Projection matrix

Like the vertex position and vertex normals, these elements go into their own buffer. Rather than create a vertex buffer from a model, you create and set this buffer manually. MSL has a set of custom data types and methods that don't exist in Swift, so you must find a way to create correlating data types within the view controller that can connect to the data types in the shaders. You're going to solve this with single instruction, multiple data (SIMD).

One of Apple's frameworks is the Accelerate framework. Accelerate contains C APIs for vector and matrix math, digital signal processing, large-number handling, and image processing. One of the modules in Accelerate is the SIMD library. The data types in SIMD line up perfectly with the data types you need to re-create in MSL.

There is one specialized data type used in this project: the Matrix4x4. SIMD has a float4x4 type that correlates to the Metal float4x4 data type, but you'll need a few specialized matrix types, and this custom Matrix4x4 data type includes convenience methods for creating instances of these types:

```
struct Matrix4x4
{
   var X: float4
   var Y: float4
   var Z: float4
   var W: float4

   init()
   {
       X = Vector4(x: 1, y: 0, z: 0, w: 0)
       Y = Vector4(x: 0, y: 1, z: 0, w: 0)
```

```
        Z = Vector4(x: 0, y: 0, z: 1, w: 0)
        W = Vector4(x: 0, y: 0, z: 0, w: 1)
    }

    static func rotationAboutAxis(_ axis: float4,
                                  byAngle angle: Float32)
        -> Matrix4x4
    {
        var mat = Matrix4x4()

        let c = cos(angle)
        let s = sin(angle)

        mat.X.x = axis.x * axis.x + (1 - axis.x * axis.x) * c
        mat.X.y = axis.x * axis.y * (1 - c) - axis.z * s
        mat.X.z = axis.x * axis.z * (1 - c) + axis.y * s

        mat.Y.x = axis.x * axis.y * (1 - c) + axis.z * s
        mat.Y.y = axis.y * axis.y + (1 - axis.y * axis.y) * c
        mat.Y.z = axis.y * axis.z * (1 - c) - axis.x * s

        mat.Z.x = axis.x * axis.z * (1 - c) - axis.y * s
        mat.Z.y = axis.y * axis.z * (1 - c) + axis.x * s
        mat.Z.z = axis.z * axis.z + (1 - axis.z * axis.z) * c

        return mat
    }

    static func perspectiveProjection(_ aspect: Float32,
                                      fieldOfViewY: Float32,
            near: Float32,
                                      far: Float32) -> Matrix4x4
    {
        var mat = Matrix4x4()

        let fovRadians = fieldOfViewY * Float32(M_PI / 180.0)

        let yScale = 1 / tan(fovRadians * 0.5)
        let xScale = yScale / aspect
        let zRange = far - near
        let zScale = -(far + near) / zRange
        let wzScale = -2 * far * near / zRange

        mat.X.x = xScale
        mat.Y.y = yScale
        mat.Z.z = zScale
        mat.Z.w = -1
        mat.W.z = wzScale

        return mat;
    }
}
```

The light position and the color are represented by a float4. The light intensity and the reflectivity are represented by a float3. The projection matrix and the model view matrix are both represented by a Matrix4x4.

Now, you have six objects of three different types that need to be bundled together and sent to the GPU. If these were all the same data type, it would be easy to bundle them into an array, but they're different types. Consequently, you must bundle them into a struct:

```
import simd

struct Uniforms {
   let lightPosition:float4
   let color:float4
   let reflectivity:float3
   let lightIntensity:float3
   let projectionMatrix:Matrix4x4
   let modelViewMatrix:Matrix4x4
}
```

Projection Matrix

If you worked through Chapter 3, “Your First Metal Application (Hello, Triangle!),” you might be wondering why you now need a projection and a model view matrix when you could render a triangle to the screen without them.

If you think back to the triangle, it didn’t come out the way you thought it should have. Each vertex is equidistant from the other two, so you should have had an equilateral triangle. But if you built and rendered the code, you will have noticed that the triangle was stretched.

If you don’t provide a projection matrix, your program will be biased and scaled by the dimensions of your screen, causing the figure to be distorted unless you compensate for the distortion with a scaling matrix. Because your screen is not square, all your models will be distorted. That’s a fine thing to ignore while you’re trying to learn all the moving parts, but you don’t want to establish bad habits. If your artist spent a lot of time perfecting a model for your game or application, you want to do it justice by making sure its aspect ratio is correct.

To keep this chapter simple, each of these uniforms will be hardcoded. It wouldn’t be too much work to create outlets to allow the user to change these values, but that adds an extra layer of complexity to an already complex chapter. You need to create six objects to populate the uniform struct:

```
// Vector Uniforms
let teapotColor = float4(0.7, 0.47, 0.18, 1.0)
let lightPosition = float4(5.0, 5.0, 2.0, 1.0)
let reflectivity = float3(0.9, 0.5, 0.3)
let intensity = float3(1.0, 1.0, 1.0)

// Matrix Uniforms
let yAxis = float4(0, -1, 0, 0)
var modelViewMatrix = Matrix4x4.rotationAboutAxis(yAxis,
                                 byAngle: rotationAngle)

modelViewMatrix.W.z = -2

let aspect = Float32(self.view.bounds.width) / Float32(self.view.bounds.height)

let projectionMatrix = Matrix4x4.perspectiveProjection(aspect,
                            fieldOfViewY: 60,
                            near: 0.1,
                            far: 100.0)
```

After you create each of your uniforms, you need to populate a uniform struct. To add this hardcoded information to a uniform buffer, you need to allocate enough memory for it. You also need to wrap the uniform struct instance in an array so that the data can be copied over to the buffers:

```
let uniform = Uniforms(lightPosition: lightPosition,
                       color: teapotColor,
                       reflectivity: reflectivity,
                       lightIntensity: intensity,
                       projectionMatrix: projectionMatrix,
                       modelViewMatrix: modelViewMatrix)

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

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

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

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

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

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

Vertex Shader

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

$L = K_d L_d \mathrm{s.n}$

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

Here is your finished vertex shader:

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

   // Lighting code
   float3 tnorm = normalize(uniforms.projectionMatrix *
                      vertexIn.normal);
   float4 eyeCoords =
       uniforms.modelViewMatrix * float4(vertexIn.position, 1.0);
   float3 s = normalize(float3(uniforms.lightPosition –
                                  eyeCoords));

   outVertex.lightIntensity =
       uniforms.intensity * uniforms.reflectivity *
                                max( dot(s, tnorm),
       0.0);
   outVertex.position =
       uniforms.modelViewMatrix * float4(vertexIn.position, 1.0);

    return outVertex;
};
```

Fragment Shader

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

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

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

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

Summary

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

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

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

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

6. Metal Resources and Memory Management

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

—Peter Drucker

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

Introduction to Resources in Metal

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

• **MTLBuffer**: An allocation of unformatted memory that can contain any type of data

• **MTLTexture**: An allocation of formatted image data with a specified texture type and pixel format

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

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

The Argument Table: Mapping between Shader Parameters and Resources

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

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

• Buffers

• Textures

• Sampler states

Sampler states are discussed in Chapter 12, “Texturing and Sampling.” This chapter focuses on the first two resource types. As of 2017, you can have, at most, 31 different buffers and 31 different textures on iOS devices. They are subject to change based on the GPU, so be sure to check the documentation. These resources are contained in an array and can be accessed by element number. So, for example, the first element in the buffer array would be at location 0.

You specify whether the argument in the argument table goes to the vertex or the fragment shader by specifying what kind of buffer or texture you are creating. As you saw in Chapter 5, you can interleave related data into the same buffer if all the data will be used in the render pass. If you are importing a model that has five different arguments to send to the vertex shader, all of the arguments can be consolidated into one buffer and one slot in the argument table.

Buffers

MTLBuffer objects are allocations of unformatted blocks of memory. Buffer data can be anything. Think back to Chapter 5, where you created two buffers of data. One buffer contained the position and normal data from the model you imported. The other buffer contained all of your uniform variables. That buffer contained floats and matrices interleaved together. You were able to interleave them because the buffer objects look at those uniforms as a giant blob of data. It doesn’t matter to the buffer whether it holds floats or matrices—it just sees the bytes that compose the buffer.

Like MTLDevice, MTLBuffer is a protocol. You don’t instantiate it directly or subclass it. Rather, there are a few factory methods that you can use to create a buffer:

• makeBuffer(length:options:) creates a MTLBuffer object with a new storage allocation.

• makeBuffer(bytes:length:options:) creates a MTLBuffer object by copying data from an existing storage allocation into a new allocation.

• makeBuffer(bytesNoCopy:length:options:deallocator:) creates a MTLBuffer object that reuses an existing storage allocation and does not allocate any new storage.

Again, the buffer doesn’t care what data it contains; it just cares about knowing how large it has to be to contain the data.

Resource Options: Storage Mode, Cache Mode, Purgeability

Constantly having to check and fetch memory from RAM is computationally expensive. Metal is all about opening up choices to you, the developer, to allow you to tune your applications as you see fit. MTLResource has several associated constants that help you to customize how data is stored and cached:

• MTLCPUCacheMode

• MTLStorageMode

• MTLPurgeableState

The cache mode determines how the CPU maps resources. You want to map your resources as contiguously as possible. It’s computationally expensive to jump around in memory, so try to map your data so it is accessed in a stream. MTLCPUCacheMode has two options: defaultCache and writeCombined. The default cache guarantees that read and write operations are executed in the expected order. The only time you want to use writeCombined is when you are creating resources the CPU will write into but read later.

The cache mode is intimately connected to the storage mode. MTLStorageMode defines the memory location and access permissions of a resource. If a resource is shared, it is accessible by both the CPU and the GPU. If a resource is private, then it is only accessible by the GPU. If a resource is managed, then both the CPU and the GPU have a copy of the resource. These copies are synchronized to reflect changes in the other copy of the resource. The last storage mode is memoryless. Memoryless storage is backed by neither the CPU nor the GPU. This may sound rather pointless, but it’s used in temporary render targets for things like rendering textures. You use it when you have data to render that does not need to persist beyond the pass in which it is generated.

You’re not going to want to hold onto all of these objects in memory forever, so you need to have a plan for how to remove them appropriately. MTLPurgeableState allows you to change the state of the resource. It’s not necessary for you to control the purgeability of every application. If you want to access the resource without changing its purgeable state, you set the state to keepCurrent. If you do not want your resource to be discarded from memory, you set the state to nonVolatile. The volatile setting states that the resource can be removed from memory if it’s no longer needed, but it’s not automatically purged. Once you determine that the resource is absolutely no longer needed, the state is set to empty.

Preparing Data for the Vertex Shader and Vertex Descriptors

This chapter has spent some time going over how to map data to memory. The end purpose of mapping memory as efficiently as possible is to make it as fast as possible to pass this data to the GPU to process. You prepare data for the vertex shader using a vertex descriptor. This section covers that in more detail.

A MTLVertexDescriptor object is used to configure how vertex data stored in memory is mapped to attributes in a vertex shader. The MTLVertexDescriptor is part of the pipeline state, and it establishes the vertex layout for the function associated with the pipeline. There is only one MTLVertexDescriptor per pipeline state.

The vertex descriptor is used to describe the vertex layouts and vertex attributes. The vertex descriptor has a MTLVertexAttributeDescriptorArray property that contains MTLVertexAttributeDescriptor objects. Look at this example:

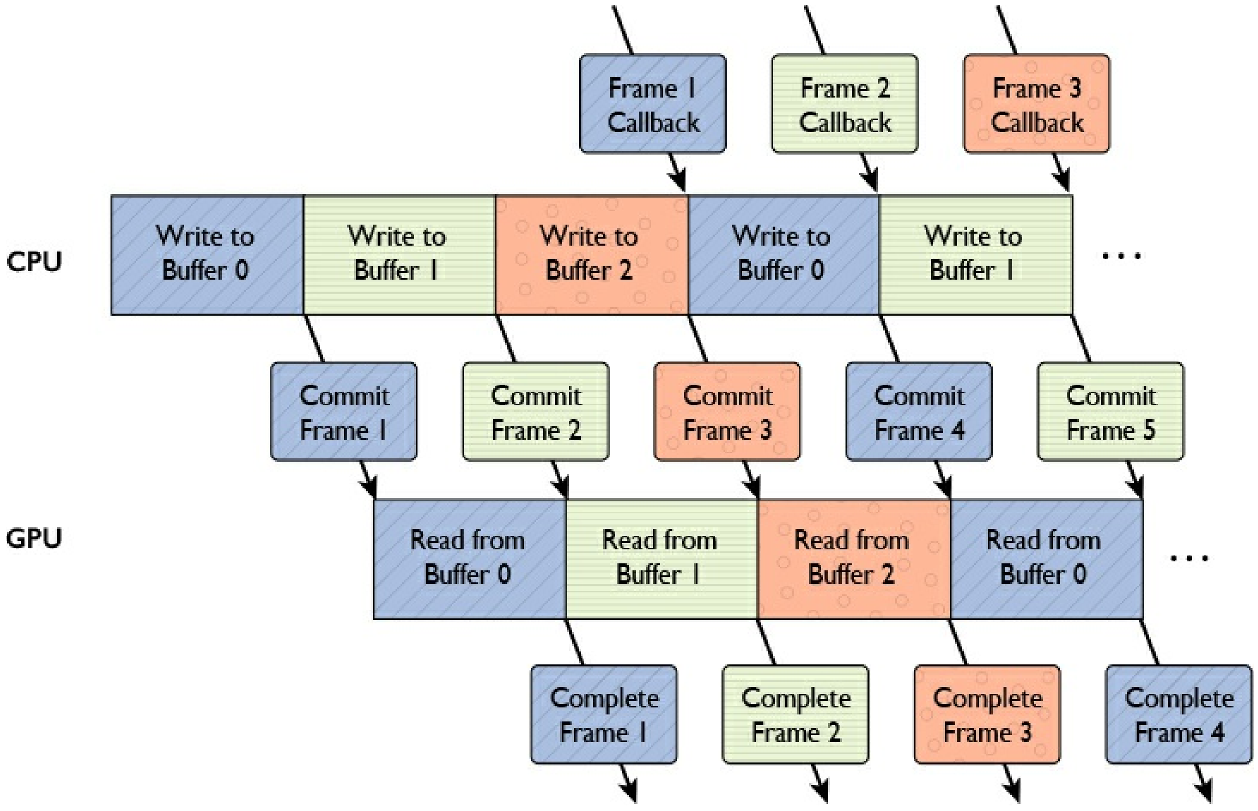
```
vertexDescriptor.attributes[0].offset = 0
vertexDescriptor.attributes[0].format = .float3 // position
vertexDescriptor.attributes[1].offset = 12
// Vertex normal
vertexDescriptor.attributes[1].format = MTLVertexFormat.float3
vertexDescriptor.layouts[0].stride = 24

let pipelineStateDescriptor = MTLRenderPipelineDescriptor()
pipelineStateDescriptor.vertexDescriptor = vertexDescriptor
```

In this example, you have two sets of information that you need to encode into the buffer: the vertex position and the vertex normal. Each one is stored as an attribute within the vertex descriptor. Each attribute has an offset and a format. You are responsible for telling the driver at what point in memory your attribute is situated and what type of data it is expected to have.

Copying to and from Buffers

While buffers themselves are immutable, the data within them is not. A buffer of data is prepared by the CPU and then processed by the GPU. These two things cannot happen at the same time on the same buffer. If the CPU overwrites the data that the GPU is currently processing, the result will be data corruption. In Metal, it's common to use a *triple buffering system*, as shown in Figure 6.1. What this means in practice is that you create three buffers. The CPU copies its data to the first buffer, then hands it off to the GPU. While the GPU is working on the first buffer, the CPU starts to copy data to the second buffer. This creates an offset so that neither the CPU nor the GPU is ever just waiting around for something to process.

Figure 6.1. *Triple buffering system*



Introduction to Textures

One common aspect of 3D graphics programming that allows for incredibly detailed objects with low overhead is texture mapping. Creating a craggy rock as a mesh and generating shadows is computationally expensive, especially if it's a part of the background and not a vital part of your application. Creating a low-poly regular shape and then applying a bumpy-looking texture to it is a good way to free up time and CPU cycles to add better features to your game without losing too much detail in your application.

A MTLTexture object represents an allocation of formatted image data that can be used as a resource for a vertex shader, fragment shader, or compute function or as an attachment to be used as a rendering destination. A texture can be one of the following types:

• A 1D, 2D, or 3D image

• An array of 1D or 2D images

• A cube of six 2D images

MTLResource has an implementation for textures: MTLTexture. Similar to MTLBuffer and MTLDevice, this is a protocol with which you create an object that conforms to it using designated methods. There are three methods to create a MTLTexture:

• **newTextureWithDescriptor**: This method is on MTLDevice to create a new texture. It uses a MTLTextureDescriptor to configure the texture object.

• **makeTextureView(pixelFormat:)**: This method is on the MTLTexture protocol. It creates and returns a new texture object that shares the same storage allocation as the source texture object. Because they share the same storage, any changes to the pixels of the new texture are reflected in the source texture, and vice versa.

• **newTextureWithDescriptor:offset:bytesPerRow**: This method is on yet another protocol, MTLBuffer. It is similar to makeTextureView(pixelFormat:) in that the new texture shares the same storage allocation as the source buffer object, so the same rules apply. In newTextureWithDescriptor:offset:bytesPerRow, you create a new texture from a texture descriptor. MTLTextureDescriptor configures your new texture object. This includes pixel format, height, and mipmap level count. All of the aspects we have been talking about so far in regard to the storage mode and type are configured here.

If you use the makeTextureView(pixelFormat:) method, you have to specify the pixel format. The MTLPixelFormat describes the organization of color, depth, or stencil data storage in individual pixels of a texture. There are three varieties of pixel formats: ordinary, packed, and compressed. For ordinary and packed formats, you need to specify three descriptions: order of color components, bit depth of the color components, and data type for the components. Format data is stored in little-endian order. Compressed format is covered a little later in this chapter.

This section is a cursory overview of textures as resources. Textures are covered more comprehensively in Chapter 12.

Loading Image Data with MTKTextureLoader

In 2015, Apple introduced MetalKit to streamline some of the most common functionality in Metal. Since uploading and applying textures is incredibly common in graphics programming, there is an implementation for it in MetalKit.

The MTKTextureLoader class simplifies the effort required to load your texture data into a Metal application. This class can load images from common file formats such as PNG, JPEG, and TIFF. All textures loaded using MTKTextureLoader conform to the MTLTexture protocol.

```
if let textureUrl = NSURL(string: fileLocation) {
    let textureLoader = MTKTextureLoader(device: device)
    do {
        diffuseTexture =
            try textureLoader.newTextureWithContentsOfURL(
textureUrl,
                options: nil)
  } catch _ {
        print("diffuseTexture assignment failed")
    }
}
```

In this code snippet, you are creating a new MTKTextureLoader from your default MTLDevice. The texture is located in your app bundle. You load the texture by calling textureLoader.newTextureWithContentsOfURL on your new texture loader. This method may fail if you try to load a texture that isn't available or if you have a misspelling. You don't want the app to crash if that happens, so you catch the error and print an error message to the console.

It's also possible to load textures from asset catalogs, Core Graphics, and Model I/O. Every one of these functions begins with textureLoader.newTextureWith. It simply depends on you to choose which source the texture comes from. This makes it easy to streamline bringing in textures from various different sources without having to remember too many differences.

Texture Views

Earlier in this chapter, you learned how to make MTLTexture objects. One way of doing it is to call a method on the MTLTexture protocol to create a texture view. This section explains the process a little more in depth.

Unlike the device method, this method doesn't allocate memory to make a copy of the texture data. This method reinterprets the existing texture image data in a new specified pixel format. It doesn't modify the original underlying data file. The pixel format of the new texture must be compatible with the pixel format of the source texture.

func makeTextureView(pixelFormat: MTLPixelFormat) -> MTLTexture specifies the new pixel format you want to convert the texture to and returns that new texture. Not all pixel formats are compatible with all other pixel formats. You cannot use this method between color pixel formats of different sizes.

The other way to make a texture view is with this method:

```
func makeTextureView(pixelFormat: MTLPixelFormat,
                     textureType: MTLTextureType,
                     levels levelRange: NSRange,
                     slices sliceRange: NSRange) -> MTLTexture
```

The previous method only allows you to change the pixel format. This method also allows you to change the texture type. Similar to MTLPixelFormat, the MTLTextureType has a limited number of other textures that a texture can be converted to. A type1D texture can be converted only to another type1D texture, whereas a type2D texture can be converted to either another type2D or a type2DArray.

Texture Arrays

It's common in 3D graphics to have more than one texture attribute to bind to an object. Texture atlases are just one example of a common practice that bundles multiple assets for one object. So far, you've learned about 1D, 2D, and 3D textures, but you haven't read about how to store these bundles of assets. These are bundled using texture arrays.

Texture arrays are used to bind several textures at once. These textures are generally related. Besides texture atlases and sprite sheets, you can use texture arrays to hold various material maps. All textures in a texture array must have a single shared width and height.

Copying to and from Textures

Textures are not always static. It's common to need to copy image data to new textures or to replace certain areas of your texture with other image data. It's important to understand how to copy this data effectively so that you do not slow down your application or create unnecessary and expensive memory operations.

There are two steps to this process: retrieving the bytes you want to use to replace an area of the texture and then replacing that area with those bytes.

```
func getBytes(_ pixelBytes: UnsafeMutableRawPointer,
              bytesPerRow: Int,
              from region: MTLRegion,
              mipmapLevel level: Int)
```

The first parameter you need for this function is an unsafe mutable raw pointer to the bytes you want access to. If you're a Swift programmer, that can look scary and unsafe, but you're a Metal programmer now! Getting close to the metal means getting comfortable with working directly with raw memory pointers. If you know what you're doing, you'll be fine. bytePerRow is the stride between rows of texture data. The region is the location of a block of pixels in the texture slice. Finally, if your texture is mipmapped, this is the level. If your texture is not mipmapped, the level defaults to zero.

Next, you need to specify what is being replaced.

```
func replace(region: MTLRegion,
             mipmapLevel level: Int,
             slice: Int,
             withBytes pixelBytes: UnsafeRawPointer,
             bytesPerRow: Int,
             bytesPerImage: Int)
```

First, you specify the region. The region is a rectangular block of pixels in an image or texture, defined by its upper-left corner and its size. Again, you specify the mipmap level, if applicable. The slice identifies which texture slice is the destination for the copy operation. The rest is the same as the getBytes method.

Compressed Texture Support

Compression is a fact of life in graphics and media programming. Without compression, a few minutes of video could take up gigabytes of storage. Being able to work with compressed textures in Metal is vitally important. Compressed formats utilize sampling and tend to be lossy. The compressed format is loaded into memory and is decompressed only when needed by the GPU.

Metal supports three compressed texture formats:

• **PVRTC**: PVRTC (PowerVR Texture Compression) operates by downsampling the source image into two smaller images, which are upscaled and blended to reconstruct an approximation of the original. One thing to keep in mind when using PVRTC format on iOS is that textures must be square, and each dimension must be a power of two.

• **ETC2**: ETC (Ericsson Texture Compression) is similar to PVRTC. It compresses each 4×4 block of pixels into a single 64-bit quantity, but PVRTC is usually higher quality.

• **ASTC**: ASTC (Advanced Scalable Texture Compression) is a newer and more flexible texture format, but it is not supported on the A7 chip.

The Blit Command Encoder

At this point, you may be wondering about the toll all of this copying is having on the CPU. A lot of graphics frameworks abstract away enough of the underlying structure that, as a beginning graphics programmer, you're not really sure if your code is primarily being processed on the CPU or the GPU. Since Metal's purpose is to expose this information to you, there is a special encoder you can use to ensure that your data is being processed on the GPU.

The MTLBlitCommandEncoder is used to encode resource-copying commands into a command buffer. These commands are used to manage the contents of textures and buffers. Thus far, you have only worked with the MTLRenderCommandEncoder. It's important to remember that with Metal you are going to use multiple command encoders to package and process data for the GPU. Bear this in mind when you go to set up another command encoder.

Multiple methods are available to you to manage data without touching the CPU:

• Copying data between two buffers

• Copying data between two textures

• Copying data from a buffer to a texture

• Copying data from a texture to a buffer

• Generate data for fills and mipmaps

• Synchronize data

Any operation that copies data from one place to another uses the MTLBlitCommandEncoder's copy method. This method is overloaded to reflect the different ways that data can be copied. The simplest example of this copy method is copying from one buffer to another:

```
func copy(from sourceBuffer: MTLBuffer,
          sourceOffset: Int,
          to destinationBuffer: MTLBuffer,
          destinationOffset: Int,
          size: Int)
```

Since textures are more complex than buffers, their copy functions will be more complex and customizable.

```
func copy(from sourceBuffer: MTLBuffer,
          sourceOffset: Int,
          sourceBytesPerRow: Int,
          sourceBytesPerImage: Int,
          sourceSize: MTLSize,
          to destinationTexture: MTLTexture,
          destinationSlice: Int,
          destinationLevel: Int,
          destinationOrigin: MTLOrigin)
```

One special generator function on the Blit Command Encoder is generating mipmaps. That situation is covered in the next section.

Generating Mipmaps

In graphics programming, you are going to have situations where your texture does not exactly match what is on the screen pixel for pixel. The user will zoom in to your model, and the textures will become really large and pixelated. The user will zoom way out, and the texture that was originally 64 pixels square will take up only 20 pixels on the screen.

The process of dealing with the second scenario is called *mipmapping*. Since it’s to be expected that your user is going to zoom in and out on your textures, Metal has built-in methods to generate these mipmaps. The process and logic behind mipmapping is explained in further detail in Chapter 12. For now, just understand that mipmaps are a way to ensure your models look nice, and not jaggy, no matter what magnification they’re at.

The MTLBlitCommandEncoder has a method on it to generate a mipmap for you:

```
func generateMipmaps(for texture: MTLTexture)
```

Mipmaps are generated with scaled images for all levels up to the maximum level. The mipmap level count for the texture must be greater than 1. Mipmap generation works only for textures with color-renderable and color-filterable pixel formats.

Summary

Metal resources are sources of data for the GPU. They can be buffers of unformatted data, or they can be textures. Textures are flexible and can be contained in arrays and even compressed.

7. Libraries, Functions, and Pipeline States

An original idea. That can't be too hard. The library must be full of them.

—Stephen Fry

When you move from simple to more complex applications, you begin needing more and different shaders. Even though you've read about and worked with shaders, you have not yet learned how to organize large sets of shaders in a logical way. This chapter outlines how Metal stores, compiles, and accesses shaders.

What Are Libraries and Functions?

You might wonder why you haven't yet learned about functions if they're so important, but in fact, you have been working with them without realizing it. MTLFunction is your shader function. A MTLFunction is anything that is written in the Metal Shading Language (MTL) and is contained in a .metal file. Functions come in three flavors: *vertex*, *fragment*, and *kernel*. Vertex and fragment functions are associated with the MTLRenderPipelineState, and kernel functions are associated with the MTLComputePipelineState.

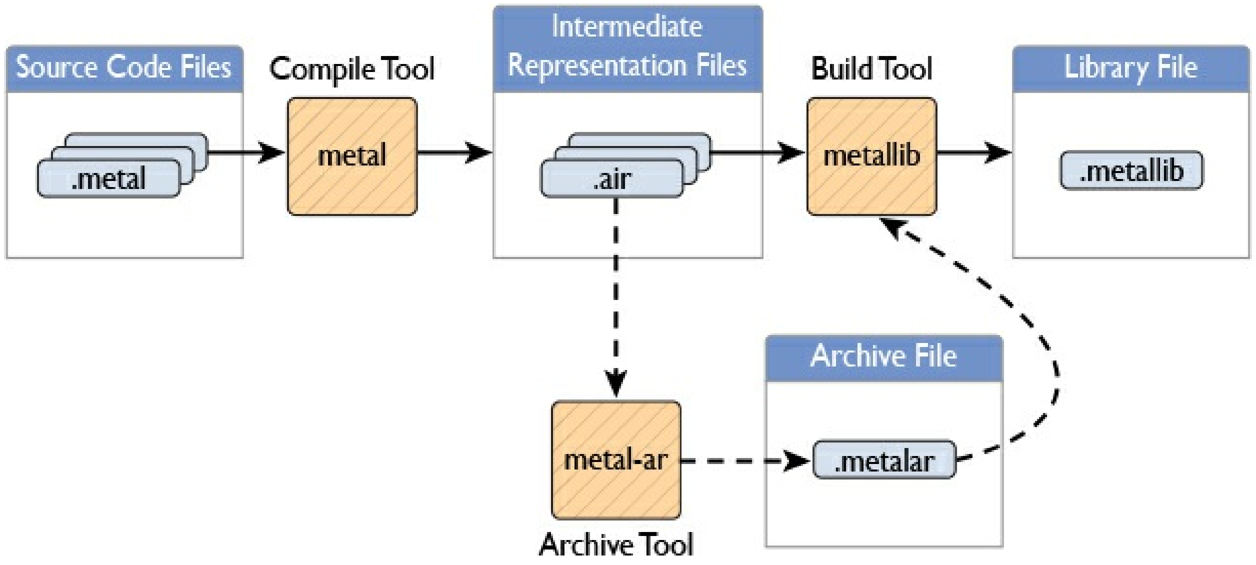
Functions are collected into MTLLibrary objects. The MTLLibrary protocol defines the interface for an object that represents a library of graphics or compute functions. Libraries can be made up of compiled code or can be retrieved from source code that is read from a string. This chapter goes into the implementation details of how to determine the best option for your program.

The Metal Two-Phase Compilation Architecture

Let's go over some compiler theory 101. You, as a programmer, know that the characters and code you type into Xcode don't translate directly into a format the computer can understand. The computer speaks machine code, which you could learn to write directly, but it would be a major pain to do. Programming languages were developed to create human-readable ways to interface with the computer. So, if the computer can't directly read the code you write, how does it work?

You know that when you build or run programs in Xcode, you are compiling your code. This is different from languages such as JavaScript in which the code is interpreted at runtime. When you trigger a build of your code, Xcode triggers a Metal front-end compiler similar to the Clang compiler used in C, C++, and Objective-C. This compiler turns your source code into an abstract syntax tree (AST). From this tree, the compiler creates an intermediate representation (IR). This is the first step of the two-phase compilation process, as shown in Figure 7.1.

Figure 7.1 *The two-phase compilation architecture*



Getting the IR doesn't mean that your code is ready to go. Multiple GPUs support Metal, and each GPU has its own unique instruction set. The second phase of the compilation process is taking this IR and translating it to the specific machine code for each GPU. The second phase is done on the specific device the program is running on. It takes the IR and builds the backend commands specific to their GPU.

This two-phase compilation structure ensures that your Metal library compiles properly on every supported GPU without having to weigh down the application bundle with compiled code that can't run on the GPU.

Creating Libraries at Compile Time and Runtime

One of the most computationally expensive operations in graphics programming is building the shaders. One of the optimizations that Metal offers is the ability to build the shaders at compile time rather than at runtime. Doing so moves an incredibly expensive operation that potentially has to be done only once to a point in the process where it doesn't affect the user. You can even use command-line tools to build your library outside of Xcode.

At build time, any .metal file present in the compile sources phase of your application target will be compiled by the Metal frontend compiler, and any functions they contain will comprise the default library. This library is accessed by a method on the default MTLDevice: newDefaultLibrary(). The return type on this method is optional, so if no default library is built at compile time, it won't crash the application.

It's also possible to bring Metal library binaries into your project. If, for example, you created an open source image-processing framework, it would be possible to import that Metal library in other people's projects. Libraries can be imported as files or as data.

In some situations, you'll need to create a function without being able to add a .metal file. One example is if you're creating a plug-in for something like Unity. In situations like this, you can create a shader from a text file that will be compiled and added to the default library. The method on MTLDevice to perform this operation synchronously is as follows:

```
func makeLibrary(filepath: String) throws -> MTLLibrary
```

A variation on this code allows for asynchronous library creation using a completion handler:

```
func makeLibrary(source: String,
                 options: MTLCompileOptions?,
                 completionHandler: @escaping
                                    MTLNewLibraryCompletionHandler)
```

Loading Functions from a Library

Successfully creating a library is a good first step, but it’s not going to do you any good if you can’t integrate the functions into your application. This section covers how to load a function from your default library into your application. MTLLibrary has a method on it called makeFunction:

```
guard let fragmentProgram = defaultLibrary.makeFunction(
    name: "lightingFragment") else {return}

guard let vertexProgram = defaultLibrary.makeFunction(
    name: "lightingVertex") else {return}
```

This method is the easiest way to pull a specific function out of the default library. The makeFunction method takes a string that represents the name of a function in the default library. If you are using a renderer, you need to pull out both a vertex and a fragment function. If you are doing computing, you need to pull out a kernel function.

Command Encoders

Connecting functions in a library to a processing pipeline is intuitive in regard to how it works for graphics processing, but what if you are interested in general-purpose GPU programming? If you wanted to set up a neural network and had a bunch of MTLFunction objects associated with that, what would you use?

MTLDevice has a command queue that is loaded with command encoders. Each encoder does a specific type of work. You would not use a graphics encoder to do general-purpose GPU programming. So far, you’ve primarily encountered the render command encoder, but that is only one of several command encoders. The following encoders are supported by Metal:

• **MTLRenderCommandEncoder**: This encoder is designed to generate a single rendering pass. It accesses both vertex and fragment shaders.

• **MTLBlitCommandEncoder**: This encoder was discussed in Chapter 6, “Metal Resources and Memory Management.” It is used only to copy resources to and from buffers and textures. This encoder’s work does not require it to use any MTLFunctions written by you.

• **MTLComputeCommandEncoder**: The GPGPU command encoder accesses kernel functions exclusively.

• **MTLParallelRenderCommandEncoder**: This encoder takes one or more MTLRenderCommandEncoder objects. It allows you to assign each render encoder to its own thread so the rendering commands can be executed in parallel. This encoder doesn’t access the MTLLibrary directly. Rather, it is a managing object for encoders that do access the MTLLibrary.

One of the most important things to remember in Metal is how to set up your data to be processed. By remembering all the options you have for the different types of command encoders, you can ensure you’re using the best tool for the job.

Render Pipeline Descriptors and State

Having a library of Metal functions is great, but you need a way to add them to the render pipeline. This is done with the use of MTLRenderPipelineDescriptor objects. The MTLRenderPipelineDescriptor specifies the rendering configuration state used during a rendering pass, including rasterization (such as multisampling), visibility, blending, tessellation, and graphics function state. Here is an example of a MTLRenderPipelineDescriptor:

```
let pipelineStateDescriptor = MTLRenderPipelineDescriptor()
pipelineStateDescriptor.vertexFunction = vertexProgram
pipelineStateDescriptor.fragmentFunction = fragmentProgram
pipelineStateDescriptor.vertexDescriptor = vertexDescriptor
```

The first thing you do to set up the pipeline state is to specify which vertex and fragment programs you are using from the default library. After the pipeline state descriptor is set, it needs to be used to initialize a render pipeline state.

The render pipeline state is one ingredient of the secret sauce that makes Metal so much more efficient than OpenGL. Much of the render state that you need is baked into this pipeline descriptor. Rather than forcing the renderer to validate the state on every draw call, it already knows that this state is valid.

After you specify everything that will go into your pipeline state, you need to add it to the render pipeline. This process can fail, so it needs to be enclosed in a do-try-catch block.

```
do {
    try pipelineState = device.makeRenderPipelineState(
                        descriptor: pipelineStateDescriptor)
} catch let error {
    print("Failed to create pipeline state, error \(error)")
}
```

Pipeline Reflection

So far, this chapter has talked about a MTLFunction being a way to access a shader that is written in a separate Metal file. The MTLFunction object does not provide access to function arguments. The only properties you can access on a MTLFunction are

• Name

• Type of function

• Vertex attributes

That list does not include details of shader or compute function arguments. Those arguments must be obtained using a MTLRenderPipelineReflection or a MTLComputePipelineReflection. The MTLRenderPipelineReflection has only two properties on it: vertexArguments and fragmentArguments. Both are optional arrays of MTLArgument objects. If you need to access any of the arguments being passed to the vertex or fragment shaders, you must access them through the pipeline reflection—and not from the MTLFunction directly.

Summary

It’s vitally important to fully understand how Metal builds and accesses libraries of shaders. Metal considers shaders to be function objects. Metal must account for many different GPU architectures while also ensuring you don’t bundle code you don’t need. It’s also important to understand the costs and benefits of the point at which you decide to implement your shader compilation.

8. 2D Drawing

Begin at the beginning and go on till you come to the end; then stop.

—Lewis Carroll

Although it's primarily known as a 3D graphics framework, Metal can also render 2D images to the screen, along with GPGPU programming.

In Metal, adding dimensions increases complexity. Because rendering a 2D image is less complicated than rendering a 3D one, it's a good place to start. This exercise familiarizes you with all the pieces you need to create a 3D image. Because this topic takes practice to master, this chapter focuses on the basics with no added complexity.

Metal Graphics Rendering Pipeline

Pulling a drawing out of thin air requires precision and processing. Each program must go through a series of steps (or rendering pipeline) during a draw call before it can accurately be rendered to the screen:

1. Pass the primitives to the vertex shader.

2. Assemble the primitives that will construct your image.

3. Rasterize the image.

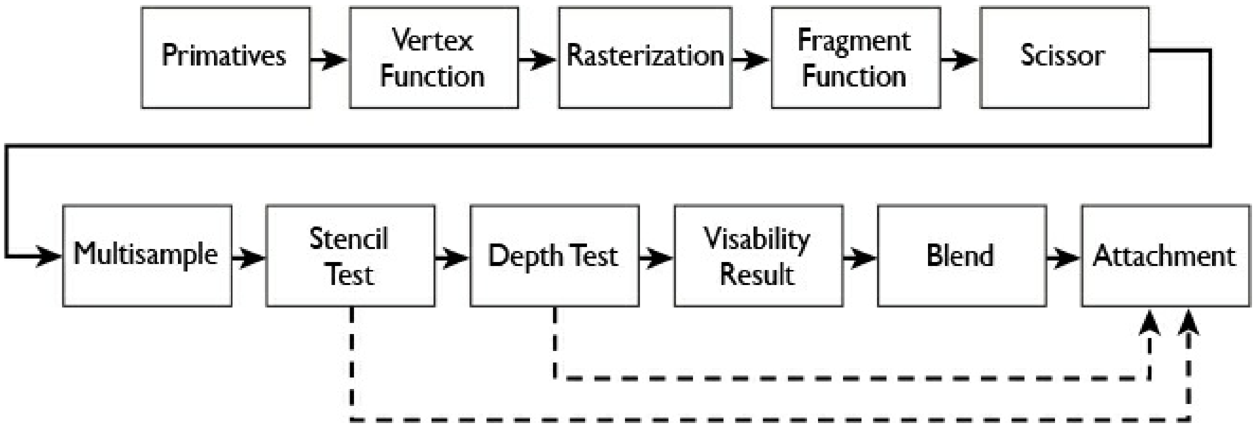
4. Pass the raster data to the fragment function.

5. Check the fixed function pipeline states on the command encoder.

6. Present the drawable to the screen.

This chapter walks you through how to create your initial set of data, process it, and pass it through the pipeline seen in Figure 8.1 until it is presented to the screen.

Figure 8.1. *Metal graphics rendering pipeline*



Sample Project: Build a Star

The “Hello, World!” of graphics projects is a 2D triangle rendered to the screen. We add slightly more complexity to this old chestnut to illustrate the point that everything comes back to triangles.

Instead of just rendering a triangle, you learn how to render a star. The skills you use to render the star can be directly mapped to rendering any shape you want. This exercise also helps you see how to set these parts up for more complex rendering.

Generally, any complex shapes you use in a production project will be imported as a model. Because this is such a small project, the vertices are entered by hand.

The goal is to generate a star with a white center and red tips. This project is available on GitHub for your reference. Let’s walk through all the parts we need to make this happen.

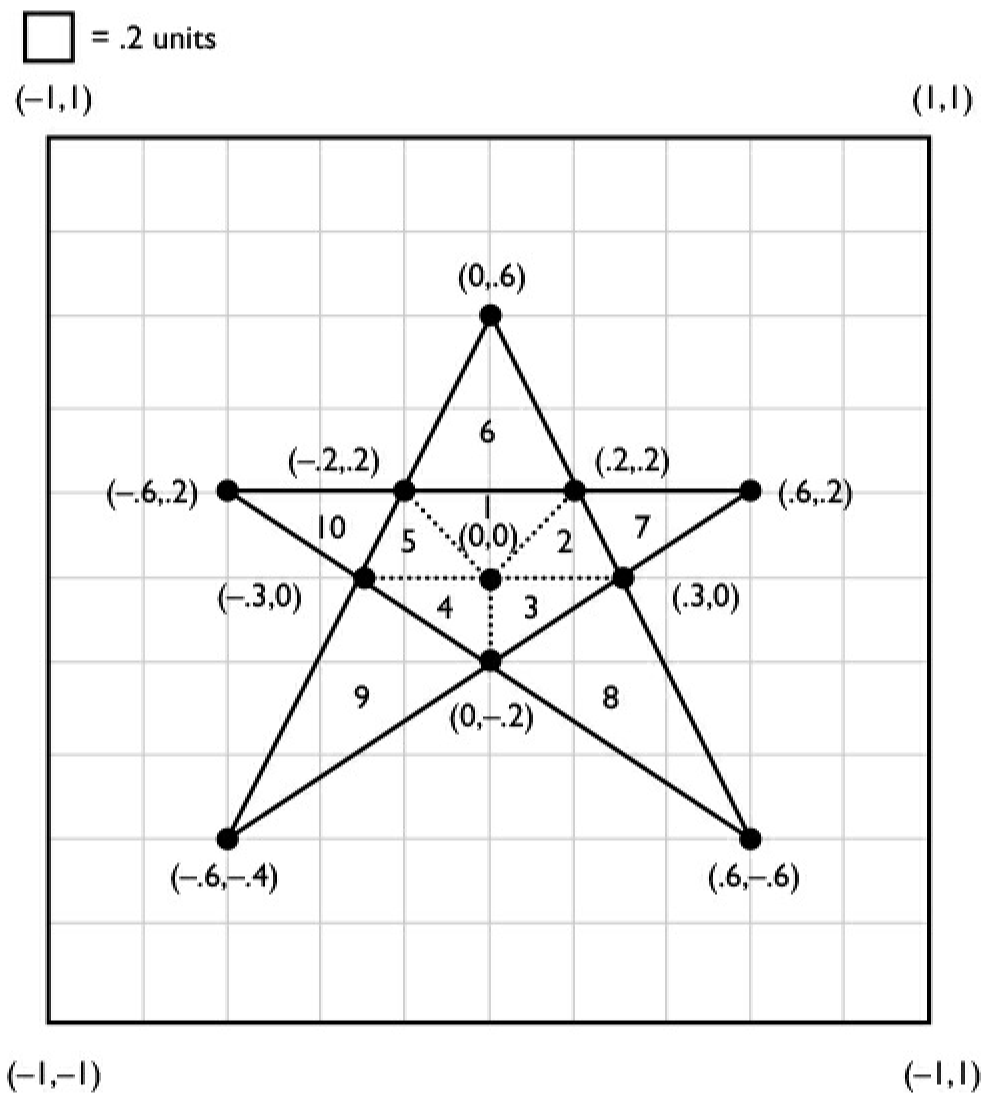
Creating Complex Shapes with Triangles and Meshes

One reason the default project in graphics programming is to render a triangle is because all shapes can be created from triangles. Sometimes, it can be difficult to make the mental transition between creating a triangle to generating a 3D pug model, so we create something slightly more complicated than a triangle but not as complex as a fully rigged human model in Maya.

A five-pointed star, as shown in Figure 8.2, has 11 vertices. There are 5 vertices for the points, 5 for the indentations in the star, and 1 for the middle of the star. The last one isn’t seen, but it’s necessary for creating all the internal triangles within the star.

The star is created from 10 different triangles. The outside 5 triangles are obvious, but what is less obvious are the 5 that come together to create the middle of the star.

Figure 8.2. *A star constructed from triangles and vertices*



First, create an array of floats for the vertex data:

```
let vertexData:[Float] =
    [
        // Internal Triangles
        0.0, 0.0, 0.0, 1.0,
        -0.2, 0.2, 0.0, 1.0,
        0.2, 0.2, 0.0, 1.0,

        0.0, 0.0, 0.0, 1.0,
        0.2, 0.2, 0.0, 1.0,
        0.3, 0.0, 0.0, 1.0,

        0.0, 0.0, 0.0, 1.0,
        0.3, 0.0, 0.0, 1.0,
        0.0, -0.2, 0.0, 1.0,

        0.0, 0.0, 0.0, 1.0,
        0.0, -0.2, 0.0, 1.0,
        -0.3, 0.0, 0.0, 1.0,

        0.0, 0.0, 0.0, 1.0,
        -0.3, 0.0, 0.0, 1.0,
        -0.2, 0.2, 0.0, 1.0,

        // External Triangles
        0.0, 0.6, 0.0, 1.0,
        -0.2, 0.2, 0.0, 1.0,
        0.2, 0.2, 0.0, 1.0,
        0.6, 0.2, 0.0, 1.0,
        0.2, 0.2, 0.0, 1.0,
        0.3, 0.0, 0.0, 1.0,

        0.6, -0.6, 0.0, 1.0,
        0.0, -0.2, 0.0, 1.0,
        0.3, 0.0, 0.0, 1.0,

        -0.6, -0.4, 0.0, 1.0,
        0.0, -0.2, 0.0, 1.0,
        -0.3, 0.0, 0.0, 1.0,

        -0.6, 0.2, 0.0, 1.0,
        -0.2, 0.2, 0.0, 1.0,
        -0.3, 0.0, 0.0, 1.0
]
```

Even though there are 10 triangles and 11 vertices, you need to specify each set of triangles. The first 12 values represent 1 triangle in the middle of the star.

Mapping 2D Vertex Data by Hand

You may wonder from where these vertex values came. Generally, if you're working with a really large and complex model, you would create your model in a CAD program, like Maya or Blender. You then import this pattern file into your project using a framework such as Model I/O.

This pattern isn't complex enough to justify the heavy lifting of one of those programs, so it was done the old-fashioned way. The star was drawn on a piece of graph paper with each square representing 0.2 units within the drawing area.

The star was drawn and the points plotted to get an approximate value of where each vertex should be.

If you want to create interesting or unique 2D patterns, draw them on graph paper to paper prototype your prospective images.

Because the goal is for the tips to be red and the rest of the star to be white, specify the color data for the vertices as well.

```
let vertexColorData:[Float] =
    [
        // Internal Triangles
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,

        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,

        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,

        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,

        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        // External Triangles
        1.0, 0.0, 0.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,

        1.0, 0.0, 0.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,

        1.0, 0.0, 0.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,

        1.0, 0.0, 0.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0,

        1.0, 0.0, 0.0, 1.0,
        1.0, 1.0, 1.0, 1.0,
        1.0, 1.0, 1.0, 1.0
]
```

All the internal triangles have all of their vertices colored white. In the external triangles, only the points are colored, so those values are rejected in the RGB values of the vertices.

There is a much more efficient way of doing this, which is detailed in Chapter 11, “Interfacing with Model I/O.” For learning purposes, this will do for the time being.

Metal Primitive Types

You’ve entered all of your vertex data, one way or another. You know that your large number of floats represent triangles that make up your star, but how does the GPU know that?

Metal has a set of primitive objects that are included in the MTLPrimitiveType enum:

• Point

• Line

• Line strip

• Triangle

• Triangle strip

Everything you construct in Metal is composed of points, lines, and triangles. Lines and triangles can be combined to create complex strips and meshes.

• *Points* are represented by a single vertex. It is the only primitive type that can’t be combined in a strip, because a strip of points is, by default, a line.

Along with the position of the vertex, the size of the point must be specified when the point is passed to the vertex shader.

• *Lines* are technically line segments and not lines, which go on continuously. A line is defined by two vertices. If you enter multiple lines in the command buffer, they will not be connected.

If you want your lines to be connect, you create them as a *line strip*. You enter all of the vertices in the buffer for the strip in the order you would like them to be drawn, and the vertex shader will connect them.

The most complex shape, and the one you will be working with most often, is the triangle.

• A *triangle* is defined by three vertices. In the Star example, each of the 10 triangles that compose the star are defined independently of one another.

Since the middle 5 triangles share two sides with one another, this part of the shape could be refactored to create a triangle strip.

• In a *triangle strip*, each of the triangles in the strip shares at least one side with another triangle. Each triangle in the middle of the strip connects to the triangle before and the triangle after.

Responding to MTKViewDelegate Methods

This project utilizes MetalKit. When Metal was first introduced, there was a lot of boilerplate code that you, the programmer, were responsible for. Among other things, you needed to remember to import the Quartz Core framework in order to create a subclass of CALayer for Metal.

Rather than remembering all this stuff and writing the same code over, this process was streamlined with the creation of MetalKit. MetalKit provides a CALayer, Metal-aware view, MKView.

To use MetalKit, you need to make your UIViewController conform to the MTKViewDelegate protocol. This protocol is fairly straightforward. It has two responsibilities:

• Changing the view’s layout

• Drawing the view’s content

drawableSizeWillChange() is responsible for responding to an impending change in drawable dimensions whenever the application notifies it that there has been a change in layout, resolution, or size. This method is required even if you leave it empty.

draw() does exactly what it says it does. It is responsible for drawing the view’s contents.

For draw() to work, it needs to be able to retrieve a *drawable*, which is covered next.

Retrieving a Drawable

A MTLDrawable is an object that represents a displayable resource to which drawing commands can be sent after being rendered. It is the last step you need to take after you finish encoding the render encoder, which is covered later in this chapter.

After you set up all of the commands that you want to send to the renderer, you need someplace to send the results. That is the MTLDrawable.

The only method on this protocol is present(). You can present immediately, or else you can pass a given time along as a parameter.

MTLDrawable exists as a property on the MTKView, so you don’t directly initialize it on your own. Instead, you get a reference to a drawable by retrieving it from your MTKView’s currentDrawable property:

```
if let renderPassDescriptor = view.currentRenderPassDescriptor,
    let currentDrawable = view.currentDrawable {
}
```

After you are finished encoding all of your commands to the render encoder, you call present():

```
commandBuffer.present(currentDrawable)
```

Creating a Command Buffer

The MTLCommandBuffer is a set of commands to be executed by the GPU. All of the objects you need to arrange and schedule go through the command buffer. It’s the cruise director for the rendering pass in Metal. If you want to have something happen on the rendering pass, the command buffer has to reserve space for it in the current command queue to make sure not only that it gets done but also that it gets done in the proper order.

The MTLCommandBuffer is responsible for creating the command encoder. There are four flavors of command encoder:

• MTLRenderCommandEncoder

• MTLComputeCommandEncoder

• MTLBlitCommandEncoder

• MTLParallelRenderCommandEncoder

At any given point in time, only one command encoder can be active for the command buffer. It’s possible to create multiple command encoders to add tasks to the buffer, but you need to end the encoding process for the current encoder before you can create another one.

To add tasks to the command queue, you need to enqueue them using the enqueue() method. You enqueue as many tasks as necessary, then you signal the compiler that you’re done by calling the commit() method. Calling commit() tells the command queue that the command buffer won’t have any additional commands encoded into it and is ready for scheduling.

Creating a Command Encoder

An application’s MTLCommandEncoder is created using a factory method on the command buffer. Its job is to translate between the API and the actual commands that are executed by the GPU. It doesn’t send anything to the GPU itself, it’s simply coordinating work that is sent to the GPU. This work could be rendering, computation, or copying data between resources.

You must take numerous steps to set up a rendering pass with the command encoder:

1. Create a MTLRenderCommandEncoder to configure your rendering commands.

2. Call the setRenderPipelineState() method to specify a MTLRenderPipelineState. The render pipeline state object contains various state that affects rendering. Being immutable, it doesn't really track anything over time.

3. Specify resources for input and output to these vertex and fragment functions.

4. Specify any other fixed-function states.

5. Draw the primitives.

6. Call endEncoding() to terminate the render command encoder.

Because this is a drawing application, the one covered in this chapter is the MTLRenderCommandEncoder. You get a new command encoder every time you call makeRenderCommandEncoder, and the command buffer doesn't store it afterwards. It can be created like this:

```
let renderEncoder = commandBuffer.makeRenderCommandEncoder(
                        descriptor: renderPassDescriptor)
```

A render command encoder is the middleman that translates from API calls to GPU-native commands. The render pass descriptor describes the set of textures to be rendered by a render pass—namely, the render pass to be performed by the render command encoder that's being created here.

For more information on the render pass descriptor, see "Render Pipeline Descriptors and State" in Chapter 7, "Libraries, Functions, and Pipeline States."

For this simple project, you create the following attachments:

```
let renderEncoder = commandBuffer.makeRenderCommandEncoder(
                        descriptor: renderPassDescriptor)
renderEncoder.label = "render encoder"

renderEncoder.pushDebugGroup("draw morphing star")
renderEncoder.setRenderPipelineState(pipelineState)
renderEncoder.setVertexBuffer(vertexBuffer,
                               offset: 256*bufferIndex, at: 0)
renderEncoder.setVertexBuffer(vertexColorBuffer,
                               offset:0 , at: 1)
renderEncoder.drawPrimitives(type: .triangleStrip,
                              vertexStart: 0,
                              vertexCount: 10,
                              instanceCount: 1)

renderEncoder.popDebugGroup()
renderEncoder.endEncoding()
```

After creating the render encoder, you're specifying that render pipeline state is the one you created for the rest of the project. If you need a refresher on what the pipeline state entails, refer to "Libraries, Functions, and Pipeline States" in Chapter 3, "Your First Metal Application (Hello Triangle!)."

After that, you need to specify the resources for your vertex and fragment functions. There are two vertex buffers, one for the locations of the vertices and one for each vertex's colors, so you need to set both of those in the code.

The last thing you need to pass to the encoder is the primitive you need to draw. You take the triangle strip you created at the beginning of the project and pass it into the render queue to be drawn.

Now that you’re done sending commands to the encoder, you need to let it know that it has received its last command by calling endEncoding().

Fixed-Function State on the Command Encoder

If you were around during the days of OpenGL ES 1.1, you know that OpenGL EW used to have what is known as a *fixed-function pipeline*. There were no fragment or vertex shaders. Recall that you already set a lot of this state on the render pass descriptor, and it now exists prevalidated in the pipeline state object. However, some state can be set more flexibly than that by setting certain properties on the render command encoder before issuing a draw call. You may be under the impression that these functions went the way of the dinosaur, but that’s not the case.

MetalKit implements and uses some of these states to make it easier to do simple things.

If you run the Metal Template directly after creating it in Xcode, you see a floating triangle in your view. The code doesn’t set any of these states explicitly, so why does the shading still work?

The renderPassDescriptor convenience method creates a MTLRenderPassDescriptor object with color, depth, and stencil attachment properties with default attachment state. You get these states for “free” even if you never set them.

MTKView manages a set of renderable textures (including a color render target and depth render target) that it sets on the render pass descriptor it generates for you. Also, these fixed-function properties aren’t set on the render pass descriptor but on the render command encoder itself.

• setBlendColor

• setCullMode

• setTriangleFillMode

• setFrontFacing

• setScissorRect

• setRenderPipelineState

Some of these states, such as the blend color, allow you to customize your drawing somewhat. Some of them, such as front facing, allow you to optimize your code by not drawing objects that will never be seen by the user.

One that you must use is setRenderPipelineState(). You need to ensure that all of the work you’ve done setting up your render pipeline state is associated with the command encoder. Setting the render pipeline state tells the render command encoder which vertex and fragment function to draw with, among other prevalidated state.

Passing Data to Shaders

The next step in the rendering process involves sending data to the shaders.

Each graphics program needs to have two different kinds of shaders: *vertex* and *fragment*. These two programs are two sides of the same coin. They’re not separate components, they are partners working together to render images to the screen.

The vertex shader is responsible for the color and position of each vertex in your program. It reads from and acts on the data in the buffers you created earlier in this chapter from the vertices you put in your arrays. Those vertices describe the position and color of each vertex.

The vertex shader takes that information and processes it before passing it on to the rasterizer. The rasterizer looks at all of the triangles and colors described with the vertices and converts it to pixels that will appear on the screen after vertex processing and primitive assembly.

This pixel data is then fed to the other shader in your program, the fragment shader. The fragment shader is responsible for doing any processing that needs to be done on a pixel level. The chief output of the fragment shader is the color of the fragment it’s processing.

For example, if you were converting an image from color to black and white, the vertex shader would pass each pixel of the image to the fragment shader, and the fragment shader would perform a calculation on the color data passed in to determine how to change it so that the image can have the color removed.

Shaders are powerful tools that allow you to do a lot of amazing things, but before they can perform any of their awesome shader magic, they need to be fed a steady stream of data.

In the star project, the data is being fed to the shaders through a vertex buffer, but that is not the only way to feed the shaders.

Shaders can take three different types of data:

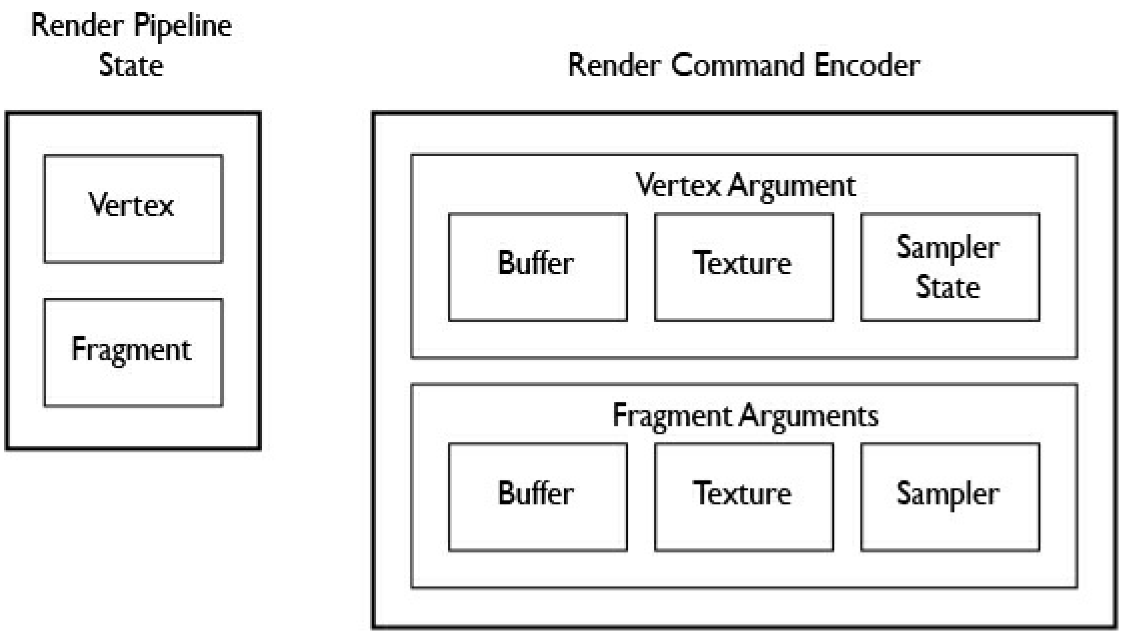
• **Buffers**: Conforms to the MTLBuffer protocol. MTLBuffers are unformatted blocks of memory that can contain any type of data.

• **Textures**: Conforms to the MTLTexture protocol. MTLTextures are formatted image data.

• **Sampler state**: Conforms to the MTLSamplerState protocol. Sampler states contain properties that influence how textures are sampled.

To use resources in your shader, you need some way to dynamically map between the resource objects and the shader function parameters. This map is called the *argument table*. You can have multiple buffers and a combination of buffers, textures, and sampler states, as shown in Figure 8.3.

Figure 8.3. *Argument table communicating between the command buffer and the shaders*



The argument table has a finite number of slots for each resource type, which limits how many resources of each type can be bound at once:

• **Buffers**: Upper limit of 31 entries

• **Textures**: Upper limit of 31 entries

• **Sampler state**: Upper limit of 16 entries

The argument table is zero indexed, so the first buffer in the argument table would be in slot 0.

The star project has two vertex buffers, one for position and one for color. In the encoder side of things, take a look at the code setting these buffers:

```
renderEncoder.setVertexBuffer(vertexBuffer,
                              offset: 256*bufferIndex,
                              at: 0)
renderEncoder.setVertexBuffer(vertexColorBuffer,
                              offset:0 , at: 1)
```

The first vertex buffer is placed in the first slot of the buffer argument table. The vertex color buffer is inserted in the next slot and is at index 1.

The encoder has done its job and has set the buffers in the arguments table. Now it is the vertex shader’s job to access those arguments and process them.

When you write your shaders, you need to be aware of what arguments the argument table will be sending over. It’s like a lock and key. If the shader doesn’t receive the arguments it is expecting, then it won’t work. You need to add arguments to the method signature that match the objects in the argument table:

```
vertex VertexInOut passThroughVertex(uint vid [[ vertex_id ]],
            constant packed_float4* position  [[ buffer(0) ]],
            constant packed_float4* color    [[ buffer(1) ]])
```

For a more comprehensive explanation of shaders and the rasterization process, please refer to Chapter 2, “An Overview of Rendering and Raster Graphics.”

Issuing Draw Calls

After all of the commands are encoded in the command encoder, something needs to trigger the rendering and rastering process, and in Metal, that event is a *draw call*.

Draw calls in Metal need to instruct the GPU what MTLPrimitive type is being drawn and where in the vertex buffer to begin and end the drawing.

```
func drawPrimitives(type primitiveType: MTLPrimitiveType,
    vertexStart: Int,
    vertexCount: Int)
```

If you wanted to create one star instance, you would call this method, and it would look something like this:

```
renderEncoder.drawPrimitives(type: .triangle, vertexStart: 0,
                             vertexCount: 29, instanceCount: 1)
```

If you wanted to draw an outline for the star, you could do another call to drawPrimitives but specify a line rather than a triangle. If you wanted to streamline the process of creating your middle set of triangles, you could create them as a triangle strip. There are many options available to you to play around with in these primitive types.

There are also different types of draw calls. You can use instanced drawing, which is explained in Chapter 10, “Advanced 3D Drawing.”

Scheduling and Enqueuing Command Buffers

One of the responsibilities of the driver is to schedule work with the GPU. All the set up you have been doing so far has been to arrange with the command buffer what work needs to be done. Once the command buffer has its orders, it needs to schedule time with the GPU to execute the commands that are encoded into it.

Once you’re done encoding all of the commands you intend to place in the command buffer, the buffer needs to be scheduled with the command queue. This process is known as *enqueuing*.

Enqueuing a command buffer reserves a place for the command buffer on the command queue without committing the command buffer for execution. When this command buffer is later committed, it is executed after any command buffers previously enqueued on the command queue.

The command buffer has a series of built-in block commands to send work out to the GPU.

The easiest way to start a numbered list is to copy/type over this existing list or insert the boilerplate and type over it. Lead into lists with a phrase or sentence. If you use a complete sentence, end it with a colon. If you use a phrase but not a complete sentence, don’t use a colon. Numbered lists should be used for linear or ordered material, such as a list of steps. Here is an example:

```
(void)addCompletedHandler:(MTLCommandBufferHandler)block;
```

When the completed handler is called, it signals the application that the shaders are done and the frame has completed.

Getting Notification of Frame Completion

The command buffer has a few different states that it can be in at any given time:

- Enqueued in the command queue
- Committed to the GPU
- Scheduled to be completed
- Work completed on the GPU

This status exists as a property on the command buffer. There is also an error property. If the commands are executed without incident, the error property is nil. If something goes wrong, the error property is set to a value that is listed in the Command Buffer Error Codes list to give you an indication about what has gone wrong.

Presenting to the Screen

The final step in this process is to present to the screen. The object that the command buffer and the MTKView share is the MTLDrawable. When the render pass descriptor was created, the current drawable was set to the property on the view.

All the work on the GPU was being performed on this drawable, and now that the work is completed, it’s time to give that drawable back to the view so that it can be rendered to the screen:

```
commandBuffer.present(currentDrawable)
```

The MTKView takes that drawable and uses the texture associated with it to present the image to the screen.

Summary

All shapes are composed of three primitive types: points, lines, and triangles. Lines and triangles can be combined into strips to create more complicated meshes.

The command queue is the object that schedules and manages work done on the GPU. The command queue organizes a series of command buffers. These command buffers package work to be sent to the GPU, which can include drawing, computation, and copying without touching the CPU. The render encoder issues draw calls, and when these have finished executing, they are presented to the screen.

9. Introduction to 3D Drawing

The key to growth is the introduction of higher dimensions of consciousness into our awareness.

—Lao Tzu

Every introduction to Metal starts with rendering a 2D shape composed of triangles. It’s a good way to present the moving parts of a framework to make an image appear on the screen. When you go from a contrived 2D project to a real-world 3D project, you introduce complexity that is inherent to all 3D projects, regardless of framework. This chapter introduces you to concepts you need to understand when you jump from two dimensions to three.

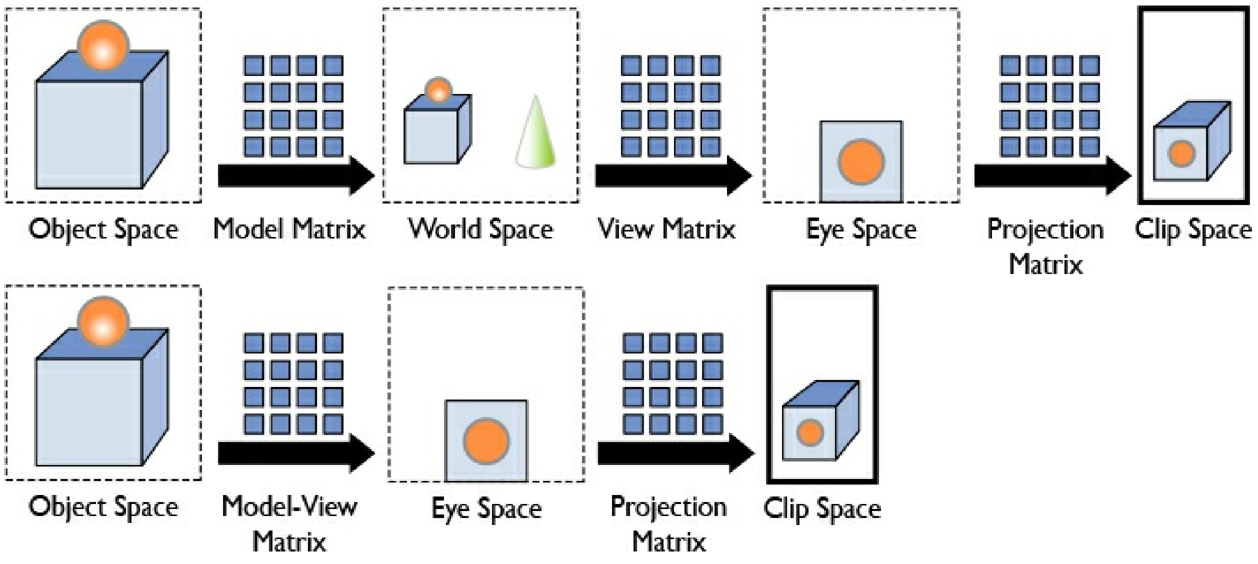
Model-View-Projection Transformations

An important aspect of 3D graphics is being able to represent an object in space in relation to a camera. If you think about this in the real world, your eyes are your camera. If you walk around a store, the shelves move through your field of view as you work your way through. If you are playing basketball and you are looking for someone to pass the ball to, the other players move in space around you. As you pass the ball, the ball moves away from the “camera.” Part of the challenge in 3D graphics is finding a way to express these realities in a virtual world.

You need to be able to crawl before you can walk. The following sections deal with how to present a virtual world from the perspective of a camera. Later sections deal with movement and overlapping objects. Before you can understand those concepts, you need to have a good grasp on what gets presented to the screen. The mathematical formulas for these transformations are described in Chapter 4, “Essential Mathematics for Graphics.”

All of your vertex data starts out in object space. Object space is concerned with how each vertex relates to the others within their own context. If you look at the position data in your vertex file, it describes the positions in object space. The vertex shader, using the model matrix, takes the vertex data and assembles an object in object space and then hands it to the next step in the process, which is world space. Figure 9.1 illustrates the transition and pipeline.

Figure 9.1. *Moving from object space to eye space to clip space*



Complex 3D scenes could have hundreds of different objects in a scene. Each object needs to be expressed as a position in relation to the origin in the world space. In one of Apple's sample projects, there are hundreds of rotating cubes floating in space. These cubes are all instances of the same object, but they have their own unique properties, such as color and position. The purpose of world space is to compose these objects within this space the same way you positioned your vertices to compose your objects.

In Chapter 4, you were introduced to transform matrices. The matrices are used to reposition the vertices from object space to world space. The model file describes how each vertex is positioned in object space. When a vertex passes through the vertex shader, the transforms described in the model file are applied to the vertex to position it properly within world space. This task is done using the view matrix.

The view matrix translates objects from world space to eye space. This translation reorients the origin to the camera position from the middle of the object. Often, the model matrix and the view matrix are combined into a model-view matrix. Even though the model-view matrix is a necessary component for any production Metal program implementing 3D, Metal does not include a built-in instance of one. It doesn't presume to know how you would choose to position your objects, so it leaves it up to you to roll your own. The following implementation is just one example of how you can implement a model-view matrix:

```
struct Matrix4x4
{
    var X: Vector4
    var Y: Vector4
    var Z: Vector4
    var W: Vector4

    init()
    {
        X = Vector4(x: 1, y: 0, z: 0, w: 0)
        Y = Vector4(x: 0, y: 1, z: 0, w: 0)
        Z = Vector4(x: 0, y: 0, z: 1, w: 0)
        W = Vector4(x: 0, y: 0, z: 0, w: 1)
    }

    static func rotationAboutAxis(_ axis: Vector4,
              byAngle angle: Float32) -> Matrix4x4
    {
        var mat = Matrix4x4()

        let c = cos(angle)
        let s = sin(angle)

        mat.X.x = axis.x * axis.x + (1 - axis.x * axis.x) * c
        mat.X.y = axis.x * axis.y * (1 - c) - axis.z * s
        mat.X.z = axis.x * axis.z * (1 - c) + axis.y * s

        mat.Y.x = axis.x * axis.y * (1 - c) + axis.z * s
        mat.Y.y = axis.y * axis.y + (1 - axis.y * axis.y) * c
        mat.Y.z = axis.y * axis.z * (1 - c) - axis.x * s

        mat.Z.x = axis.x * axis.z * (1 - c) - axis.y * s
        mat.Z.y = axis.y * axis.z * (1 - c) + axis.x * s
        mat.Z.z = axis.z * axis.z + (1 - axis.z * axis.z) * c

        return mat
    }
}
```

This struct represents a matrix object. It’s initialized with an identity matrix, which is a safe default value for a matrix in computer graphics. You then change the defaults to values that calculate an axis as an angle of rotation based on an axis and an angle passed in by the programmer. This action resets the identity matrix to create a new matrix that provides the vertex shader the information necessary to translate the object from object space to eye space.

To implement these changes, you need to choose a rotation angle and a rotation axis:

```
var rotationAngle: Float32 = 0
let yAxis = Vector4(x: 0, y: -1, z: 0, w: 0)
var modelViewMatrix = Matrix4x4.rotationAboutAxis(yAxis,
                                 byAngle: rotationAngle)
```

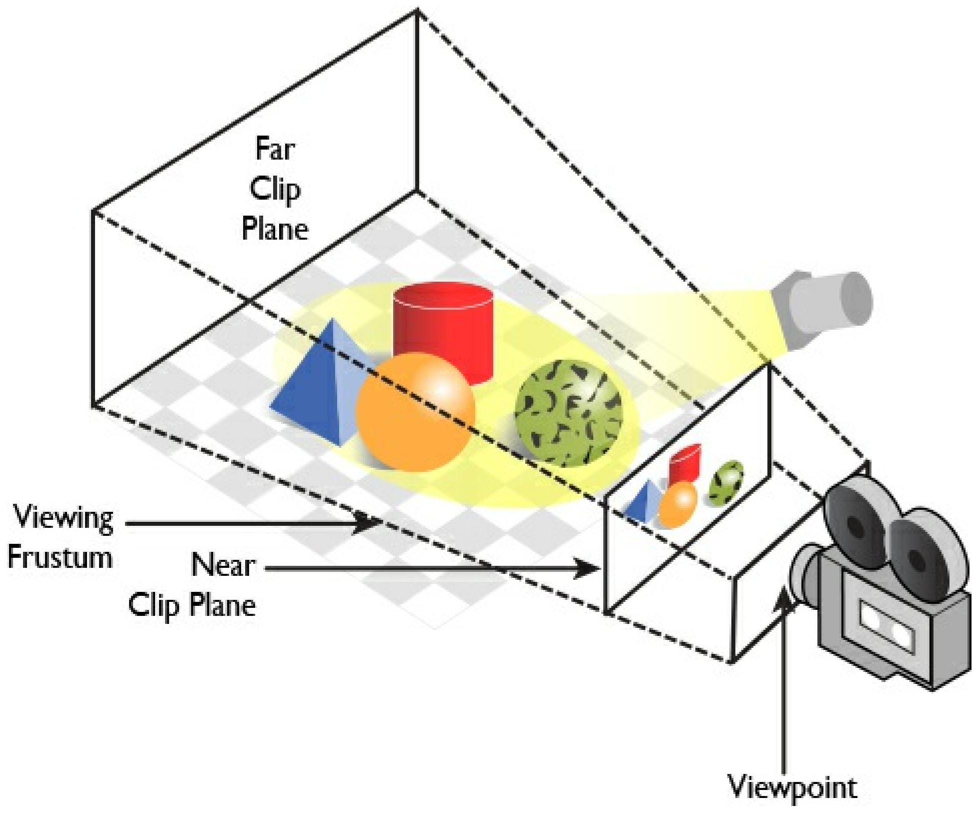
GLM and Other Math Libraries

Most people who do OpenGL don’t roll their own matrix functions. There are numerous open source libraries, such as OpenGL Mathematics (GLM), that work well with the OpenGL Shading Language (GLSL) and implement these functions for you. Although these libraries do not currently work directly with Metal, they are open source, and the code is in C/C++. They are a valuable resource to draw on while you’re getting your feet under you with graphics programming—they can assist you in much the same way that translating GLSL code to Metal is helpful as you create your first shader programs. If you feel particularly ambitious, you could do your own open source project based on the equations found in the math libraries.

Clip Space and the View Frustum

Now that you’ve successfully converted the origin to world space, you need to determine what objects actively appear on the screen. Figure 9.2 illustrates how the view frustum determines what is within clip space. Think back to the store walkthrough example. As you walk through the store, you pass shelves. Those shelves exist in relation to one another in a way that does not change based on where the camera is oriented, so you have two separate origins. Those shelves don’t cease to exist when they’re not actively in your field of view, but it would be a terrible waste of processing time to render objects that are not currently visible. You have, on average, a mere 16 milliseconds to render each frame. You want to dedicate every one of those milliseconds to something that will be seen. To do so, you need yet another matrix to determine whether or not the current objects are visible. This matrix is the *projection matrix*.

Figure 9.2. *The view frustum converting objects from eye space to clip space*



There are two different ways you can implement your projection matrix:

• **Orthographic** discards all depth perception from your scene and renders it as purely 2D. Although orthographic projection is an option, you're more likely to want to create a perspective projection matrix.

• **Perspective** projection matrices render objects that are further away as smaller and maintain the general feel of your constructed world space. An example projection matrix follows:

```
struct Matrix4x4
{
    var X: Vector4
    var Y: Vector4
    var Z: Vector4
    var W: Vector4

    init()
    {
        X = Vector4(x: 1, y: 0, z: 0, w: 0)
        Y = Vector4(x: 0, y: 1, z: 0, w: 0)
        Z = Vector4(x: 0, y: 0, z: 1, w: 0)
        W = Vector4(x: 0, y: 0, z: 0, w: 1)
    }

    static func perspectiveProjection(_ aspect: Float32,
                                      fieldOfViewY: Float32,
                                      near: Float32,
                                      far: Float32) -> Matrix4x4
    {
        var mat = Matrix4x4()

        let fovRadians = fieldOfViewY * Float32(M_PI / 180.0)

        let yScale = 1 / tan(fovRadians * 0.5)
        let xScale = yScale / aspect
        let zRange = far - near
        let zScale = -(far + near) / zRange
        let wzScale = -2 * far * near / zRange
        mat.X.x = xScale
        mat.Y.y = yScale
        mat.Z.z = zScale
        mat.Z.w = -1
        mat.W.z = wzScale

        return mat;
    }
}
```

The projection matrix requires four pieces of information to calculate properly:

• **Aspect**: The ratio of the width divided by the length. Common aspect ratios are 4:3 and 16:9. The ratios change depending on the dimensions of the Metal drawable.

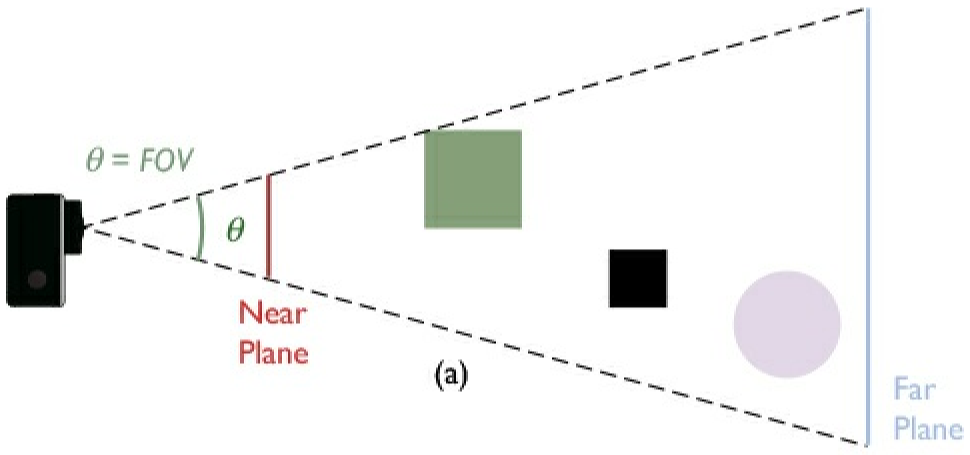
• **Field of view for the y-axis**: How much “zoom” is applied to the camera.

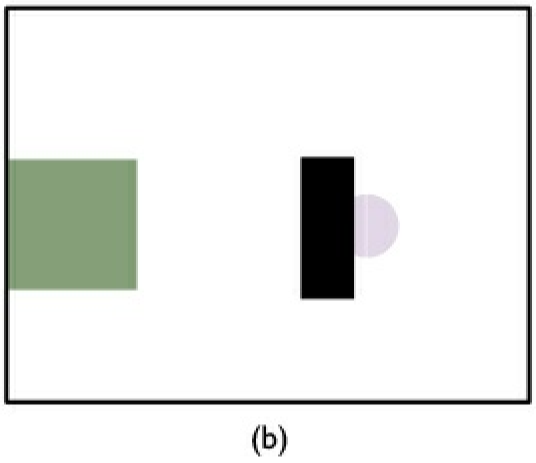
• **Near plane**: Value given to the nearest plane (usually close to 0). Anything between the near plane and the camera is not drawn.

• **Far plane**: Value given to the “back” of the view frustum. Pretend you’re looking at a room. The far plane represents the back wall of your view frustum. Anything behind the far plane is not drawn.

To get a decent projection matrix, as shown in Figure 9.3, you need to get the aspect ratio of the current drawable and pass it into the projection matrix.

Figure 9.3. *An accurate perspective matrix*





```
let aspect = Float32(metalLayer.drawableSize.width) /
Float32(metalLayer.drawableSize.height)

let projectionMatrix = Matrix4x4.perspectiveProjection(aspect,
                                             fieldOfViewY: 60,
                                                    near: 0.1,
                                                   far: 100.0)
```

Now that you have a scene oriented in world and camera space, you need to apply the most important thing in computer graphics programming to it: lighting.

Shading Models

Lighting is one of the most important concepts in visual media. It's the difference between an image looking fantastic and an image looking terrible.

Think about various pictures you have of yourself. Pictures taken in a dark bar by someone who has been drinking can make you look decades older and 20 pounds heavier. A good photographer lights your face in a flattering way. Without lighting, your model won't appear on the screen. Having a good understanding of various lighting algorithms and techniques is vital to being a successful graphics programmer. It's also important to understand the performance cost of your algorithms and to weigh the costs and benefits of implementing each type.

Flat Shading

The easiest and least computationally expensive shading method is flat shading. The reason that it's cheap and easy is the same reason anything in life is: It's not very good. Flat shading is used to shade each polygon of an object based on the angle between the polygon's surface normal and the direction of the light source. As a result, each face is uniformly shaded the same color. If a corner of the face is close to a light source, the entire face is illuminated despite not behaving in a physically realistic manner.

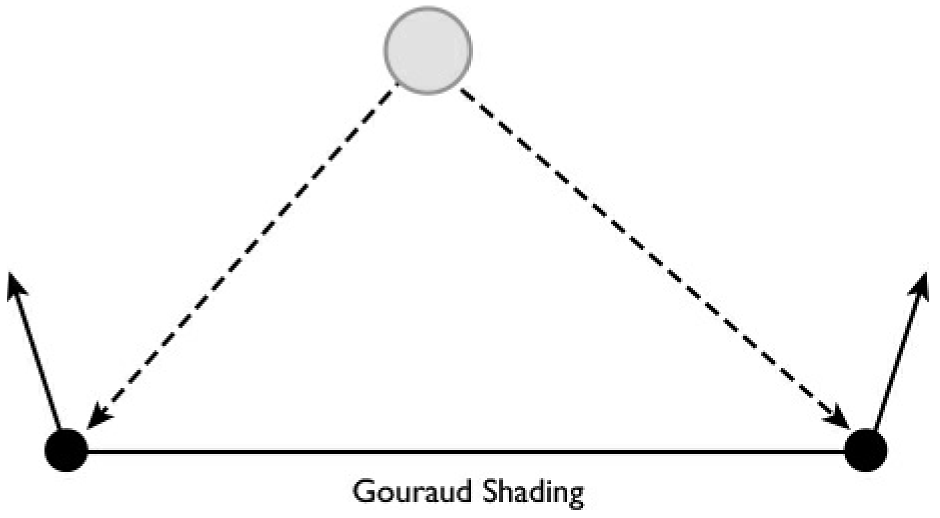
Of all of the shading methods available to you, flat shading is the least realistic and the worst looking. Flat shading is a characteristic of arcade video games and early home video-game systems. It was the predominant shading system until the early 1990s. Use flat shading if you want your graphics to have a retro feel.

Gouraud Shading

In the early 1990s, flat shading was largely replaced by Gouraud shading. Gouraud shading was the first prominent smooth-shading method. If you look at flat shading, there are noticeable faces for every shape. Gouraud shading smooths out these shapes and allows you to create better-looking shapes without having to include thousands of polygons.

Gouraud shading depends on interpolation, as shown in Figure 9.4. The algorithm takes the illumination at each vertex, and it weighs how close each vertex is to the point it is shading and assigns it a weight. It then calculates how much illumination the point should have based on where it is in relation to the light source. The shader you implemented in Chapter 5, "Introduction to Shaders," used Gouraud shading because the illumination was calculated per vertex.

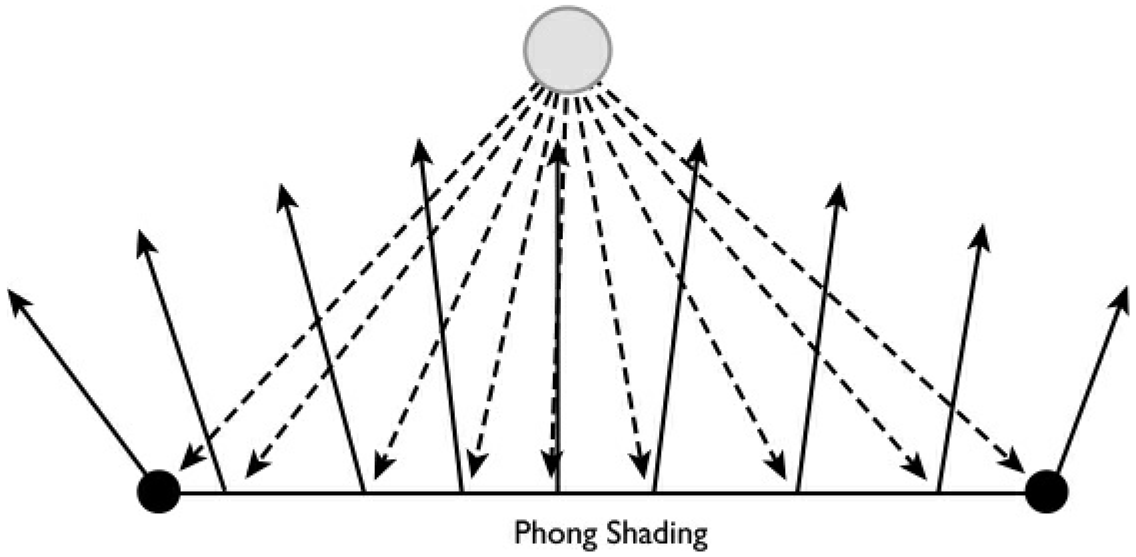
Figure 9.4. *Linear interpolation of illumination in Gouraud shading*



Phong Shading

Phong shading is even better than Gouraud shading. Rather than calculating the illumination per vertex, it calculates the illumination per fragment, as illustrated in Figure 9.5. Phong shading also depends on linear interpolation, but it doesn't just depend on the illumination at each vertex. It computes pixel colors based on interpolated normals and a reflection model. Phong shading uses linear interpolation as opposed to interpolation across polygons.

Figure 9.5. *Linear interpolation of illumination in Phong shading. This method is more accurate but requires more processing.*



Phong shading is better than Gouraud shading, but it's also more processor intensive. Phong shading specifically does far better than Gouraud shading in regard to reflection models that have small, specular highlights. If you have particularly large polygons and there is a specular reflection that hits in the middle of the face and not at any of the vertices, the Gouraud shading model will not register that there is a highlight to be rendered, and the highlight will be completely lost.

Basic Lighting

There is more to lighting than just knowing whether it is calculated on a vertex or a fragment basis. There are several different types of lights that generate completely different effects. Most complex modeled scenes in AAA video games or animated movies are composed by dozens, if not hundreds, of instances of all of these kinds of lights. It's not an either/or choice. It's how many of each type of light you need to create the effect that you are looking for.

Ambient

Ambient light is generally what you think of when you think about lighting. It applies an even amount of light to all sides. It doesn't cast shadows—it just adds a base level of illumination to all objects. This is helpful if you just want to see what is going on in your scene, but it's not particularly interesting.

Directional Diffuse

Ambient light is light that has bounced around multiple surfaces before creating a general illumination. However, that is not how all light behaves in the real world. In the real world, you have light sources such as the sun or a light bulb. These light sources will hit your object before bouncing off to create ambient light, and you need to understand how they affect the look of your object.

Diffuse lights tend to have a soft, fuzzy feel to them. Think about how it would look if you pointed a flashlight at a tennis ball. A tennis ball has a fuzzy, irregular surface. That surface causes the rays of light to scatter in a chaotic way. This creates a soft lighting effect rather than a sharp effect that you would see on the surface of a mirror.

Directional Specular

Diffuse and specular light are similar, yet they are subtly different. Both deal with strong point lights, but they react to them differently. Diffuse light scatters like on the surface of a tennis ball. Specular light reacts like a shiny surface. Picture a metal ball. If you shine a flashlight on the ball, the surface reflects most of the light in a direct and harsh manner, unlike the soft manner in which the tennis ball reflects it.

Think of the lens flares in every J.J. Abrams movie. The camera pans past a light source. The light briefly flares across the screen and blocks out all other objects on the screen. This is an example of a specular light source. It is a bright shiny point of light that passes through the camera's field of view.

Lighting is about more than just position. It's about how to simulate the way light reacts in the natural world. By determining how specular or diffuse your lighting source is for your objects, you are signaling what type of material they could be created from.

Animation

Movement is an integral part of interactive computer graphics. If nothing in your scene moves, it's nothing more than a static piece of art. Users will interact with your application, and it will respond to them. This involves movement, and it's important to understand how that works. This sections is by no means comprehensive. There are books that are hundreds of pages long describing computer animation. This section provides just a brief overview of what you need to know to get up and running.

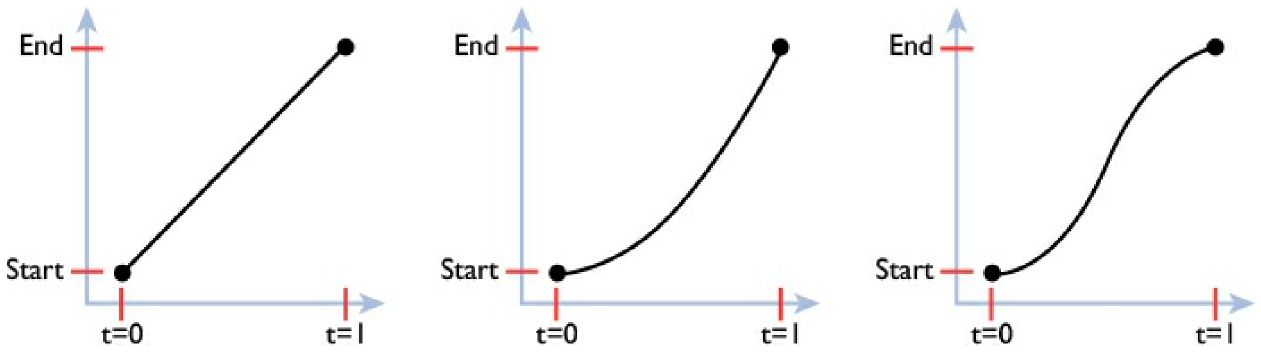
The simplest type of animation you can implement is frame-based animation. In most games, there is a game loop. This loop gets called on every frame, and all of your game logic, animation, and so forth, must be executed in a set amount of time to maintain a decent frame rate. If you place your animation call within this loop, your animation will be called consistently. However, problems can occur if your loop slows down or runs inconsistently. Your animation will run much more slowly on older and less powerful machines, which means your users will have radically different experiences. Your character movement can also look uneven if you tie your animation calls to the game loop.

A more consistent way to do animation is through interpolation, which involves keyframes. A *keyframe* is a point during the animation where something important happens. Pretend you are animating a car. The frame where the car starts moving and the frame where the car stops moving are the keyframes of the animation. To ensure a consistent animation between those two events, an algorithm calculates how far the car must move per frame between the keyframes. The process of computing these intermediate frames is called *tweening*.

A car is a physical object. It takes the car some time to get up to speed. Likewise, it takes the car some time to slow to a stop. It would be an unnatural animation if the car were to immediately go from 0 to 60 or from 60 to 0. To gradually increase or decrease movement, you use easing equations, illustrated in Figure 9.6. To get the car up to speed, you would use a quadratic ease-in.

To slow the car down, you would use an ease-out. Easing equations enable you to emulate natural-looking motion.

Figure 9.6. *Easing in and easing out of animation*



Summary

When you go from 2D to 3D, you have a lot of information to consider. You have to calculate the aspect ratio of your screen and convert your models from object space to world space to clip space. You must consider how you want to light your scenes and weigh the cost and benefits of different shading models. Additionally, you must determine the animation approach that will work best for your graphics.

10. Advanced 3D Drawing

Insanity: doing the same thing over and over again and expecting different results.

—Albert Einstein

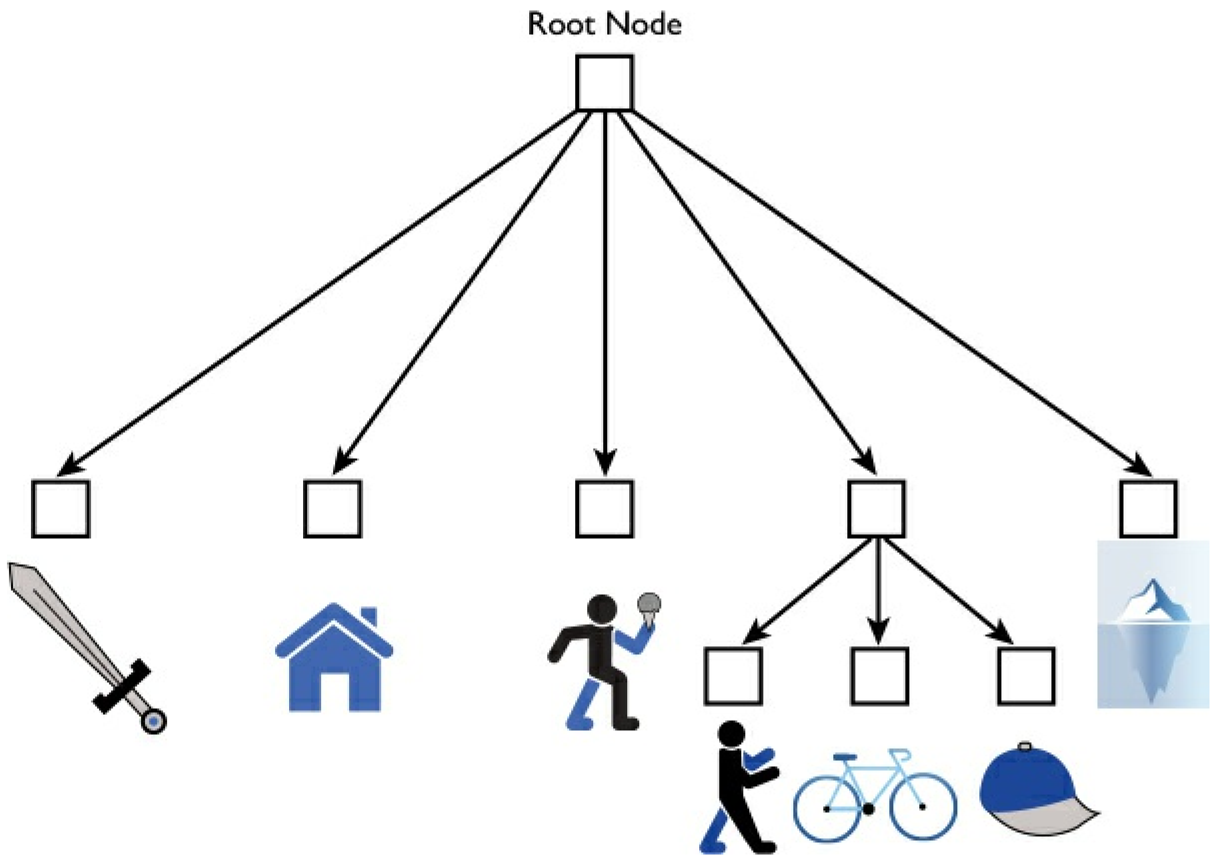
So far in this book, you have rendered only a single object, and you may be feeling a bit daunted. If you have a full 3D project, such as the Zen Garden project that debuted with Metal, you may be thinking you'd have to add millions of lines of code to get a project that complex working. Don't worry—Metal has you covered. This chapter introduces you to the concepts you need to understand to create large and complex 3D scenes.

Constructing a Hierarchical World with Scene Graphs

When you have thousands of objects in a scene, the first thing you must decide is how you will organize them. The most manageable way to organize objects is to implement a scene graph.

A *scene graph* is a hierarchical data structure (see Figure 10.1). If you've worked with the SpriteKit framework, you know that it implements a scene graph structure. The SKScene node owns, and is responsible for, all the objects underneath it. You commonly have situations with characters and multiple parts associated with each character. You can make one node of character the parent of all the other nodes. This parent node is then added to the scene graph and is treated as one object by the graph instead of as a collection of multiple objects. This makes it easier to consistently transform all the nodes in space without applying the same transform to every member of the object. On a micro level, it's possible to translate each individual node within its own relative object space.

Figure 10.1. *Scene graph*



Scene graphs are an elegant way to deal with many objects that need to respond to an overall world space and a localized object space in relation to other objects.

Instanced Rendering

Not every piece of geometry in your application will be unique. In your application, the geometry is more like “apples to apples,” where every single Granny Smith apple tree is a clone. If you have created a particle system, you know that each particle is not a unique geographic mesh. Each particle has a base geometry that is shared with every other particle.

The fact that every particle shares a similar geometry is intentional. When every particle is identical, it allows you to make some optimizations. You don’t need to set up a separate draw call for each object. You can set up the entire rendering pass for thousands of objects with nearly the same amount of code it takes to set up just one pass. This approach has the dual advantage of not taking up a massive amount of code and minimizes the load on the CPU.

Ring Buffer

In previous examples where you had only one draw call, you followed a procedure for setting up that draw call:

1. Pushing the data, such as uniforms or surface normal, to the GPU

2. Binding the shaders, buffers, and textures

3. Issuing the draw call

This is not the best way to approach doing thousands of draw calls. These steps take time. It’s better to have a lot of this data in place before you try to issue a draw call so that you don’t have to set up draw calls for each of your tens of thousands of draws. (Preparing your data before issuing a draw call is similar to the concept of *mise en place*—before you begin cooking, you prepare all of the ingredients ahead of time. That way, the vegetables already in the wok don’t burn as you prepare the ones that are added later. It makes the process far more efficient.)

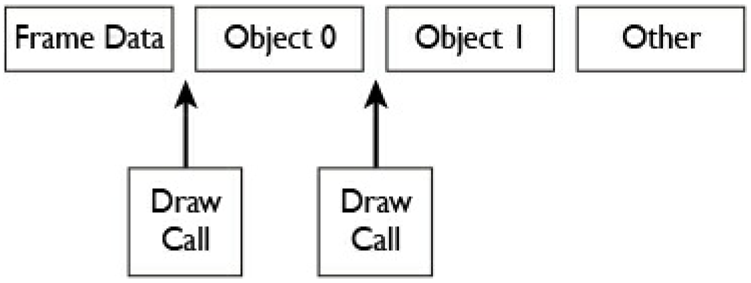
Instead of wasting time setting up the code tens of thousands of times, set up a constant buffer to contain all the data in one data structure. This constant buffer is set up outside of the render loop and contains the data to render one frame.

There are several advantages to this technique. One is that you can use parts of the same data for every object in the constant buffer. If you are drawing 1,000 cubes that all share the same mesh, for example, you can have each bit of data reference the same mesh rather than having lots of duplicate data.

The following code creates a ring buffer, as shown in Figure 10.2.

```
// Constant buffer ring
var constantBuffers : Array<MTLBuffer> = [MTLBuffer] ()
var constantBufferSlot : Int = 0
var frameCounter : UInt = 1
```

Figure 10.2. *How constant buffer memory is laid out*



Ring buffers are a common concept in both graphics and audio programming. In Metal, the CPU and the GPU share space on a chip. If you had only one constant buffer, the CPU would spend a frame setting up all the data to be sent to the GPU. Then, on the next frame, both the CPU and GPU would be referencing the same spot in memory. The CPU would overwrite those settings before the GPU had a chance to process the data. This is no good. So rather than making just one buffer to contain all of your data, you create an array of MTLBuffer objects. While the GPU is processing the data in the first buffer, the CPU is adding data to the next buffer. When the CPU reaches the end of the array, it loops back and starts over again on the first buffer, hence the term *ring*.

You need to know how large each buffer in the buffer ring has to be. This is calculated by taking the number of objects and the size that each object is going to be and allocating that amount of memory for each constant buffer in the ring:

```
let CONSTANT_BUFFER_SIZE : Int = OBJECT_COUNT *
                                 MemoryLayout<ObjectData>.size +
                                 SHADOW_PASS_COUNT *
                                 MemoryLayout<ShadowPass>.size +
                                 MAIN_PASS_COUNT *
                                 MemoryLayout<MainPass>.size
```

You don’t want too few possible frames being processed because you don’t want to overwrite data. Nor do you want an endless number of buffers to be created. A good safe value for number of frames in a ring buffer is three—one to be processed by the CPU, one to be processed by the GPU, and one waiting in the wings to be picked up by the CPU:

```
let MAX_FRAMES_IN_FLIGHT : Int = 3
```

Next, populate the constant ring buffer with the correct number of frames:

```
for _ in 1...MAX_FRAMES_IN_FLIGHT {
    if let buf : MTLBuffer = device!.makeBuffer(
         length: CONSTANT_BUFFER_SIZE,
         options: MTLResourceOptions.storageModeShared)
    else {return}
    constantBuffers.append(buf)
}
```

This is a simple for loop that checks the maximum number of buffers you want to instantiate. It then creates those buffers and adds them to the constant buffer array. Next, you need to find some way to keep track of which buffers are currently in use and which ones are available.

```
semaphore = DispatchSemaphore(value: MAX_FRAMES_IN_FLIGHT)
```

To keep track of which frames are in use, you need to set up a semaphore. Dispatch semaphores can be used to control access to a resource across multiple execution contexts. Long before we had electronic communication, a semaphore system was developed as a means to communicate across distances using a series of flags held in different positions to represent letters and words. This concept has been adapted into computer science to control access to a shared resource.

```
mainCommandBuffer.addCompletedHandler { completedCommandBuffer in

    let end = mach_absolute_time()
    self.gpuTiming[Int(currentFrame % 3)] = end –
              self.gpuTiming[Int(currentFrame % 3)]

    let seconds = self.machToMilliseconds *
         Double(self.gpuTiming[Int(currentFrame % 3)])

    self.runningAverageGPU = (self.runningAverageGPU *
          Double(currentFrame-1) + seconds) / Double(currentFrame)

    self.semaphore.signal()
}
```

Each frame has a binary value of 0 or 1. If it’s free to be used, the semaphore shows its value as 0. When it goes into use, it changes to 1, which tells the compiler that this buffer isn’t open to be written to by the CPU. Once the GPU has finished with a frame, that value is set back to 0.

The last step is to increment the current buffer:

```
// Increment our constant buffer counter
// This will wrap and the semaphore will make sure we aren't using
// a buffer that's already in flight
constantBufferSlot = (constantBufferSlot + 1) %
                           MAX_FRAMES_IN_FLIGHT
```

This code increments the index of the current buffer. To ensure that you do not get an out-of-bounds error on your constant ring buffer, you run a modulo operation using the maximum number of frames.

Per-Frame Data

Now that you have a constant ring buffer system set up, you need to populate it with data. There are two general types of data you need to be concerned about: per frame and per object. Per-frame data is data for objects that don't share geometry, including your camera and lights as well as any world data necessary to render the scene properly, such as the projection matrices.

Per-frame data is stored in the first segment of the command buffer. It makes sense to create data structures representing both the per-frame and the per-object data to store in the buffers. The following example is of a structure that would hold your per-frame data:

```
struct ShadowPass
{
    matrix_float4x4 ViewProjection;
    matrix_float4x4 pad1;
    matrix_float4x4 pad2;
    matrix_float4x4 pad3;
};

struct MainPass
{
    matrix_float4x4 ViewProjection;
    matrix_float4x4 ViewShadow0Projection;
    vector_float4     LightPosition;
    vector_float4     pad00;
    vector_float4     pad01;
    vector_float4     pad02;
    matrix_float4x4 pad1;
};
```

Such structures hold global values that will not be instanced more than once. This assumes a multipass render target and that each struct holds the information necessary for each pass. Multipass rendering is covered in Chapter 13, "Multipass Rendering Techniques."

Next, you need to populate the current constant buffer. This is similar to how you populate any buffer. You need to calculate how much space each render pass's parameters will take. Knowing the space needed for each pass's parameters allows you to properly calculate the stride and offset of each set of data. If the first set of data takes up 24 bytes, for example, you can specify that the next set of data would begin on the next byte.

```
// Select which constant buffer to use
let constantBufferForFrame = constantBuffers[currentConstantBuffer]

// Calculate the offsets into the constant buffer for the shadow
// pass data, main pass data, and object data
let shadowOffset = 0
let mainPassOffset = MemoryLayout<ShadowPass>.stride + shadowOffset
let objectDataOffset = MemoryLayout<MainPass>.stride +
                       mainPassOffset

// Write the shadow pass data into the constants buffer
constantBufferForFrame.contents().storeBytes(of: shadowPassData[0],
    toByteOffset: shadowOffset, as: ShadowPass.self)

// Write the main pass data into the constants buffer
constantBufferForFrame.contents().storeBytes(of: mainPassFrameData,
    toByteOffset: mainPassOffset, as: MainPass.self)
```

After you write your per-frame data, it's time to add your per-object data.

Per-Object Data

Per-object data is data that is unique to each object. Each object instance will have certain aspects that are shared among those instances and are reused. The shared data is not included in the struct containing your per-object data. The ObjectData struct holds only data that is unique to each object.

```
struct ObjectData
{
    matrix_float4x4 LocalToWorld;
    vector_float4 color;
    vector_float4 pad0;
    vector_float4 pad01;
    vector_float4 pad02;
    matrix_float4x4 pad1;
    matrix_float4x4 pad2;
};
```

This per-object data becomes a property on the renderable object:

```
class RenderableObject
{
    let mesh : MTLBuffer?
    let indexBuffer : MTLBuffer?
    let texture : MTLTexture?

    var count : Int

    var scale : vector_float3 = float3(1.0)
    var position : vector_float4
    var rotation : vector_float3
    var rotationRate : vector_float3

    var objectData : ObjectData
```

If you look at the top of this class declaration, you see a list of properties that will be common among all instances of this class: the mesh, index buffer, and scale. The aspects of the object that change are included in an instance of the ObjectData struct.

To keep track not only of each object but also of how many objects you must deal with, it makes sense to create an array to hold each renderable object:

```
var renderables : ContiguousArray<RenderableObject> =
    ContiguousArray<RenderableObject>())
```

Next, you set up a loop to add these renderable objects to the array:

```
// MARK: Object Creation
do {
    let (geo, index, indexCount, vertCount) = createCube(device!)

    for _ in 0..<OBJECT_COUNT {
        // NOTE returns a value within -value to value
        let p = Float(getRandomValue(500.0))
        let p1 = Float(getRandomValue(100.0))
        let p2 = Float(getRandomValue(500.0))


        let cube = RenderableObject(m: geo, idx: index,
                           count: indexCount, tex: nil)
        cube.position = float4(p, p1, p2, 1.0)
        cube.count = vertCount

        let r = Float(Float(drand48())) * 2.0
        let r1 = Float(Float(drand48())) * 2.0
        let r2 = Float(Float(drand48())) * 2.0

        cube.rotationRate = float3(r, r1, r2)

        let scale = Float(drand48()*5.0)

        cube.scale = float3(scale)

        cube.objectData.color = float4(Float(drand48()),
                                       Float(drand48()),
                                  Float(drand48()), 1.0)
        renderables.append(cube)
    }
}
```

The previous code is fairly straightforward. It loops through the object count set within the program to create a set number of instances of a renderable cube. All the properties on the renderable object are created. Some, such as the position and color, are procedurally and randomly generated. After all the properties are set, the object is appended to the array.

Because binding the mesh to the buffer is a computationally expensive operation, you want to do it only once. You can do so by specifying the mesh associated with the first object in the array:

```
// We have one pipeline for all our objects, so only bind it once
enc.setVertexBuffer(renderables[0].mesh, offset: 0, at: 0)
```

You’re almost finished encoding the render pass. You just need to calculate the offset for all the objects in the renderable array, end the encoding, and commit the command buffer.

```
var offset = objectDataOffset
for index in 0..<objectsToRender {
    renderables[index].DrawZPass(enc, offset: offset)
    offset += MemoryLayout<ObjectData>.size
}

enc.endEncoding()

commandBuffer.commit()
```

Summary

Metal has many objects built into the framework that can be expanded on to allow the program to do more work on the GPU without negatively affecting it. You use scene graphs to organize all the instances of an object present in a frame. You can create ring buffers to contain large groups of instanced render objects. You can set up all of your data in arrays of buffers that are all bound at once to avoid expensive operations on the CPU. These practices fit within the Metal paradigm of not performing expensive work more often than you have to.

11. Interfacing with Model I/O

How do you make the timelessness of inert, silent objects count for something? How to use the, in a way, dumbness of sculpture in a way that acts on us as living things?

—Antony Gormley

If you've ever played a complex 3D game, such as *Mass Effect* or *Call of Duty*, you know that the video game world is populated by environmental objects, people, and lights. Millions of polygons are represented in every frame of those video games. So far, you've worked with incredibly simple objects that you have described by hand. A fully rigged and modeled human character is endlessly complex and would be impossible to enter by hand. Modeling programs get around having to do this by creating pattern files that describe these objects, but how do you get them into your application? Before 2015, you would write a parser. Since 2015, there is a better solution to this problem: Model I/O. This chapter gives you an idea of how to import these files into your application, customize them, and connect the data to Metal for further customizations.

What Are Model Files?

Back in Chapter 3, "Your First Metal Application (Hello, Triangle!)," and Chapter 8, "2D Drawing," you entered all of your vertex position and color data by hand. That's doable if your object is simple, but once you get beyond primitive shapes, it becomes a nightmare. The only time you would ever enter this data by hand is as a learning exercise to see how to set up the vertex buffers. Adding a shape of any complexity is simply not done by hand.

So if it's not done by hand, how is it done?

Think about 3D modeling programs, such as Blender and Maya. These programs export files that are representations of models you create in them. You may not be aware of it, but many of the files that you interact with on a daily basis are simply XML files describing various settings that exist in your project. Interface Builder in Xcode is backed by an XML file, which is one reason it's been historically difficult to merge changes done by two different people into a storyboard. Adobe Illustrator files are XML documents describing each vector shape present in your project. The base application, be it Xcode or Illustrator, parses that XML file to reconstitute your project when you open that file after it's been saved.

Model files from Maya and Blender include a lot of information about your model. They include the position of all the vertices, how the vertices are connected, color, texture, lighting, and so on. Before Model I/O, when you wanted to import a model file from one of these programs, you had to go in and parse it yourself. You had to look at the file and figure out what all the different settings meant. You then had to write a script to read this file and parse out what each setting was that you needed to import. Then you had to connect those settings to the parts of your code that needed them. So, if you were importing a simple box, you had to set up a vertex buffer for each vertex position, how they were assembled, what colors were associated with each vertex, and any other information you chose to include in that model. It was a lot of work. Since there are a few commonly used file formats, other people like you were doing this exact same work for every project they were working on. This was a poor use of developer resources, so Apple created support for five common file formats for 3D models:

• **Alembic**: .abc

• **Wavefront Object**: .obj

• **Polygon**: .ply

• **Universal Scene Description**: .usd

• **Standard Tessellation Language**: .stl

Of these file formats, only USD and Alembic support importing animations. If you are working on a complex immersive scene with characters that need to move around the project, being able to import those animations is vital to your project.

It’s illustrative to take a look at these files just to get a feel for what information they have encoded in them, but it’s no longer necessary to understand them inside and out. Being able to simply import these files into your project and trust that they will work relieves you of a great amount of stress to get even the simplest 3D task done in iOS.

This chapter has a project associated with it: importing a model, setting up the model, adding some customization, and exporting it to be used in a modeling application. This simple project uses the Newell Teapot, a well-known and open source model for 3D graphics programming. This project focuses only on bringing in the model data. The work of setting up the shaders is detailed in Chapter 5, “Introduction to Shaders.”

Reading a File Format

A model file is nothing more than a specialized parsed language, similar to XML or JSON, detailing various pieces of information about a model. Some can be simple with only vertex position data. Others can be complex with texture mapping, lighting, and camera descriptions. Some model file formats, such as .obj, can be read from a text editor. Others are represented in binary and require a parser to understand because they are not human readable.

For example, a typical .obj file might look something like this:

```
# List of geometric vertices, with (x,y,z[,w]) coordinates,
# w is optional and defaults to 1.0.
 v 0.123 0.234 0.345 1.0
 # List of texture coordinates, in (u, v [,w]) coordinates,
 # these will vary between 0 and 1, w is optional and
 # defaults to 0.
 vt 0.500 1 [0]
 # List of vertex normals in (x,y,z) form; normals might not be
 # unit vectors.
 vn 0.707 0.000 0.707
 # Parameter space vertices in ( u [,v] [,w] ) form;
 # free form geometry statement ( see below )
 vp 0.310000 3.210000 2.100000
 # Polygonal face element (see below)
 f 1 2 3
 f 3/1 4/2 5/3
 f 6/4/1 3/5/3 7/6/5
 f 7//1 8//2 9//3
```

You need to be familiar with what each abbreviation stands for. v stands for “vertex” clearly, but what about f and mtllib? Do you need to bring everything in a model file into your project, or can you bring in just certain things?

The Internet is your friend. There are reference pages to these and other file formats, so you can decode what you are looking for. If you're using a file format that isn't supported by Model I/O, you need to make sure you understand these formats because you will need to write your own parser. It's also not necessary to import every piece of information a model file has. In the example project for this chapter, the shaders need to know the vertex positions and the vertex normals, but nothing else. You can be safe and bring in everything, or you can determine what your requirements are and import only those. It's up to you.

Importing a Model

The foundation of Model I/O is built around the MDLAsset class. MDLAsset is an indexed container for 3D objects and materials. These materials are instances of MDLObject. MDLObject has three subclasses that represent aspects of your model: MDLMesh, MDLCamera, and MDLLight.

In our earlier simple projects, we didn't set up custom cameras and lights, so the only MDLObject we would have in that instance would be an instance of MDLMesh representing the object's geometry.

A MDLAsset is initialized from a URL. It is simply a path inside your application bundle to a resource:

```
let teapotPath = Bundle.main.path(forResource: "wt_teapot",
                                       ofType: "obj")
let teapotURL:URL = URL.init(fileURLWithPath: teapotPath!)
let teapotModel = MDLAsset(url: teapotURL)
```

An MDLAsset is composed of several properties and methods:

• **Allocators**: Allocators are objects responsible for creating buffers containing mesh vertex data loaded from the asset.

• **Descriptors**: MDLVertexDescriptor describes how mesh vertex data is loaded from the asset.

• **Import and export**: MDLAsset supports both the importing and exporting of asset objects. A suite of methods supports this functionality.

• **MDLObject children**: This is the meat of what you think of when you think of an MDLAsset. It contains all the mesh, camera, and lighting objects in an asset.

The .obj file format is very simple. It represents only 3D geometry, so all children contained in an .obj file must be instances of MDLMesh.

What is a mesh? Glad you asked. Meshes are covered next.

MDLCamera and MDLLight

If you watched the Model I/O WWDC video back in 2015 (and why wouldn't you have?!), you noticed that a lot of the presentation discussed the MDLCamera and MDLLight objects. It reviewed all the ways you can set up a camera to mimic real-world lenses and apertures. It also went over how you can re-create in code the light fixture you have in your office.

That sounds cool. You probably thought you could bypass a bunch of complicated shader code by creating a bunch of MDLCamera and MDLLight objects, like you would in something like Unity. Sorry to burst your bubble, but it doesn't work that way.

Think about the MDLMesh object. It's a container that holds all the vertex data from an object file. When you create an MDLMesh, it doesn't create all that vertex data that you use in your teapot. It simply provides an object that parses the model file for you and stores the vertices in a way that makes it easy to feed them to the GPU.

Some of the more sophisticated file formats, such as Alembic, support the creation of lights and camera objects. MDLCamera and MDLLight are container objects to hold that data from file formats that already support it. So, if you're looking for a quick way to add some lighting to your .obj file, you're out of luck. In Metal, such as in life, there is no such thing as a free lunch.

Meshes and Submeshes

An MDLMesh object is composed of at least one MDLSubmesh. Unlike our previous triangle and star projects, most 3D objects are not all one piece. If you play a game like *The Sims* where your characters have different outfits for different situations, such as work or sleep, each part is its own submesh. Because your Sim isn't going to wear a swimsuit to the office, your program has the option to not render that submesh to the character when the character isn't going to the pool.

MDLMesh isn't constrained to using just the information that you share with it through your model file. It includes generators, modifiers, and bakers to enhance the data present in the model. It can generate additional geometry beyond what is present in the model file. Modifiers can compute normals and tangents and can make vertices unique. Bakers can generate ambient occlusion and light maps that can be exported with the model back to the modeling program.

MDLMesh's job is to take all the data that is present in the model file and populate the MDLMeshBuffers with that data. This data is then described by the MDLVertexDescriptor so the vertex shader knows how to make sense of the data in the buffers. Finally, these buffers then feed the data to the vertex shader, as described in Chapter 2, "Overview of Rendering and Raster Graphics."

One of the most important differences between an MDLMesh and an MDLSubmesh is that the mesh holds the vertex buffer that is referenced by all submeshes, whereas the submeshes hold a reference to their corresponding index buffer.

First, you need to create a MTKMesh array to hold all the meshes from the MDLAsset you created with the teapot model:

```
var meshes: [MTKMesh]!
```

This array will be referenced by a few different objects in the class, so it needs to be a property inside the class but outside of all methods.

Render State Pipeline

In previous chapters, you set up the MTLRenderPipelineDescriptor with vertex data entered by hand. Because you're working with far more complex objects now, you still need to set up these vertex descriptors, but you're going to set them up slightly differently to take advantage of all the scaffolding Model I/O gives you.

You've specified that you are getting your vertex data from the vertex descriptor. Let's set that up:

```
vertexDescriptor.attributes[0].offset = 0
// position
vertexDescriptor.attributes[0].format = MTLVertexFormat.float3
vertexDescriptor.attributes[1].offset = 12
// Vertex normal
vertexDescriptor.attributes[1].format = MTLVertexFormat.float3
vertexDescriptor.layouts[0].stride = 24
```

This code describes how data each vertex that will be laid out by the vertex shader. The position has an *x*, a *y*, and a *z* coordinate. Each coordinate uses 4 bytes of data for a total of 12 bytes. Because the position is the first attribute described in the vertex descriptor, its offset is 0. The next attribute is the vertex normal, which also uses 12 bytes of data. Its offset is 12 because the first 12 bytes are reserved by the position attribute. After you describe how much data you need for each aspect of each vertex, you set your stride to be the total number of bytes necessary to allocate each vertex (in this case, 24).

After all of your memory is allocated, you need to hand that to the render pipeline state:

```
let pipelineStateDescriptor = MTLRenderPipelineDescriptor()
pipelineStateDescriptor.vertexFunction = vertexProgram
pipelineStateDescriptor.fragmentFunction = fragmentProgram
pipelineStateDescriptor.colorAttachments[0].pixelFormat =
    view.colorPixelFormat
pipelineStateDescriptor.sampleCount = view.sampleCount
pipelineStateDescriptor.vertexDescriptor = vertexDescriptor

do {
   try pipelineState = device.makeRenderPipelineState(
                  descriptor: pipelineStateDescriptor)
} catch let error {
    print("Failed to create pipeline state, error \(error)")
}
```

Next, you need to set up asset initialization.

Asset Initialization

You have memory set aside for various attributes for each vertex. That's all well and good, but the program needs a way to know what each byte in each vertex represents. Is this set of ones and zeros a position? A color? A texture? MetalKit has a function to sort this out: MTKModelIOVertexDescriptorFromMetal. This function translates from Metal's particular vertex descriptor to Model I/O's more general vertex descriptor, but we need to provide more information so that Model I/O knows which vertex attribute in the model to associate with each vertex attribute in the Metal descriptor:

```
let desc = MTKModelIOVertexDescriptorFromMetal(vertexDescriptor)
var attribute = desc.attributes[0] as! MDLVertexAttribute
attribute.name = MDLVertexAttributePosition
attribute = desc.attributes[1] as! MDLVertexAttribute
attribute.name = MDLVertexAttributeNormal
let mtkBufferAllocator = MTKMeshBufferAllocator(device: device!)
let url = Bundle.main.url(forResource: "wt_teapot",
                              withExtension: "obj")
```

```
let asset = MDLAsset(url: url!, vertexDescriptor: desc,
                   bufferAllocator: mtkBufferAllocator)

do {
 meshes = try MTKMesh.newMeshes(from: asset,
         device: device!, sourceMeshes: nil)
}
catch let error {
 fatalError("\(error)")
}
```

Referring to the previous section, you created and allocated memory for two different types of data: position and vertex normal. Each has a corresponding MDLVertexAttribute that is added to the vertex descriptor. If you didn't already create the MDLAsset for the teapot, it can be created here with the vertex descriptor.

The last step you need to do to connect your imported model to MetalKit is to set up your render state and start your draw calls.

Render State Setup and Drawing

All rendering processes must complete with a draw call. To draw the meshes, the application must specify which mesh and submesh objects it needs to reference. This setup also takes advantage of the indexed drawing you learned about in Chapter 10, "Advanced 3D Drawing." This is the code to set up the draw call:

```
let mesh = (meshes?.first)!
let vertexBuffer = mesh.vertexBuffers[0]
commandEncoder.setVertexBuffer(vertexBuffer.buffer,
                offset: vertexBuffer.offset, at: 0)

let submesh = mesh.submeshes.first!
commandEncoder.drawIndexedPrimitives(submesh.primitiveType,
                            indexCount: submesh.indexCount,
                              indexType: submesh.indexType,
                   indexBuffer: submesh.indexBuffer.buffer,
             indexBufferOffset: submesh.indexBuffer.offset)
```

Hooking up the shaders is beyond the scope of this chapter. This chapter shares a project with Chapter 5. If you want to see how to set up a complete project between loading the model and writing the shaders, refer to both the project in Chapter 5 and this one.

Exporting Files

One last thing you can use Model I/O for is to export your model files. You might wonder why you would ever need to do this. If you're working with an artist and you want to make changes to send back to the artist, you can do it in code and export the modified model. Exporting the model is a simple process:

```
url = URL(string: "/Users/YourUsername/Desktop/exported.obj")
try! teapot.export(to: url!)
```

Summary

Model I/O is the simplest way that iOS has to import 3D models into a Metal application. It was designed to integrate seamlessly with MetalKit. Model I/O supports five common file types. It also supports importing meshes, cameras, and lights. Each MDLAsset has, at minimum, one submesh. The programmer provides the object (allocator) that knows how to allocate memory to store the model data, Model I/O loads the data into these buffers, and Metal can then be used to easily render the resulting meshes.

12. Texturing and Sampling

Texture is something we forget—it makes outfits look very expensive. You can do a monochromatic outfit, if you're afraid of things that are more colorful and printed, and still create interest.

—Stacy London

Shaders are incredibly powerful programs capable of displaying a lot of interesting and realistic effects. However, the more effects you add, the more processing power and time your program takes. Every millisecond is precious, so it's important to optimize your project as much as possible. One easy way to get the most bang for your buck is to map textures to your objects rather than procedurally generating your textures in shaders. This chapter introduces you to texture mapping and sampling in Metal.

Texture Mapping

Texturing is similar to shading. Your program scans the surface one pixel at a time and determines what the pixel should look like at that pixel. In Chapter 6, "Metal Resources and Memory Management," you learned about buffers and textures. Thus far, you have worked only with buffers. Rather than receiving a buffer of color information and running it through an algorithm, the program receives texture resources. The resources are applied to the surface one pixel at a time.

Pretend you're applying wallpaper to a wall. You start at one corner. You make sure the wallpaper is straight and aligned with the sections on each side of it. If this task is done properly, the wallpaper is applied seamlessly and creates the illusion of a single texture.

So far in this book, you've been using Cartesian coordinates where each point lays on an x, y, and z-axis in space. To differentiate between a mesh's coordinates and a texture's coordinates, a different coordinate space is used.

The coordinate space for texture coordinates is the UV coordinate space. This space is similar to XY. The texture exists as a normalized coordinate space where each value exists between 0.0 and 1.0. To differentiate between the surface of the object in world space and the location of a fragment on a texture, the fragments of a texture are referred to as *texels*.

This process works well if you have a perfectly square surface and perfectly square texture. It doesn't work as well when you need to map a texture to a curved surface. Many options are available for this contingency in whatever 3D modeling program you are using to create your assets. The model is exported with a texture atlas that includes the skin of the model. The pieces are placed in a single texture file and a pointer to each texel is included in the model file.

It's fairly simple to create a texture object in Metal. For a more in-depth explanation of MTLTexture objects, refer to Chapter 6.

```
var depthTexture: MTLTexture! = nil
```

Your texture requires a texture descriptor so that Metal can process it as you intend:

```
let depthTextureDescriptor =
    MTLTextureDescriptor.texture2DDescriptor(
        pixelFormat: .depth32Float,
        width: Int(layerSize.width),
        height: Int(layerSize.height),
        mipmapped: false)
depthTexture = device.makeTexture(
    descriptor: depthTextureDescriptor)
```

This texture, as is clear from its name, is a depth texture. The MTLTextureDescriptor is set to 2D descriptor and initialized. It requires a pixel format of the texture, the size of the view, and whether or not you want it mipmapped. Since the texture is the same size and shape as the mesh, you set mipmapping to false. Lastly, you initialize your depthTexture with the depth texture descriptor.

Mipmapping

One aspect of graphics is that they change size. If you've ever zoomed into a low-resolution JPEG, the image gets pixelated. It is fairly easy to conceptualize that when you zoom into a fragment, it takes up progressively more pixels on the screen. What happens when you go in the opposite direction and you have four or five fragments representing one pixel on your screen? Neither linear nor nearest is set up to accommodate this situation. This section covers how Metal handles it.

The solution for minifying a texture with overlapping pixels is known as *mipmapping*. A mipmap is a sequence of textures that all contain the same image but at lower and lower resolution. The original, full-sized texture exists at the base level of the mipmap. Each level above that progressively downsamples by a factor of 2. For example, if you had a texture that was 16 × 16, the base level would also be 16 × 16. A full mip chain for this texture would consist of levels of size 16 × 16 (base), 8 × 8, 4 × 4, 2 × 2, and 1 × 1, for a total of five levels. The mipmap generator then precomputes and stores the averages of the texture over various areas of different size and position.

Because this is such a common operation, Metal has a built-in implementation to generate mipmaps on the MTLBlitCommandEncoder.

```
func generateMipmaps(for texture: MTLTexture)
```

You don't need to specify how many levels of mipmap you need. The number of levels is calculated on the basis of the number of fragments present in each texture.

Sampling

In Metal, everything is about packaging state information to be sent to the GPU. Sampling is no different. A sampler is an object that encapsulates the various render states associated with reading textures: coordinate system, addressing mode, and filtering. Setting these states incorrectly can lead to poorly sampled textures and aliasing.

In Chapter 5, "Introduction to Shaders," you started out with a solid color that was applied to each pixel. That color was processed by the shader to account for lighting position. Using a texture is similar. Instead of passing a solid color to the shader, you are sampling the color at that position in the texture.

Sampling is a fascinating and mathematically complex process that has enough material around it to justify its own book. Chapter 16, "Image Processing in Metal," provides a further explanation of sampling. It's not important at this point for you to understand every nuance of the math surrounding it, so this is a very brief overview.

Your program checks a location in the texture and interpolates what the color is most likely to be at that location on the texture. If you don't take enough samples of the texture, the interpolator will not get an accurate representation of what the texture looks like. When sampling isn't done properly and the program cannot properly reconstruct the texture, you get *aliasing*—the jagged edges you've no doubt seen on some images.

Address Modes

You've learned about normalized coordinate systems where a percentage is used instead of a solid unit of measurement for an object or a texture. This is one common way to deal with textures, but it is not the only way. If you have negative coordinates or the measurement of a texture goes beyond 1, you need a way to address it. That way is, you guessed it, *address modes*.

The Metal framework has an enumeration of address modes called MTLSamplerAddressMode, which has the following values (see Figure 12.1):

• **clampToEdge**: Texture coordinates are clamped between 0.0 and 1.0, inclusive.

• **mirrorClampToEdge**: Between −1.0 and 1.0, the texture coordinates are mirrored across the axis. Outside −1.0 and 1.0, the texture coordinates are clamped.

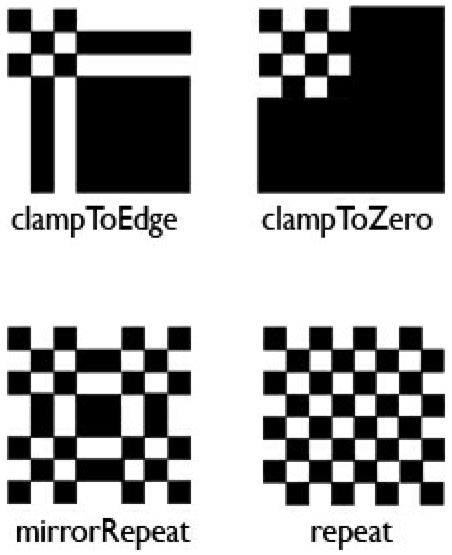
• **repeat**: Texture coordinates wrap to the other side of the texture, effectively keeping only the fractional part of the texture coordinate.

• **mirrorRepeat**: Between −1.0 and 1.0, the texture coordinates are mirrored across the axis. Outside −1.0 and 1.0, the image is repeated.

• **clampToZero**: Out-of-range texture coordinates return transparent zero (0,0,0,0) for images with an alpha channel and return opaque zero (0,0,0,1) for images without an alpha channel.

• **clampToBorderColor**: Out-of-range texture coordinates return the value specified by the borderColor property.

Figure 12.1. *Examples of different address modes*



Make sure you’re accurate in how you want to address your textures. As you can see, there are a few different effects that you can get by changing the address mode, and choosing the wrong one might not give you the effect you’re looking for.

Filter Modes

The process of reconstructing an image from formatted image data is known as *filtering*. There are two different modes for filtering: linear and nearest.

Nearest is the fastest and least computationally expensive. Nearest simply uses the color of the texel closest to the pixel center for the pixel color. This is a fairly crude method that results in aliasing and blocky artifacts. It also works much better with magnification than it does with minification. On minification, it can be incredibly computationally intensive and deliver terrible results.

Linear is more accurate because it samples the four nearest pixels and averages their values. Because you are sampling more texels, it is a more expensive operation. However, it’s not so expensive that it can’t be used in real-time rendering. It’s a filter that is usually good enough.

Anisotropy

The concepts discussed previously work well if you have a texture that is square to the camera—but this is rarely the case. If you are applying a texture to a landscape, the perspective of the texels in the texture are different up close to the camera from the perspective as you move farther back into the horizon.

Anisotropic filtering adjusts the sampling done to the texture according to how it’s angled on the screen. As you get closer to the texture being completely parallel to the camera, you need more sampling to avoid aliasing and blurriness. Anisotropic filtering is smart and adjusts the sampling distribution to sample more when it’s necessary and sample less when it’s not.

Precompiled Sampler States

Samplers have the same advantage as textures in Metal in that you can create precompiled sampler states. The sampler and the way you set it up are not going to change in the middle of the rendering pass, so you can remove some of the computational expense by prebaking this state into the command encoder as you would any other resource function.

To set up a sampler, you need a resource property and a sampler state property at the top of your class:

```
var diffuseTexture: MTLTexture! = nil
var samplerState: MTLSamplerState! = nil
```

In this example, you use a diffuse texture that will be contained within a MTLTexture resource. You also create a MTLSamplerState to act as a container for the state you wish to set. You set that state next:

```
let samplerDescriptor = MTLSamplerDescriptor()
samplerDescriptor.minFilter = .nearest
samplerDescriptor.magFilter = .linear

samplerState = device.makeSamplerState(
    descriptor: samplerDescriptor)
```

To set the state for the MTLSampler, you need a MTLSamplerDescriptor. This descriptor contains all the possible settings you may wish to specify for the rendering pass. Among these are

• Address mode of each dimension of your texture

• Options for your minification, magnification, and mipmap filters

• Level of detail clamp

• Maximum number of samples for your anisotropic filter

In this simple example, you just set the samplerDescriptor.minFilter and samplerDescriptor.magFilter. If you recall from earlier in this chapter, .nearest works differently depending on whether you are minifying or magnifying a texture, so it makes sense to use .nearest on one and .linear on the other.

Finally, you need to specify to the command encoder which fragment texture and fragment sampler state you wish to use:

```
commandEncoder.setFragmentTexture(diffuseTexture, at: 0)
commandEncoder.setFragmentSamplerState(samplerState, at: 0)
```

Now that you have specified to the encoder what sampler state and texture you're using, you need to tell the program what you want them to do with them. Samplers can also be created inside of shader functions without touching the GPU.

Passing Textures and Samplers to Graphics Functions

To pass a texture to a sampler, you need to have a texture object. This is slightly more complicated than just including an image in your application bundle. In Chapter 6, you learned about MTLResource objects. A MTLTexture is one type of MTLResource. In the following section, you learn how to create a MTLTexture object and how to set up the shader functions that will interact with the texture.

Creating a Texture

This section details the code necessary to create a MTLTexture object. This method is broken up into a few manageable chunks to make explaining it less daunting. The first section of this method declares the method signature and sets up the local properties you need to create your texture:

```
func textureForImage(_ image:UIImage, device:MTLDevice)
    -> MTLTexture?
{
    let imageRef = image.cgImage!

    let width = imageRef.width
    let height = imageRef.height
    let colorSpace = CGColorSpaceCreateDeviceRGB()

    let rawData = calloc(height * width * 4,
        MemoryLayout<UInt8>.size)

    let bytesPerPixel = 4
    let bytesPerRow = bytesPerPixel * width
    let bitsPerComponent = 8
```

You need to use a few properties that change depending on what image is passed into the function—namely, properties specifying the size of the image. Since the images are of variable size, you need to calculate how much memory you need to allocate to contain the raw value of the image data. Then you specify how many bits exist for each component and how many bytes you require for each row.

Next, you need to do some work with the Core Graphics context:

```
    let options = CGImageAlphaInfo.premultipliedLast.rawValue |
        CGBitmapInfo.byteOrder32Big.rawValue

    let context = CGContext(data: rawData,
                            width: width,
                            height: height,
                            bitsPerComponent: bitsPerComponent,
                            bytesPerRow: bytesPerRow,
                            space: colorSpace,
                            bitmapInfo: options)

    context?.draw(imageRef, in: CGRect(x: 0,
        y: 0,
        width: CGFloat(width),
        height: CGFloat(height)))
```

First you set up your bitmap and alpha options. Those are used as a parameter when you create your CGContext, along with several of the local properties you declared earlier. You use this context to draw the CGImage you created initially from the UIImage you passed in:

```
    let textureDescriptor =
    MTLTextureDescriptor.texture2DDescriptor(
        pixelFormat: .rgba8Unorm,
        width: Int(width),
        height: Int(height),
        mipmapped: true)
    let texture = device.makeTexture(descriptor: textureDescriptor)

    let region = MTLRegionMake2D(0, 0, Int(width), Int(height))

    texture.replace(region: region,
                    mipmapLevel: 0,
                    slice: 0,
                    withBytes: rawData!,
                    bytesPerRow: bytesPerRow,
                    bytesPerImage: bytesPerRow * height)
    free(rawData)

    return texture
```

Finally, you can create your texture object. You create a MTLTextureDescriptor by passing in the texture's width, height, and pixel format and specifying whether or not this object will be mipmapped. This information is used to create your texture. You then use this texture to replace the region you specified. You don't need to allocate memory you are no longer using, so that memory is freed and the texture is returned.

Using this method to create a new texture from an image is easy. You simply pass in the UIImage you want to use and the MTLDevice.

```
diffuseTexture = self.textureForImage(UIImage(named: "bluemarble")!, device: device)
```

Metal Texture Functions

Texture data is similar to other data that is passed to the shaders. It's a resource represented by a series of float values. As with any shader function, you need to create data structures in the .metal file that coordinate with the values in the argument table being passed by the main program:

```
struct TexturedInVertex
{
    packed_float4 position [[attribute(0)]];
    packed_float4 normal [[attribute(1)]];
    packed_float2 texCoords [[attribute(2)]];
};

struct TexturedColoredOutVertex
{
    float4 position [[position]];
    float3 normal;
    float2 texCoords;
};

struct Uniforms
{
    float4x4 projectionMatrix;
    float4x4 modelViewMatrix;
};
```

In a simple sampler project, you are referencing vertex positions, normals, and texture coordinates in the argument table. These are set in the main program and referenced by the shader:

```
vertex TexturedColoredOutVertex vertex_sampler(
    device TexturedInVertex *vert [[buffer(0)]],
    constant Uniforms &uniforms [[buffer(1)]],
    uint vid [[vertex_id]])
{
    float4x4 MV = uniforms.modelViewMatrix;
    float3x3 normalMatrix(MV[0].xyz, MV[1].xyz, MV[2].xyz);
    float4 modelNormal = vert[vid].normal;

    TexturedColoredOutVertex outVertex;
    outVertex.position = uniforms.projectionMatrix *
        uniforms.modelViewMatrix * float4(vert[vid].position);
    outVertex.normal = normalMatrix * modelNormal.xyz;
    outVertex.texCoords = vert[vid].texCoords;

    return outVertex;
}
```

The vertex buffer calculates how to translate the position from the object to the correlating position within the texture. Think about the game *Battleship*. Each player has a corresponding grid of locations. The shader determines which box in the object's grid correlates to the current location in the texture's grid. That corresponding color is passed to the fragment shader:

```
fragment half4 fragment_sampler(
    TexturedColoredOutVertex vert  [[stage_in]],
    texture2d<float, access::sample> diffuseTexture [[texture(0)]],
    sampler samplr [[sampler(0)]])
{
    float4 diffuseColor = diffuseTexture.sample(samplr,
        vert.texCoords);
    return half4(diffuseColor.r,
        diffuseColor.g, diffuseColor.b, 1);
}
```

The fragment shader takes the current color from the vertex shader and calculates a diffuse lighting value for the fragment. Textured areas of the object farther away from the light source are modified to create the illusion that the light exists and behaves the way it would in the real world.

Summary

Texturing and sampling are incredibly important tools in your Metal toolbox. They easily add a lot of interest and detail to your 3D scenes without much computational overhead in your application. Metal has many built-in methods to help resize and sample your textures to avoid artifacts and aliasing. Understanding the best way to resample textures when scaling up or down can help you create better-looking applications without affecting performance.

13. Multipass Rendering Techniques

Leeloo Dallas Multipass!

—The Fifth Element

Multipass rendering is common, especially in increasingly complex scenes. This technique allows you to create better and more realistic rendering without bogging down the GPU.

When (and Why) to Use Multipass Rendering

Lighting is the most important reason to use multipass rendering. Light is a subtle and complex phenomenon to replicate in graphics programming. There are so many different types of light and materials that trying to create realistic lighting is the holy grail of realistic graphics programming.

A rendered image is made of layers. The first layer is the underlying meshes and geometry that make up a scene. The next layer is the ambient light that illuminates the scene uniformly. After that, there is a layer for each light that you add to the scene. These lights cast shadows, which create a more realistic scene.

Many different factors are at play: shadows cast by the rays of light from each light source and secondary and tertiary reflections when light hits a surface and bounces off and lands on another surface. Often, the only real way to accomplish realistic lighting is to set up a rendering pass for each light source. A scene can easily have multiple pin lights along with an ambient light. Each light source needs a rendering pass.

Metal Render Pass Descriptors

This section focuses on Metal render pass descriptors. To set up multiple render passes, you need to have a good grasp of what constitutes a render pass descriptor and how that changes when you have multiple render passes.

A MTLRenderPassDescriptor is basically a collection of attachments. GPU programming is fundamentally bundling up data to send to the GPU along with any settings to explain that data. Those settings are encapsulated in a MTLRenderPassDescriptor. Unlike MTLPipelineDescriptor, of which you can have only one, you can have multiple MTLRenderPassDescriptors. In fact, most applications would be severely limited if you were constrained to just one. These descriptors can be used for texturing, shadow mapping, and lighting. They can be singular, or they can be collected together into an array:

```
let mainRPDesc = MTLRenderPassDescriptor()
var shadowRPs : Array<MTLRenderPassDescriptor> =
                     [MTLRenderPassDescriptor]()
```

There are three types of information that can be attached to a render pass descriptor:

• Color

• Depth

• Stencil

Color attachments are arrays of MTLRenderPassColorAttachmentDescriptor objects. The attachment descriptor objects serve as the output destination for color pixels generated by a render pass. The pixels are specified using a *clear color*. Clear colors are not transparent, although they can be. Clear colors specify the color to use when the previous color attachment is cleared.

Depth attachment objects serve as the output destination for depth pixels generated by a render pass. This depth is stored as a double and is known as the *clear depth*. As with the clear color, this is the depth to use when the depth attachment is cleared. Depth attachments also include functionality for *multisampling antialiasing (MSAA) depth resolve*. MSAA depth resolution is a less computationally expensive way of doing image resampling to avoid aliasing.

The last attachment type is a MTLRenderPassStencilAttachmentDescriptor. A stencil is like a mask, which you may have used in Photoshop. It has a shape over which an effect can be painted, or stenciled, where there are areas of the image that receive an effect and areas that do not receive an effect.

Every one of these attachment types is a subclass of MTLRenderPassAttachmentDescriptor. This class has a few properties that apply to all the attachment types. For each type, you must set its texture property. There are a few optional properties on the texture, such as level, slice, and depthPlane, that can be set for the mipmap level, slice, and depth plane on 3D textures.

Creating and Managing Render Targets

These attachments and information often come from MTLTextureDescriptor objects. Here is an example of how you might set up a color attachment using a texture descriptor:

```
do {
    let texDesc = MTLTextureDescriptor()
    texDesc.width =  Int(frame.width)
    texDesc.height =  Int(frame.height)
    texDesc.depth = 1
    texDesc.textureType = MTLTextureType.type2D

    texDesc.usage = [MTLTextureUsage.renderTarget,
                     MTLTextureUsage.shaderRead]
    texDesc.storageMode = .private
    texDesc.pixelFormat = .bgra8Unorm

    mainPassFramebuffer =
      device!.makeTexture(descriptor: texDesc)

    self.mainRPDesc.colorAttachments[0].texture =
                              mainPassFramebuffer
}
```

In this code snippet, a MTLTextureDescriptor describes a texture that is created by the main render pass frame buffer. Since the color attachment is a subclass of MTLRenderPassAttachmentDescriptor, you are required to set its texture property.

One of the big wins with GPU programming is the ability to enable multithreaded rendering. If you choose to multithread, you need to set up the render pass encoding asynchronously on a dispatch queue:

```
// Generate the command buffer for Shadowmap
if multithreadedRender {
    dispatchGroup.enter()
    dispatchQueue.async {
        self.encodeShadowPass(shadowCommandBuffer,
                              rp: self.shadowRPs[0],
             constantBuffer: constantBufferForFrame,
                       passDataOffset: shadowOffset,
                 objectDataOffset: objectDataOffset)
        dispatchGroup.leave()
    }
} else {
    encodeShadowPass(shadowCommandBuffer,
                             rp: self.shadowRPs[0],
            constantBuffer: constantBufferForFrame,
                      passDataOffset: shadowOffset,
                objectDataOffset: objectDataOffset)
}
```

Whenever possible, you want to merge render targets together. Creating unnecessary render command encoders affects performance. Not every render encoder can be merged into a single render pass. A few criteria must be met in order to merge two render command encoders:

• The encoders are created in the same frame.

• The encoders are created from the same command buffer.

• The encoders share the same render target.

• The encoders don’t sample from the render targets in the other render encoder.

• One encoder’s store actions are either store or dontCare, and the second encoder’s load actions are either load or dontCare.

• The encoders are created on contiguous areas of memory with no other encoders between them.

These rules can apply to more than two encoders. If a group of 10 encoders is able to merge with the encoder before and after it, all 10 encoders can be merged and chained together.

Revisiting Load-Store Actions

The last bit you need to know about MTLRenderPassAttachmentDescriptor are *load* and *store actions*. MTLLoadAction is the action performed at the start of a rendering pass for a render command encoder. This is an enum with three values: load, clear, and dontCare. load preserves the existing contents of the texture. clear writes a new value to every pixel in the attachment. dontCare allows each pixel to take on any value at the start of the rendering pass because you don’t care what it is.

It’s important to choose the right load action for your render target. If you choose a computationally expensive operation that isn’t necessary, you’re burning computation time that could be better used elsewhere. If all the render target pixels are rendered to, choose the dontCare action. There are no costs associated with this action, and texture data is always interpreted as undefined. If the previous contents of the render target do not need to be preserved and only some of its pixels are rendered to, choose the clear action. Finally, if the previous contents of the render target need to be preserved and only some of its pixels are rendered to, choose the load action.

The other side of the coin is MTLStoreAction. This action is performed at the end of a rendering pass for a render command encoder. This enum has a few more cases than MTLLoadAction:

• **dontCare**: After the pass is complete, the attachment is not stored and is left undefined.

• **store**: The final result is stored in the attachment.

• **multisampleResolve**: This is used when you have multiple samples. These samples are resolved down to a single value and stored using the resolveTexture property, but the content of the attachment is left undefined.

• **storeAndMultisampleResolve**: This resolves the samples down to a single value and stores it to the attachment.

• **unknown**: This is a temporary state if you don't know your attachment's store action up front. You must specify a valid state before you finish encoding your commands to the render command encoder.

• **customSampleDepthStore**: This is a render target action that stores depth data in a sample position-agnostic representation. This is a rather advanced and sophisticated technique.

If the texture is a single-sample texture, you primarily need to choose between dontCare and store. If you don't need to preserve the contents of the render target, then you choose dontCare. If you do need to preserve it, then you choose store.

Things get more complicated when you have multisample textures. You need to ask yourself three questions:

• Does my multisample content need to be preserved?

• Is my resolve texture specified?

• Are my resolved contents preserved?

If the answer to these questions is yes, choose storeAndMultisampleResolve. If the resolved texture and contents need to be preserved but the multisampled contents do not, choose multisampleResolve. If only the multisampled contents need to be preserved, choose store. If nothing needs preserving, you dontCare.

The load and store actions have to be specified before the render pass descriptor is committed to the command encoder:

```
do {
    let rp = MTLRenderPassDescriptor()
    rp.depthAttachment.clearDepth = 1.0
    rp.depthAttachment.texture = shadowMap
    rp.depthAttachment.loadAction = .clear
    rp.depthAttachment.storeAction = .store
    shadowRPs.append(rp)
}
```

Before you get started, be sure to have an understanding of how you want to choose your load and store actions.

Summary

More complex scenes require the use of multiple render passes. Multiple render passes are usually used for lighting effects. Be sure to consolidate render passes whenever possible to optimize your program. It’s also important to understand what your load and store needs are.

14. Geometry Unleashed: Tessellation in Metal

Only those who attempt the absurd will achieve the impossible. I think it’s in my basement. . . . let me go upstairs and check.

—M. C. Escher

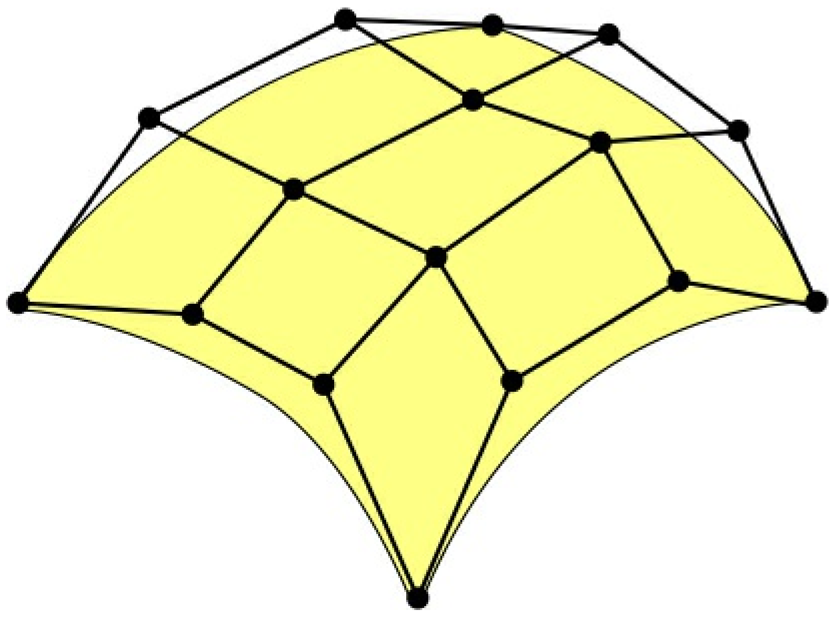
In graphics programming, you have to balance details with performance. Having incredibly realistic models can look amazing, but it will also be a major drag on your frame rate. One way you’ve already learned to add detail without geometry is texture mapping, but that has its own limitations. An image of a mountainside will not generate new shadows as the light source moves because it’s a flat image. There is another way to add detail without creating a drag on your GPU: *tessellation*.

Removing Bottlenecks by Using Tessellation

Tessellation is the process of taking a descriptor that does not have a lot of geometry and using a process to procedurally generate the missing geometry. When tessellation is used, the triangles are not stored in memory. Typically, when triangles are stored in memory and you pass your buffer of triangles to the fragment shader, the larger it is, the longer it takes to pass the triangles along. Tessellation removes this bottleneck by removing triangles that need to be fed through a buffer to the vertex shader. Hence, you gain performance.

Tessellation allows you to pass a low-resolution model to the shaders. It adds a few steps to the pipeline that split this geometry into smaller, more refined geometry. Rather than passing an entire mesh to the pipeline, you send a *patch*. A patch is a parametric surface composed of spline curves rather than triangles. These curves are made up of a set of control points and control vertices, as shown in Figure 14.1. The control points and vertices generally describe the structure in a rough manner without a lot of detail. The detail can be interpolated by the tessellator without weighing down the vertex buffers.

Figure 14.1. *A patch composed of control points*



Catmull-Clark Subdivision

This section covers the most common method used in computer science: *Catmull-Clark subdivision.*

In 1978, Edwin Catmull and Jim Clark published a paper on recursively generating surfaces using subdivision. Catmull spent some of his early career working for George Lucas at a subdivision of Industrial Light and Magic that was eventually sold to Steve Jobs for $10 million and was renamed Pixar. Catmull is a pioneer of computer graphics research, and his contributions to this field have been incalculable.

Subdividing a mesh of control points involves four basic steps:

1. Add a new point to each face, called the *face point*.

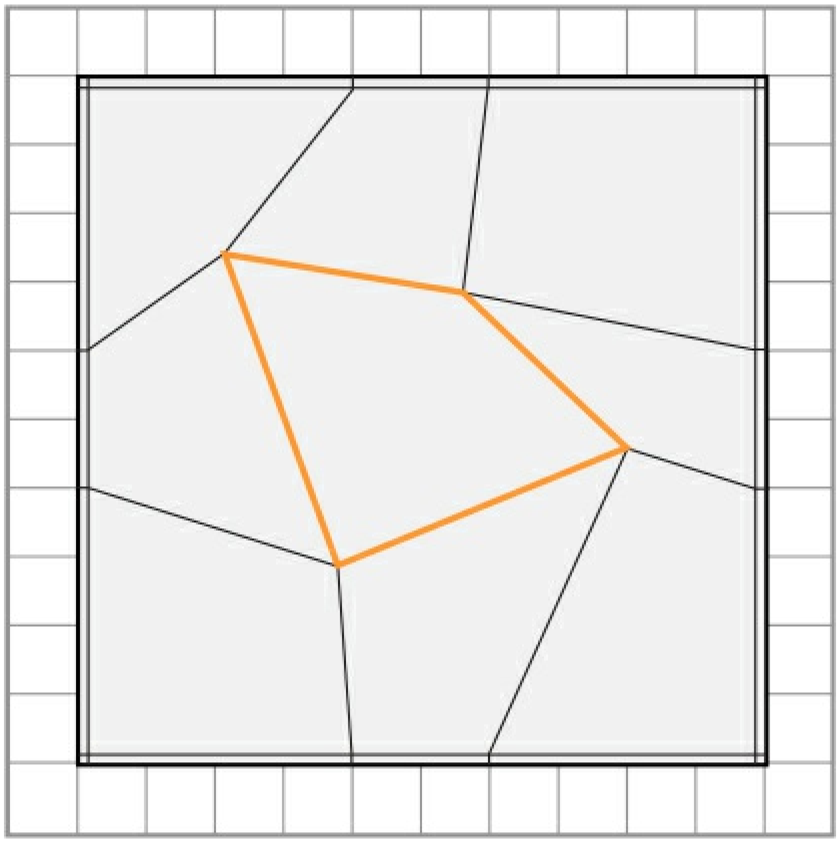
2. Add a new point to each edge, called the *edge point*.

3. Move the control point to another position, called the *vertex point*.

4. Connect the new points.

Figure 14.2 shows an example of a face before tessellation. The face point needs to be located at the average position of all the control points for the face. This position is determined by adding all the positions of the control points in that face, then dividing the total by the number of control points. For each edge point, you need to take the average of the two control points on either side of the edge, and the face points of the touching faces.

Figure 14.2. *A patch before Catmull-Clark subdivisions*



Source: www.rorydriscoll.com/2008/08/01/catmull-clark-subdivision-the-basics

Setting up the vertex points is more involved than setting up the face and edge points. Instead of utilizing a normal average as you did with the previous points, you need to calculate a weighted average. The equation to determine where the control point gets moved to:

$(Q/n) + (2R/n) + (S(n - 3)/n)$

The variables in this equation stand for the following:

- Q is the average of the surrounding face points.

- n is the valence, which is a fancy term for the number of edges that connect to that point.

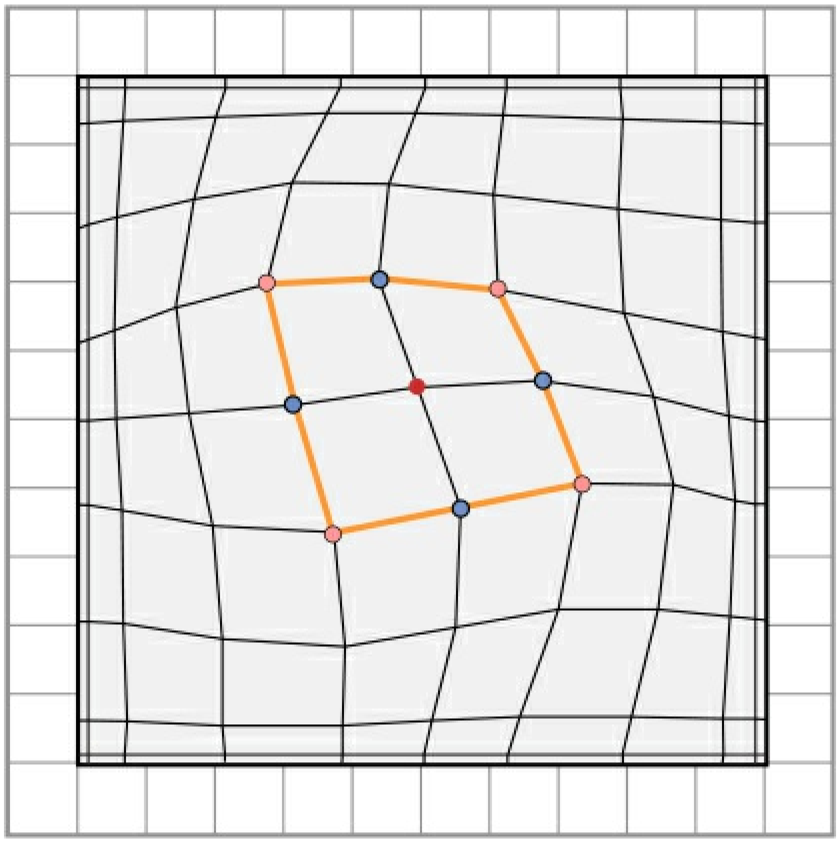
- R is the average of all surrounding edge midpoints.

- S is the control point.

In this weighted equation, the most weight is given to the average of all surrounding edge midpoints. If many edges connect to the point, the control point becomes more significant. If there is a low number of edges, the contribution of the average of the surrounding face points and the control point are pretty equal but still less than the average of the surrounding edge midpoints.

Finally, connect all the points. This smooths the original image and creates a much larger number of faces, as shown in Figure 14.3.

Figure 14.3. *A patch after Catmull-Clark subdivisions*



Source: www.rorydriscoll.com/2008/08/01/catmull-clark-subdivision-the-basics

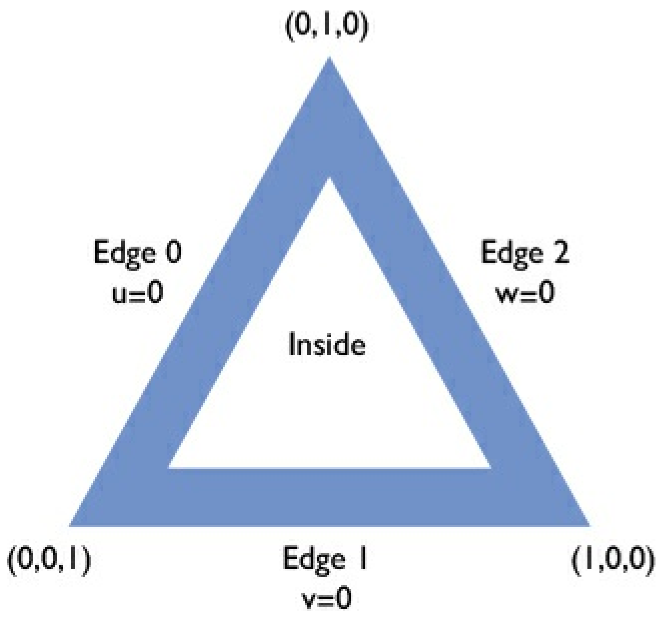
Per-Patch Tessellation Factors

One important aspect of tessellation is to specify how much additional detail you want. This is where *tessellation factors* come in. The tessellation factor is the number of subdivisions per patch. The more subdivisions you have, the more detailed the geometry becomes and the smoother the model gets. There is a bit of a performance cost, so you need to choose the smallest number you need to get the effect that you want.

There are two different types of tessellation patches you will subdivide in Metal: triangles and quadrilaterals. There is a Metal struct for each type: MTLTriangleTessellationFactorsHalf and MTLQuadTessellationFactorsHalf. In addition to the shapes of the patches, you also specify tessellation factors according to whether it’s an edge or an inside the shape.

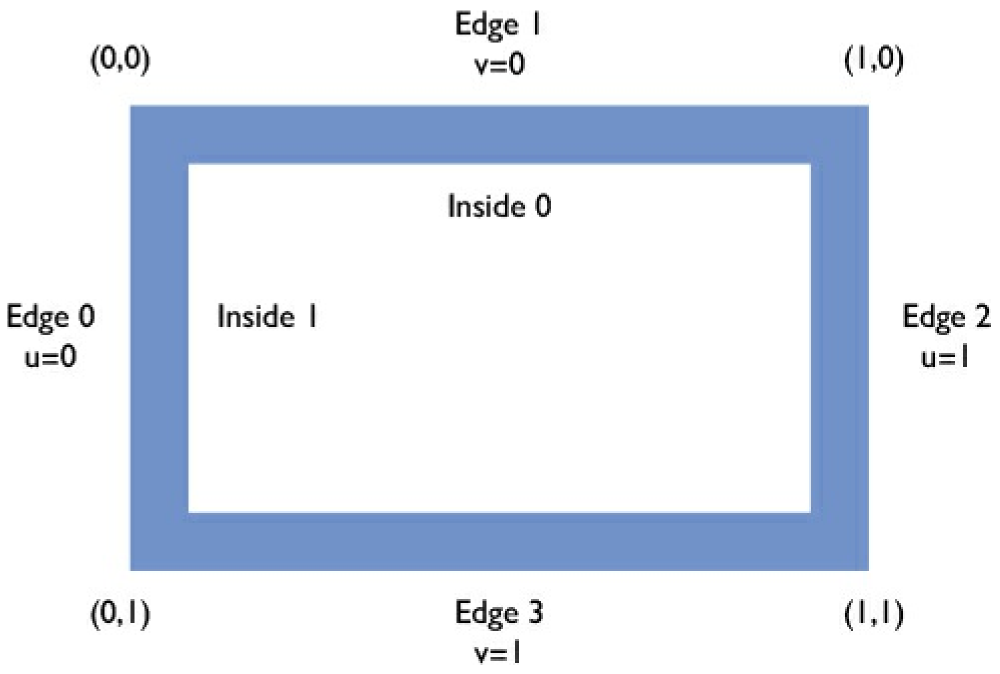
MTLTriangleTessellationFactorsHalf has two properties: edgeTessellationFactor and insideTessellationFactor. edgeTessellationFactor specifies a factor for each edge within the triangle. This property is an array with three values. For triangle patches, the position in the patch is a (u, v, w) coordinate that indicates the relative influence of the three vertices of the triangle on the position of the vertex, as shown in Figure 14.4. The (u, v, w) values range from 0.0 to 1.0, where u+v+w=1.0. The first member of the array is the tessellation factor for the u edge, followed by the v edge, then finally the w edge. Triangle patches have only one inside tessellation factor to set, so it is relatively straightforward.

Figure 14.4. *Triangle patch tessellation factors*



For MTLQuadTessellationFactorsHalf, the position in the patch is a (u, v) Cartesian coordinate that indicates the horizontal and vertical position of the vertex relative to the quad patch bounds. The (u, v) values range from 0.0 to 1.0 each, as shown in Figure 14.5. This struct's edgeTessellationFactor is an array of four values. The first value is for the edge of the patch where u=0 (edge 0). This is followed by the tessellation factor for the edge of the patch where v=0 (edge 1), then u=1 (edge 2), and finally v=1 (edge 3). Unlike the triangle patch, quad patches specify two inside tessellation factors. The first value in the array provides the horizontal tessellation factor for all internal values of v. The second value in the array provides the vertical tessellation factor for all internal values of u.

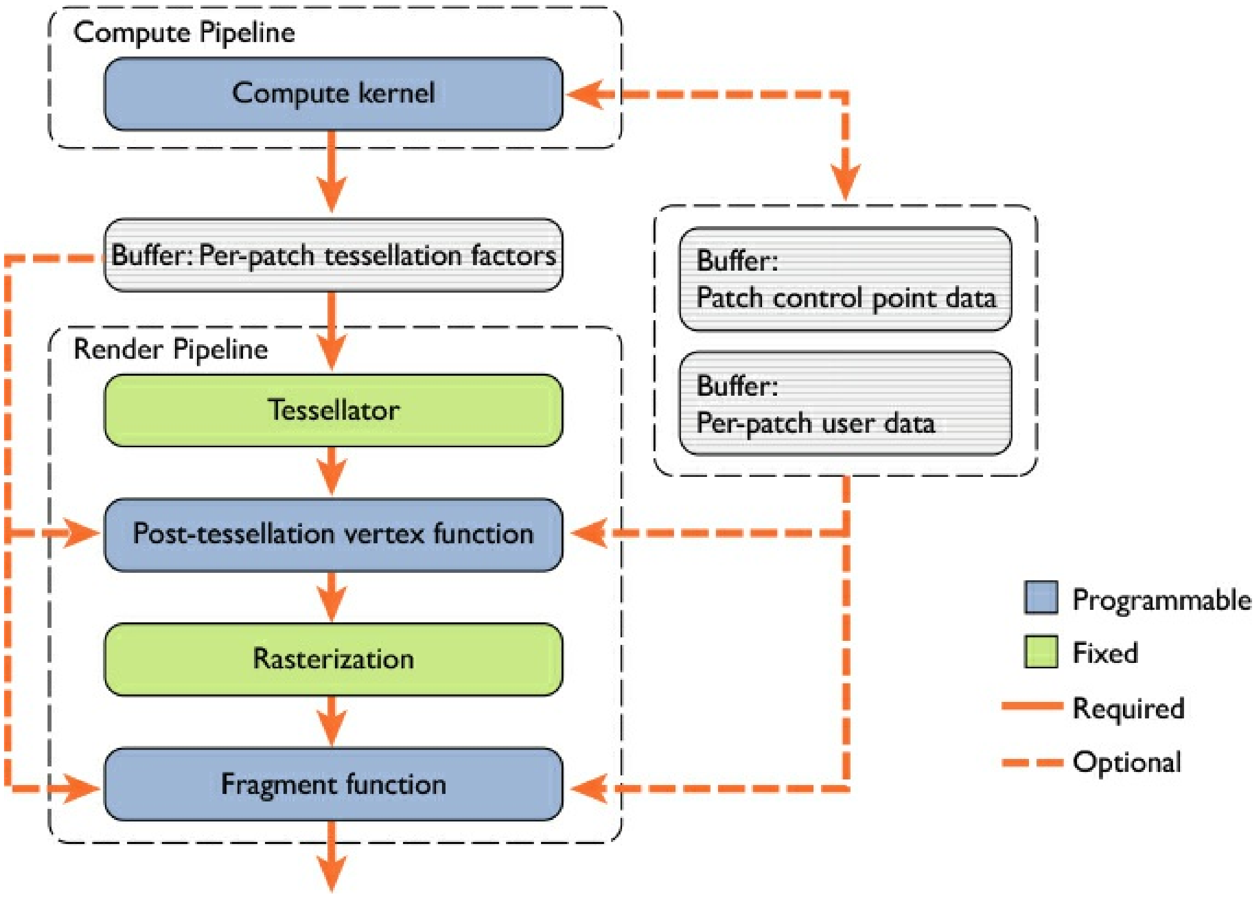
Figure 14.5. *Quad patch tessellation factors*



Metal Tessellation Fixed-Function Pipeline

The Metal tessellation pipeline, which is presented in Figure 14.6, differs from every other pipeline you will learn about in this book in that it requires an instance of all three shader types: vertex, fragment, and kernel. This chapter gives a brief overview of the role the kernel shader plays in the tessellation pipeline. There is a far more comprehensive overview of the kernel shader and the compute pipeline in the second section of this book.

Figure 14.6. *The Metal tessellation pipeline*



Before your buffers hit the vertex shader, you need to implement a compute kernel. If you remember the explanation of tessellation earlier, you know that the geometry initially entering the tessellation pipeline is quite primitive. It makes sense that the tessellation kernel is the first step in this pipeline because if you sent your primitive geometry directly to the vertex shader, it wouldn't get you anywhere.

The tessellation pipeline consists of some fixed-function stages and some programmable stages. The *tessellator* is the fixed-function stage. It creates a sampling pattern of the patch surface and generates graphics primitives that connect these samples. These fixed-function options are set when you create your MTLRenderPipelineDescriptor. Here is an example of how a render pipeline descriptor would look in an application that uses tessellation:

```
let renderPipelineDescriptor = MTLRenderPipelineDescriptor()
renderPipelineDescriptor.vertexDescriptor = vertexDescriptor
renderPipelineDescriptor.colorAttachments[0].pixelFormat =
     mtkView.colorPixelFormat
renderPipelineDescriptor.fragmentFunction =
    library?.makeFunction(name: "tessellation_fragment")
renderPipelineDescriptor.isTessellationFactorScaleEnabled = false
renderPipelineDescriptor.tessellationFactorFormat = .half
renderPipelineDescriptor.tessellationControlPointIndexType = .none
renderPipelineDescriptor.tessellationFactorStepFunction = .constant
renderPipelineDescriptor.tessellationOutputWindingOrder =
    .clockwise
renderPipelineDescriptor.tessellationPartitionMode =
    .fractionalEven
renderPipelineDescriptor.maxTessellationFactor = 64;
renderPipelineDescriptor.vertexFunction =
    library?.makeFunction(name: "tessellation_vertex_triangle")
var renderPipeline: MTLRenderPipelineState?
do {
    renderPipeline = try device.makeRenderPipelineState(
        descriptor: renderPipelineDescriptor)
} catch let error as NSError {
    print("render pipeline error: " + error.description)

}
```

The maxTessellationFactor specifies the maximum tessellation factor to be used by the tessellator when tessellating a patch. The default factor is 16, and the maximum factor is 64. If you choose a value between 16 and 64, you need to set the tessellationPartitionMode property, which derives the number and spacing of segments used to subdivide a corresponding edge. These modes are included in the MTLTessellationPartitionMode enum:

• **pow2**: This is the default value. The result is rounded up to the nearest integer n, where n is a power of two. For example, if the maxTessellationFactor is set to 30, the result would be rounded up to 32.

• **integer**: The result is rounded up to the nearest integer n. This integer could be either even or odd.

• **fractionalOdd**: The tessellation level is rounded up to the nearest odd integer n. If n is 1, the edge is not subdivided. Otherwise, the corresponding edge is divided into n-2 segments of equal lengths.

• **fractionalEven**: The tessellation level is rounded up to the nearest even integer n.

Another fixed function property is isTessellationFactorScaleEnabled. Its default value is false. If this value is true, a scale factor is applied to the tessellation factors after the patch cull check is performed but before the tessellation factors are clamped to the value of maxTessellationFactor. The scale factor is applied only if the patch is not culled.

The tessellationFactorFormat is the format of the tessellation factors specified in the tessellation factor buffer. This is an instance of MTLTessellationFactorFormat, and it must have the default value of .half.

The tessellationControlPointIndexType is the size of the control point indices in a control point index buffer. You can choose .none as an option, but this option should be used only when drawing patches without a control point index buffer. Otherwise, your options are 16-bit or 32-bit unsigned integers.

The tessellationFactorStepFunction determines the tessellation factors for a patch from the tessellation factor buffer. It has four options:

• **constant**: The default. For all instances, the tessellation factor for all patches in a patch draw call is at the offset location in the tessellation factor buffer.

• **perPatch**: A per-patch step function. For all instances, the tessellation factor for all patches in a patch draw call is at the offset + (drawPatchIndex * tessellationFactorStride) location in the tessellation factor buffer.

• **perInstance**: A per-instance step function. For a given instance ID, the tessellation factor for a patch in a patch draw call is at the offset + (instanceID * instanceStride) location in the tessellation factor buffer.

• **perPatchAndPerInstance**: A per-patch and per-instance step function. For a given instance ID, the tessellation factor for a patch in a patch draw call is at the offset + (drawPatchIndex * tessellationFactorStride + instanceID * instanceStride) location in the tessellation factor buffer.

The final property you need to set for the tessellator is tessellationOutputWindingOrder. As is clear from its name, it sets the winding order of triangles output by the tessellator. It is represented by the MTLWinding enum. The only options for this are clockwise and counterclockwise. The default is clockwise.

Setting Up a Tessellation Kernel

Your compute kernel is the programmable part of the pipeline. The compute kernel needs to perform the following functions:

• Compute per-patch tessellation factors

• Compute per-patch user data (optional)

• Compute or modify patch control point data (optional)

As with every shader function you have in Metal, you need to add a few data structures to the file to receive data set up in the main program and stored in the argument table:

```
// Control Point struct
struct ControlPoint {
    float4 position [[attribute(0)]];
};

// Patch struct
struct PatchIn {
    patch_control_point<ControlPoint> control_points;
};
```

The first structure receives the control points stored in the attribute buffer. These control points compose a patch that is fed to the vertex shader.

```
// Triangle compute kernel
kernel void tessellation_kernel_triangle(
    constant float& edge_factor [[ buffer(0) ]],
    constant float& inside_factor [[ buffer(1) ]],
    device MTLTriangleTessellationFactorsHalf* factors [[buffer(2)]],
    uint pid [[ thread_position_in_grid ]])
{
    // Simple passthrough operation
    factors[pid].edgeTessellationFactor[0] = edge_factor;
    factors[pid].edgeTessellationFactor[1] = edge_factor;
    factors[pid].edgeTessellationFactor[2] = edge_factor;
    factors[pid].insideTessellationFactor = inside_factor;
}
```

The kernel function is simply taking in the edge factors and the inside factor that were set in the main program and setting those into the MTLTriangleTessellationFactorsHalf property in the argument table. After this, the per-patch tessellation factors get sent to the tessellator. The results of the tessellator, along with the buffer of per-patch user data and per-patch tessellation factors are referenced by the post-tessellation vertex function.

Post-Tessellation Vertex Function

Post-tessellation vertex functions are similar to regular vanilla vertex functions, but they have to be equipped to receive the patch type and the control points. A post-tessellation vertex function must be at least one of the following types of arguments:

• Resources such as buffers (declared in the device or constant address space), textures, or samplers

• Per-patch data and patch control point data, which are either read directly from buffers or passed to the post-tessellation vertex function as inputs declared with the [[stage_in]] qualifier

• The patch identifier, qualified as [[patch_id]]

• The per-instance identifier, qualified as [[instance_id]]

• The base instance value added to each instance identifier before reading the per-instance data ([[base_instance]])

• The location on the patch being evaluated, qualified as [[position_in_patch]]

Here is a simple pass through post-tessellation vertex function:

```
// Triangle post-tessellation vertex function
[[patch(triangle, 3)]]
vertex FunctionOutIn tessellation_vertex_triangle(
    PatchIn patchIn [[stage_in]],
    float3 patch_coord [[ position_in_patch ]])
{
    // Barycentric coordinates
    float u = patch_coord.x;
    float v = patch_coord.y;
    float w = patch_coord.z;

    // Convert to Cartesian coordinates
    float x = u * patchIn.control_points[0].position.x +
        v * patchIn.control_points[1].position.x + w *
        patchIn.control_points[2].position.x;
    float y = u * patchIn.control_points[0].position.y +
        v * patchIn.control_points[1].position.y + w *
        patchIn.control_points[2].position.y;

    // Output
    FunctionOutIn vertexOut;
    vertexOut.position = float4(x, y, 0.0, 1.0);
    vertexOut.color = half4(u, v, w, 1.0);
    return vertexOut;
}
```

This function uses the control point struct created earlier in the chapter to pass in the control points. It also keeps track of the current position in the patch. In the proper shader function, it translates the object space coordinates to Cartesian coordinates and outputs the vertices to the rasterizer, as usual.

Draw Patches

Draw calls for tessellation are slightly different than they are for other render encoder applications. In normal applications, you tell the encoder what kind of object you're drawing (point, line, or triangle). Instead of drawPrimitives, you use a variation on drawPatch:

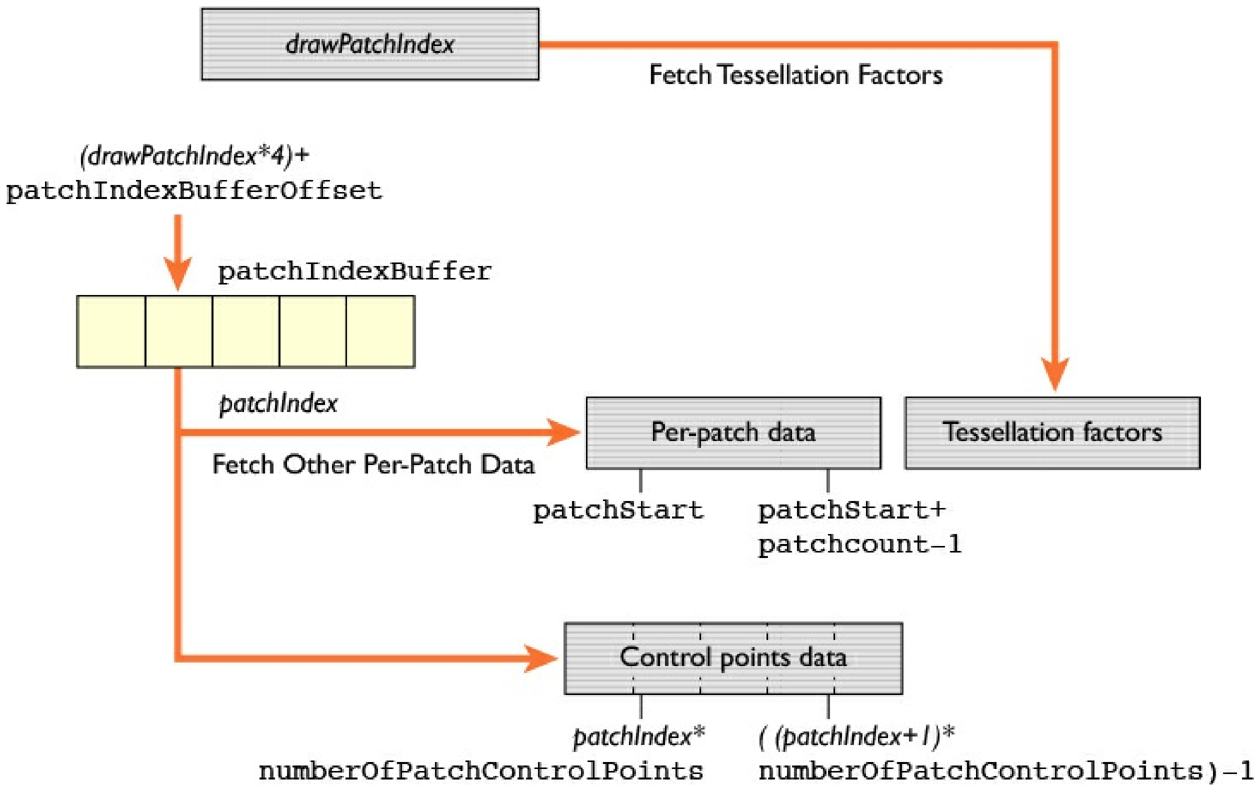
```
renderCommandEncoder.drawPatches(numberOfPatchControlPoints: 3,
                                             patchStart: 0,
                                             patchCount: 1,
                                       patchIndexBuffer: nil,
                                 patchIndexBufferOffset: 0,
                                          instanceCount: 1,
                                           baseInstance: 0)
```

There are four flavors of draw calls for tessellation patches. The first and most straightforward draw patch call is the one detailed above. A more complex draw patch is as follows:

```
func drawPatches(numberOfPatchControlPoints: Int,
                             patchStart: Int,
                             patchCount: Int,
                       patchIndexBuffer: MTLBuffer?,
                 patchIndexBufferOffset: Int,
                          instanceCount: Int,
                           baseInstance: Int)
```

This call renders a number of instances of tessellated patches, as seen in Figure 14.7. The first parameter is the number of patch control points. This value must be between 0 and 32. The second parameter is the start index for the patch. The next value is the number of patches in each instance. Next, you specify the MTLBuffer containing the patch indices if one exists.

Figure 14.7. *Per-patch draw call*



If there is an offset, you set that as well. Next, you specify the number of instances you want to draw, and finally the index of the first instance to draw.

Another way of setting up this draw call is to use an indirect buffer:

```
func drawPatches(numberOfPatchControlPoints: Int,
                       patchIndexBuffer: MTLBuffer?,
                 patchIndexBufferOffset: Int,
                         indirectBuffer: MTLBuffer,
                   indirectBufferOffset: Int)
```

You’ll notice this method signature is slightly shorter. A number of properties are moved to the indirectBuffer argument. This utilizes the MTLDrawPatchIndirectArguments struct. It contains the base instance, instance count, patch start, and patch count.

Another version of this draw call not only requires this indirect buffer but also uses a control point index buffer instead of patches:

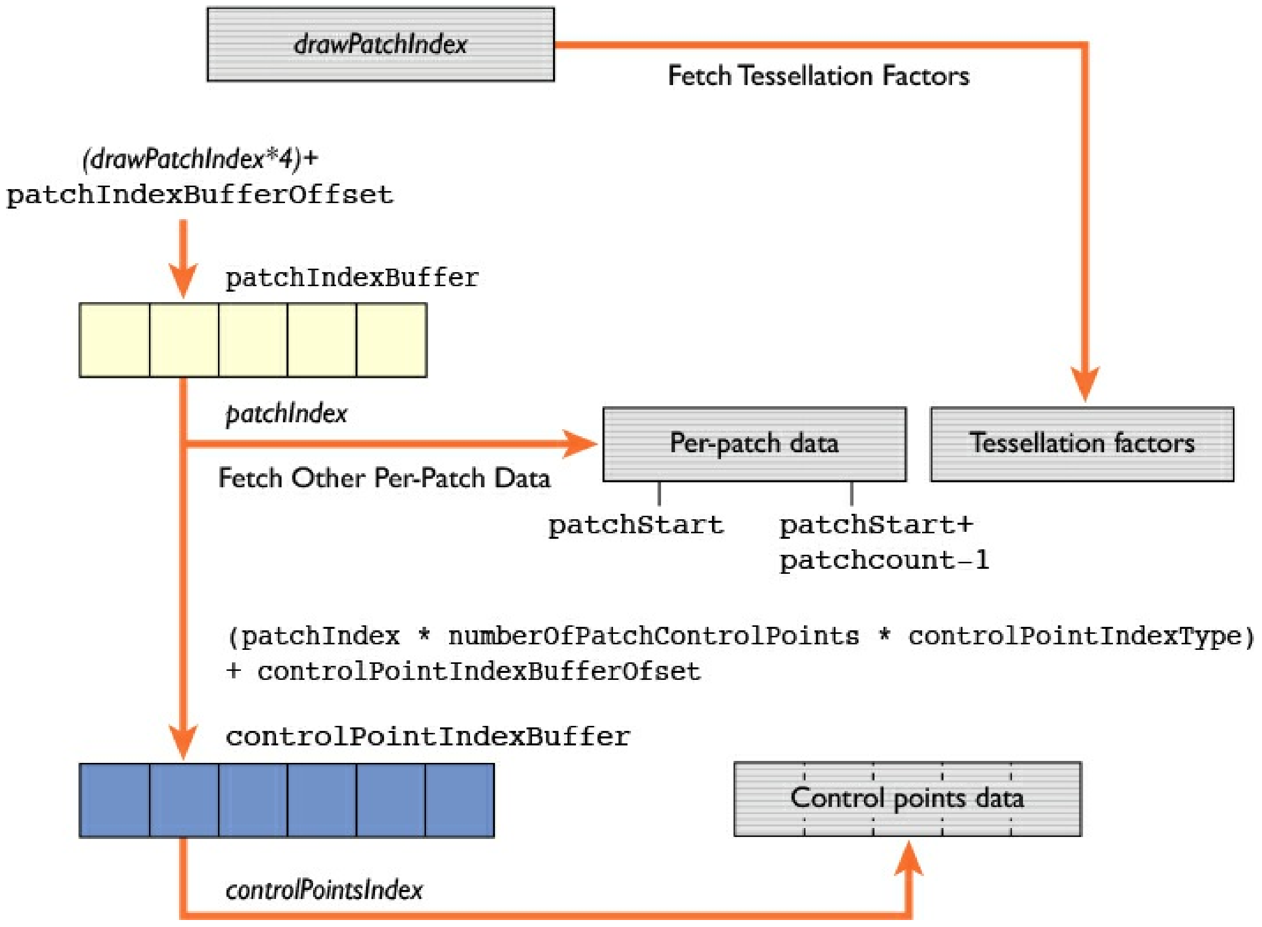
```
func drawIndexedPatches(numberOfPatchControlPoints: Int,
                              patchIndexBuffer: MTLBuffer?,
                        patchIndexBufferOffset: Int,
                       controlPointIndexBuffer: MTLBuffer,
                 controlPointIndexBufferOffset: Int,
                                indirectBuffer: MTLBuffer,
                          indirectBufferOffset: Int)
```

Pay attention to the fact that not only do you need to include a buffer of patch indices, but you also need to include one for your control points, as shown in Figure 14.8.

If you want to take advantage of the control points in a draw call without having to use an indirect buffer, one last method is available:

```
func drawIndexedPatches(numberOfPatchControlPoints: Int,
                                        patchStart: Int,
                                          patchCount: Int,
                                    patchIndexBuffer: MTLBuffer?,
                              patchIndexBufferOffset: Int,
                             controlPointIndexBuffer: MTLBuffer,
                       controlPointIndexBufferOffset: Int,
                                       instanceCount: Int,
                                        baseInstance: Int)
```

Figure 14.8. *Control point buffer*



Summary

Tessellation is a sophisticated way to add nuance and detail to your models without having to load down the GPU with millions of polygons. Meshes are tessellated on the basis of Catmull-Clark subdivisions to procedurally generate additional geometry to smooth out roughly sketched models. This process uses all three types of Metal shader functions available in the framework.

III: Data Parallel Programming

15. The Metal Compute Pipeline

Contrariwise, if it was so, it might be; and if it were so, it would be; but as it isn’t, it ain’t. That’s logic.

—Lewis Carroll

One of the big promises of Metal is that it allows programmers to do general-purpose GPU programming (GPGPU programming). Over the years, as GPUs have gotten more powerful, clever programmers have realized that they can offload to the GPU a lot of work that is not necessarily graphics related. This chapter focuses on GPGPU programming in Metal.

Metal Framework Concepts

Even though this chapter specifically discusses GPGPU programming, many previous chapters introduced concepts you are expected to be familiar with before you read this chapter. If your primary interest is GPGPU programming and you skipped directly to this chapter, skim these chapters before proceeding:

• Chapter 4, “Essential Mathematics for Graphics”

• Chapter 5, “Introduction to Shaders”

• Chapter 6, “Metal Resources and Memory Management”

• Chapter 7, “Libraries, Functions, and Pipeline States”

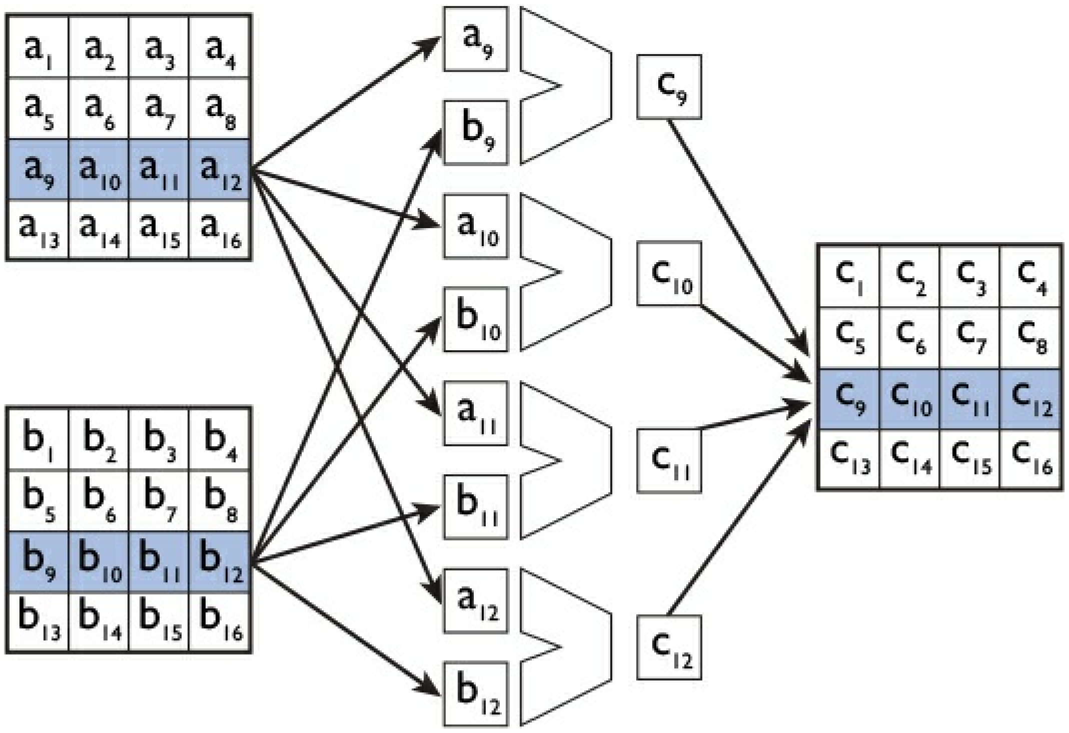
Introduction to GPU Programming

Over the last 10 years, GPU programming has become a buzzword for throughput and speed. What exactly is GPU programming, and why is it so fast? The first piece of this puzzle is to look at what differentiates the GPU from the CPU.

The *central processing unit (CPU)* is the brain of your computer. It’s responsible for everything on your computer. A large portion of your CPU must be dedicated to on-chip cache memory. Consequently, the CPU can dedicate only a limited amount of space to computational logic. The GPU has the flexibility to be far more specialized. The GPU maxes out its available real estate for computational logic.

GPUs maximize their real estate with *arithmetic logic units (ALUs)*. An ALU is a combinational digital electronic circuit that performs arithmetic and bitwise operations. On the CPU, these operations are primarily integer binary numbers. On the GPU, these operations are primarily floating-point numbers. We humans think of a mathematical operation differently than a computer does. To us, multiplying two numbers is a single operation, but to a computer, it involves multiple operations on the bit level. By having a wide memory bus and many ALUs, the GPU can fetch more operations at a time, as shown in Figure 15.1. By scaling up, the GPU can be more efficient for large numbers of operations than it is on just one or two.

Figure 15.1. *ALUs processing in parallel*


a_1 a_2 a_3 a_4
a_5 a_6 a_7 a_8
a_9 a_{10} a_{11} a_{12}
a_{13} a_{14} a_{15} a_{16}
b_1 b_2 b_3 b_4
b_5 b_6 b_7 b_8
b_9 b_{10} b_{11} b_{12}
b_{13} b_{14} b_{15} b_{16}
c_9 c_{10} c_{11} c_{12}
c_1 c_2 c_3 c_4
c_5 c_6 c_7 c_8
c_{13} c_{14} c_{15} c_{16}

CPUs also have a limited number of cores on each chip. Having more than one core enables a chip to implement parallel programming. Each core can complete only one task at a time, so a dual core processor, for example, can perform twice the amount of work as a single-core processor at any given point in time. Multiple cores, however, come with added complexity, because the cores have to coordinate how they process their tasks without overwriting memory that is being accessed by both cores.

GPUs take this parallel processing to a new level. Rather than being limited to two, four, or even a dozen cores, GPUs have thousands of cores. Therefore, GPUs can execute thousands of commands simultaneously. This design enables incredibly fast performance, but it also requires a great deal of coordination and organization on the part of the programmer.

Concurrency versus Parallelism

There is some confusion about the difference between concurrency and parallelism. They are sometimes used interchangeably, but they are different concepts.

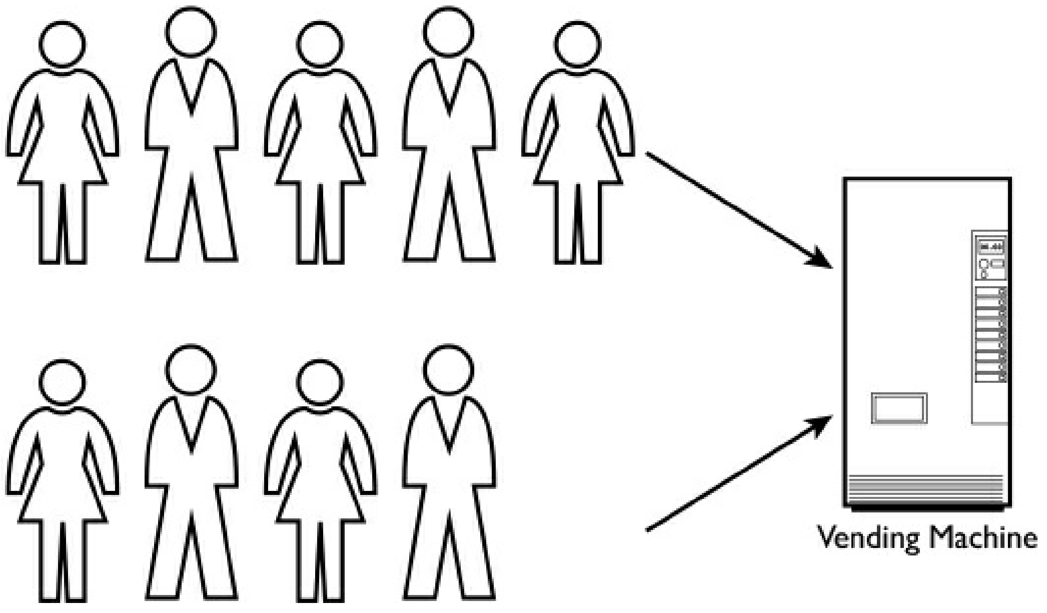
If you’ve ever paused a program running in the Xcode simulator and looked at a stack trace, there are usually around 16 threads running at any given time. You didn’t create these threads—they were implemented behind the scenes by the Cocoa frameworks. These threads are not all processing at the same time.

Say that you’re an independent game developer. On any given day, you could be writing code, designing artwork, and marketing, but you can’t do all of these things at the same time. You’re doing these actions concurrently. You are like a CPU running three threads of work, but you can only do them one at a time.

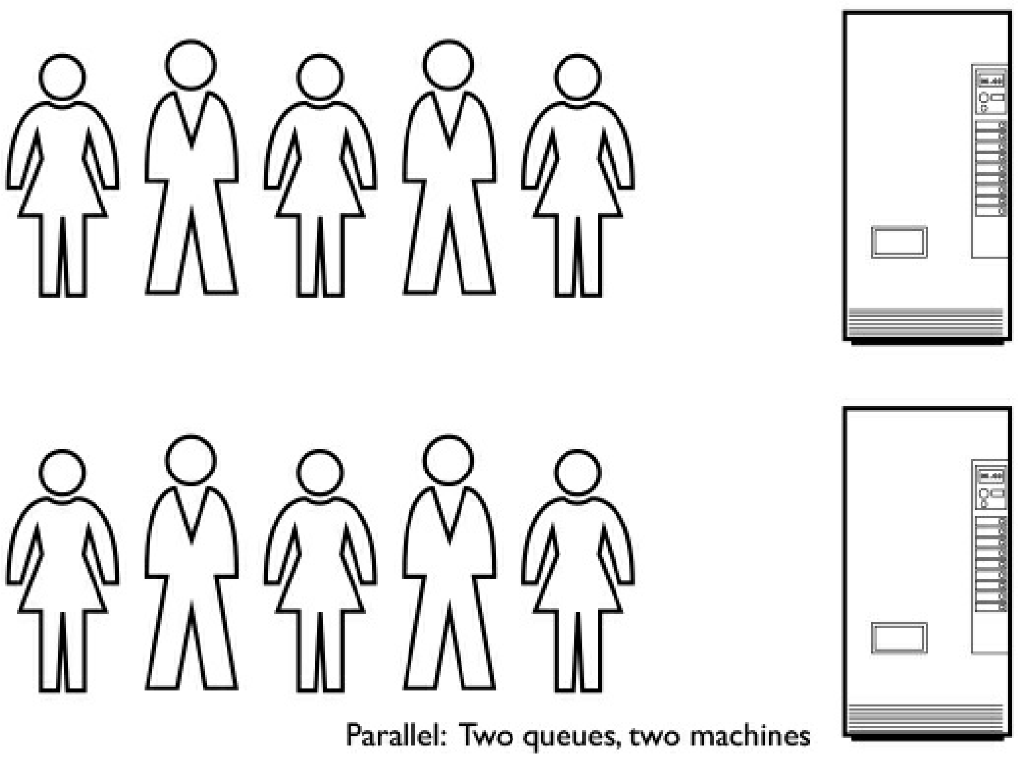
Let’s say you hire someone to create your artwork. Having someone else do the artwork frees you up to do other work. You can optimize this arrangement if you have the artist work in parallel with you so that when your other work is finished and you are ready for your artwork, it’s already waiting for you.

Because you can execute only one thread at a time, you gain a lot of efficiency by delegating work to the artist. Now imagine you have an entire studio of artists who can work on different parts of your game all at the same time. The GPU is like that animation studio. Because the GPU has thousands of cores, the more work you can send to the GPU, the more work can be done simultaneously, as shown in Figure 15.2.

Figure 15.2. *Concurrency versus parallelism*



Concurrency: Two queues, one machine



Parallel: Two queues, two machines

You may be wondering why you have multiple threads if only one task can be performed at any given moment. Different tasks can be put on different threads that have different priorities. You're familiar with the admonishment not to put all of your work on the main thread. The main thread's priority is user interaction. You don't want to block user interaction until a networking call has completed. Any non-user interface function call can be put on a background thread with lower priority.

Apple's Grand Central Dispatch (GCD) framework monitors all of your threads and schedules work. It checks to see if the user is interacting with the application. If the user begins interacting, GCD stops all other work to deal with the requirements of the main thread. Once the work is completed, GCD can go back to scheduling other tasks to be done in the background. The A9 chip is so powerful that all of this happens seamlessly without the user being aware of all the resource juggling going on behind the scenes.

There are many programming models you can use to create parallelism. Parallelism can be built into computer architecture, on the data level, and on the instruction level. GPUs are built on data parallelism. CPU parallelism is beyond the scope of this book, but the main takeaway is that the GPU is capable of being far more parallel than the CPU, and sending work to it gives you huge performance advantages.

Using GPUs for General Computation

One big question you may have is, What kind of work can the GPU do? You know you can't send network calls to the GPU to be processed, because the GPU doesn't do that kind of work. So what kinds of work can it do?

The GPU is designed to perform lots of math operations in parallel. This entails multiplying every element in a 1,000-element array of numbers by two, for example. It also entails applying matrix multiplication and matrix operations to large data sets. There are many scientific and mathematical disciplines that are built on linear algebra concepts. These include, but are not limited to the following:

• 2D and 3D computer graphics

• Image processing

• Cryptography

• Markov chains

• Neural networks

• Economics

Any mathematical operation that can be processed in a matrix can be performed in parallel on the GPU. Any large data sets that need to have a consistent operation performed on every member can be performed on the GPU. The GPU does not deal with conditional logic very well, so operations with a lot of if/else statements should be avoided at all costs.

To get an idea about what operations are easy to perform on the GPU, look through the Metal Shading Language specification. As to what discipline you apply it to, that is up to you. Math is the foundation of many scientific disciplines, so don't limit yourself to the common use cases found on the Internet.

Kernel Functions

In Chapter 5, "Introduction to Shaders," you became familiar with vertex and fragment shaders. There is still one other type of shader function: the kernel shader, which is solely for Metal compute programs. With compute programs, unlike with 3D graphics programs, you don't have to worry about rendering. You don't have a buffer of vertex data that needs to be translated from object space to world space, so you don't need a separate vertex shader to feed your vertex data to the kernel shader. In many ways, the compute pipeline is much simpler than the graphics rendering pipeline.

Kernel functions are also the only MTLFunction type that is set up for data parallel computing. As discussed in the section, "Concurrency versus Parallelism," the GPU has thousands of cores. The kernel function can execute its computation over a 1-, 2- or 3-dimensional grid.

Kernel functions must also have a void return type. Vertex and fragment functions exist on a rendering pipeline, which means that the result of each function must be returned and passed to the next section of the pipeline. Kernel functions do not return output. They apply mathematical operations over a buffer of values and can be processed in place.

Here is a prototypical example of a kernel function signature:

```
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 function must be prefaced with the name kernel to differentiate it from a vertex or fragment function. The return type must be void. This is followed by the name of the kernel function. As you do with other types of functions, you pass in locations of arguments from the argument table as arguments to your kernel function.

The Metal Compute Command Encoder

All work sent to the GPU is encoded into a MTLCommandEncoder instance. There are four different types of command encoders. So far in this book, you've learned how to use three of them. This chapter introduces the fourth and final one: the Metal compute command encoder (MTLComputeCommandEncoder).

Setting up the MTLComputeCommandEncoder is a bit more involved than simply instantiating it like any other object. It requires several steps and setting up a few other objects first:

```
var device: MTLDevice!
var commandQueue: MTLCommandQueue!
var library: MTLLibrary!

func setupMetal() {
    device = MTLCreateSystemDefaultDevice()
    commandQueue = device.makeCommandQueue()
    library = device.newDefaultLibrary()
}
```

The method to set up the MTLComputeCommandEncoder, makeComputeCommandEncoder(), exists on the MTLCommandBuffer object. The MTLCommandBuffer object is initialized by the makeCommandBuffer() method on the MTLCommandQueue. The MTLCommandQueue in turn is created by the makeCommandQueue() method on the MTLDevice. Consequently, to initialize a MTLComputeCommandEncoder, you need to set up a chain of other Metal objects, all of which initialize the next one in the chain.

After you get all three of these objects initialized and in place, you can call the method on MTLCommandBuffer to initialize the MTLComputeCommandEncoder:

```
let commandBuffer = commandQueue.makeCommandBuffer()
let computeEncoder = commandBuffer.makeComputeCommandEncoder()
```

This might seem a little confusing at first, but these are all object instances you would be required to create anyway. Try to look at it as a sanity check to ensure that you are setting up your objects properly.

Creating a Compute Pipeline State

As you do for the rendering pipeline, you create a pipeline state here for the compute pipeline. Setting up as much as possible up front is the best way to gain efficiency when programming the GPU. The first step is to create a compute pipeline state variable:

```
var computePipelineState: MTLComputePipelineState!
```

After you create that variable, you need to set it up:

```
func setupPipeline() {
    if let computeFunction = library.makeFunction(
        name: computeFunctionName) {
     do {
          computePipelineState = try device.makeComputePipelineState(
              function: computeFunction)
     }
         catch let error as NSError {
          fatalError("Error: \(error.localizedDescription)")
         }
     }
    else {
     fatalError("Kernel functions not found at runtime")
    }
}
```

The only piece you need to set up the state is the kernel function. You need to use this function to set up a Metal library to be used by the pipeline state. If for some reason you cannot set up that library, you want to ensure that the program fails. Finally, you need to encode the state into the compute pipeline.

```
computeEncoder.setComputePipelineState(computePipelineState)
```

So far, everything has been somewhat familiar. Up next, you’re going to learn how to set up parallel work groups, a feature that is unique to the compute encoder.

Issuing Grids of Work

This section covers how to set up parallel processing on the GPU. You can control a few moving parts that enable you to customize how the work gets processed. The choices that you make in this regard have a significant impact on the performance you’re able to coax from the GPU.

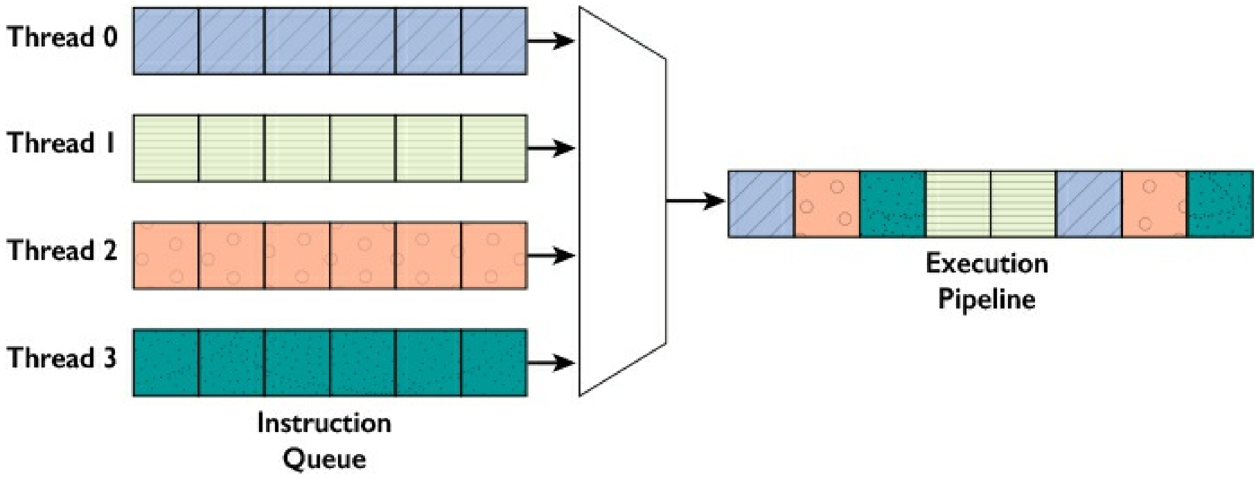
Tasks performed on the GPU are somewhat similar to any large project you have to accomplish. The project can be broken up into discrete tasks that can be accomplished by different actors in a group. Part of your job as the GPU’s project manager is to determine the optimal number of individuals needed to accomplish this work and how to divvy up those tasks among those actors.

Thread Groups

The individual units that perform work are known as *threads*. Threads are one of a collection of parallel executions of a kernel function invoked on a device by a dispatch command. A thread is distinguished from other executions within the collection by its thread position in grid and thread position in threadgroup.

It’s not efficient for the GPU to micromanage every single thread on the GPU. Instead, they are assembled in *threadgroups*. A thread is executed by one or more processing elements as part of a threadgroup executing on a compute unit, as seen in Figure 15.3. They also share an instruction pointer, which means that they all execute the same instruction at the same time. The number of threads that can be executed on a threadgroup is the *thread execution width*. As with many concepts in computer science, the maximum execution width is a power of two. It changes according to what version of the iPhone you are using. More powerful iPhones have deeper execution widths. This concept is explored in Chapter 16, “Image Processing in Metal.”

Figure 15.3. *Threads executed in a pipeline*



First, you need a property for both the threadgroup size and the threadgroup width:

```
var threadgroupSize, threadsPerThreadgroup: MTLSize!
```

Next, you need to calculate how large the threadgroup size and the threadgroup width will be:

```
func setupThreads() {
    let threadgroupWidth = 8
    let µ = side / threadgroupWidth +
        (side == threadgroupWidth ? 0 : 1)

    threadgroupSize = MTLSize(width: µ, height: µ, depth: 1)
    threadsPerThreadgroup = MTLSize(width: threadgroupWidth,
                               height: threadgroupWidth, depth: 1)
}
```

Lastly, you need to encode these into the compute command encoder:

```
computeEncoder.dispatchThreadgroups(threadgroupSize,
                                    threadsPerThreadgroup:
                                    threadsPerThreadgroup)
```

That is all the set up you need on the CPU side of things. The rest is set up on the GPU side.

Finding Your Way in the Grid inside the Kernel Function

In your main code, you set up the number of threads you have per threadgroup and the number of threadgroups per grid. Those are passed when you encode a grid of work. You also need to pass in the current thread position in the grid:

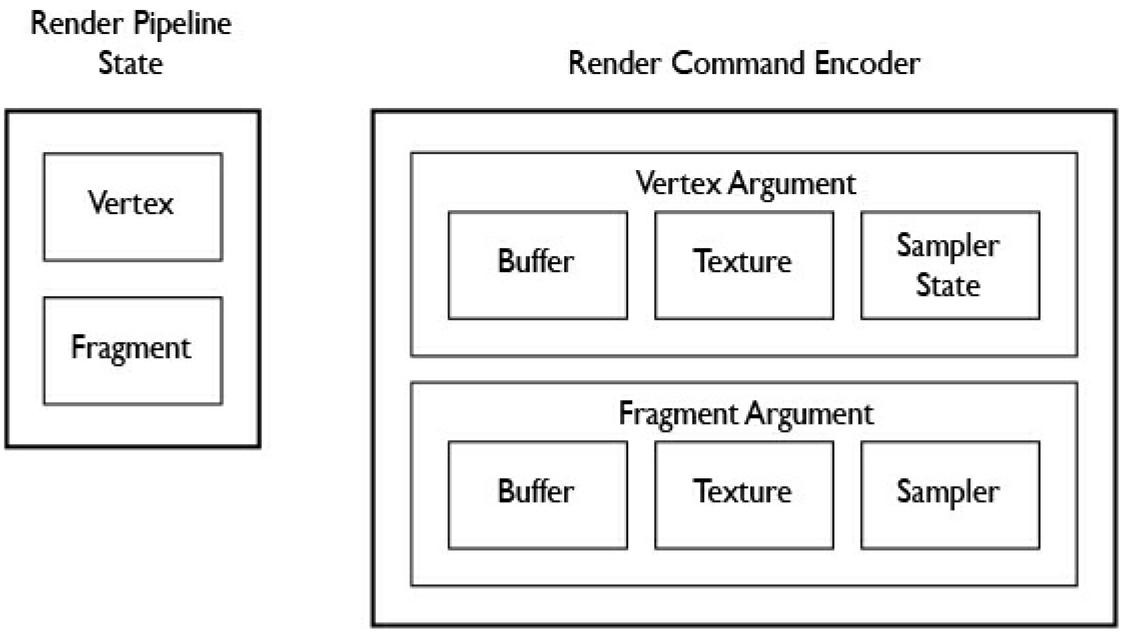
```
void kernel myKernel(
     uint2 S [[ threads_per_threadgroup ]],
     uint2 W [[ threadgroups_per_grid ]],
     uint2 z [[ thread_position_in_grid ]]) {
}
```

This is similar to the parameter in the fragment shader that tracks which pixel is currently processing. You are not responsible for computing the position of the current thread in the grid or in its respective threadgroup, but you can ask for these values by adding parameters to your kernel function that are attributed with the appropriate attributes.

Reading and Writing Resources in Kernel Functions

As you do with the render command functions, you need to coordinate resources between the CPU and the GPU. If you recall from earlier chapters, Metal utilizes an argument table, as shown in Figure 15.4, to hold resources that are shared between the CPU and the GPU. In the render encoder, three types of data can be encoded. For the compute encoder, there is an additional argument category.

Figure 15.4. *Compute encoder argument table*



Four kinds of data can be shared between the CPU and GPU in the compute pipeline:

• Buffer

• Texture

• Sampler

• Threadgroup memory

Threadgroup memory is a new category. The maximum number of entries you can have in the threadgroup memory section of the argument table is 31. Within those threadgroups, you have a maximum of 512 threads per threadgroup on iOS. This expands to 1024 threads on macOS. The maximum total threadgroup memory allocation for the argument table is 16,352 bytes. That doubles to 32 kilobytes on macOS. These bytes must be broken up into blocks of 16 bytes on both operating systems.

You learn more about threads and threadgroup management in Chapter 16.

Summary

The render command encoder and the compute command encoder are more similar than they are different. The compute encoder allows you to do general-purpose GPU programming on iOS for the first time since the platform was introduced. The compute encoder allows you to utilize parallel computing and not just concurrency. On the GPU, you can do high-performance data processing and mathematical operations because of the architecture differences between the CPU and the GPU.

16. Image Processing in Metal

Everything we see hides another thing, we always want to see what is hidden by what we see.

—René Magritte

One of the most common kernel processing operations is image processing. Image processing has been a popular GPU programming application since shaders were introduced. It encompasses a large number of uses from creating an interesting Snapchat filter effect to neural network training. Because you are basically just processing a texture, you don't need to set up an entire rendering pipeline to do image processing, as there are no vertices to position. Processing can be as simple as modifying a single fragment and as complex as looking at massive grids of pixels in relation to one another. This chapter introduces the options available to you and how to implement them.

Introduction to Image Processing

What is color? Color is a narrow band of the visible light spectrum. Slower light waves are red, and faster light waves are blue or violet. This is the physical reality of light and color, but it's not the reality that you, as a programmer, have to deal with, because it's not how the computer represents color.

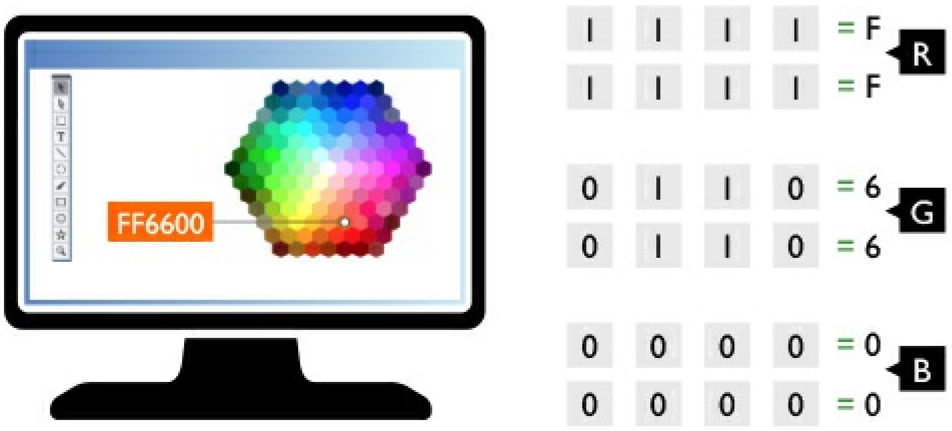
Computers are digital. Everything the computer understands is represented by numbers, and those numbers are represented differently to the computer than they are to us. Humans primarily use a base 10 counting system. You have 10 unique values and, on the next value, the number increments to another significant digit.

The computer is base 2, which means that after you get past 0 and 1, instead of going to 2, it goes to 10. Each value you have in memory, which can be a 0 or a 1, is known as a *bit*. In computer science, most storage methods and processes are based on powers of 2 because of the nature of bits. One bit can hold two separate values. Two bits can hold four values, and so on.

You might remember back in the old days when color monitors had only 256 different color values. Each pixel in a screen had only 8 bits to hold color data. If you've ever worked with Photoshop and wondered why you are choosing a color value between 0 and 255, it's because current hardware and software has 8 bits of data dedicated to each color channel. Groups of 8 bits constitute a *byte*.

All colors that you see on a screen are represented by three color channels: red, green, and blue, as shown in Figure 16.1. There is a fourth channel for the alpha value, which is how transparent the color is. This means that the computer has 32 bits of data to represent color on the screen. Because the computer can quantify this data in numerical form, these colors can be manipulated mathematically.

Figure 16.1. *Binary color representation*



Creating a Metal Texture

An image, at its core, is a collection of numbers that describe what color each pixel is at any particular location. Because colors can be translated to a consistent numerical system, we're able to process their numeric representations mathematically, which is really cool.

GPUImage

If you want a good library of image processing shaders to play with and translate to Metal, look into the open source GPUImage framework. Unlike Core Image, all the shaders are open source and exposed to the programmer. These shaders are written in GLSL, but the differences between GLSL and MSL are minimal. This is a fantastic shader resource.

The process, in a nutshell, involves taking an image of some form, converting it to a buffer of image data, processing it through a kernel, then reassembling it into an image that can be rendered on the screen. Because you're doing away with the rendering pipeline, this final step of displaying the image on the screen is up to you.

This method takes the name of an image and returns a MTLTexture object. Because this method may fail at some point along the way, the return type is optional. This is a long method, so it's been broken into two sections.

```
func textureForImage(imageName: String,
                     device:MTLDevice) -> MTLTexture? {

    let image = UIImage.init(named: imageName)!

    let imageRef = image.cgImage!

    let width = imageRef.width
    let height = imageRef.height
    let colorSpace = CGColorSpaceCreateDeviceRGB()

    let rawData = calloc(height * width * 4,
        MemoryLayout<UInt8>.size)

    let bytesPerPixel = 4
    let bytesPerRow = bytesPerPixel * width
    let bitsPerComponent = 8

    let options = CGImageAlphaInfo.premultipliedLast.rawValue|
        CGBitmapInfo.byteOrder32Big.rawValue
    let context = CGContext(data: rawData,
                            width: width,
                            height: height,
                            bitsPerComponent: bitsPerComponent,
                            bytesPerRow: bytesPerRow,
                            space: colorSpace,
                            bitmapInfo: options)

    context?.draw(imageRef, in: CGRect(x: 0,
         y: 0,
         width: CGFloat(width),
                                       height: CGFloat(height))
```

Most of this first set of code is setting up the CGContext. First, you need to create a UIImage from the name passed in. The UIImage cannot be used directly with a CGContext, and it needs to have the cgImage property referenced.

Next, you set up properties used to initialize the context object. These could be done inline, but that would be rather verbose. Next, you create the context with all the properties you've created for the width, height, and bit properties. Lastly, you draw the cgImage into the CGContext and set up the MTLTexture:

```
    let textureDescriptor = MTLTextureDescriptor.texture2DDescriptor(pixelFormat: .rgba8Unorm,
                                                  width: Int(width),
                                                 height: Int(height),
                                              mipmapped: true)

    let texture = device.makeTexture(descriptor: textureDescriptor)

    let region = MTLRegionMake2D(0, 0, Int(width), Int(height))

    texture.replace(region: region,
                    mipmapLevel: 0,
                    slice: 0,
                    withBytes: rawData!,
                    bytesPerRow: bytesPerRow,
                    bytesPerImage: bytesPerRow * height)

    free(rawData)

    return texture
}
```

MTLTexture objects can be processed by kernel functions the same way that they are processed by vertex and fragment functions. This resource can be added to an argument table and referenced by both the CPU and the GPU.

Now that you have a grasp on how to set up an image to be processed, let's take a look at a few examples of image processing.

Desaturation Kernels

One of the simplest image processing algorithms is a desaturation shader. Going over this shader gives you a good introduction to the moving parts of image processing. Saturation is the intensity of the color at any given pixel. A completely desaturated image is in black and white. In fact, a desaturation filter is usually used by hipsters to convert an image to black and white to make it look more artistic. Here is the code that goes into a desaturation filter:

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

```
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);
}
```

Outside the kernel is a struct that holds a saturation percentage. The struct is connected to a slider in the user interface that allows the user to adjust the level of saturation in the image.

Four arguments are being passed into the desaturation filter:

• Input texture

• Destination texture

• Amount of saturation

• Current thread position in the thread grid

The kernel shader is a little bit like the transporter from *Star Trek*. The kernel takes an original image/texture, performs a mathematical operation on each pixel, then translates it to a destination texture. That processing is dependent on the value set by the user in the user interface.

The inColor variable correlates to the RGBA value of the pixel that is currently being processed. The alpha is not affected by anything we are doing here, so it is ignored in the next line. The value variable generates the grayscale value of the current pixel. Take a closer look at how the value variable is calculated:

```
float value = dot(inColor.rgb, float3(0.299, 0.587, 0.114));
```

This is an algorithm for calculating luminance. *Luminance* is the relative brightness of any given pixel. Humans perceive luminance far more acutely than they do color. Humans also perceive green far more strongly than they do red or blue. This algorithm calculates the relative luminance of any given pixel and returns a completely desaturated pixel. Because you don't want a completely desaturated image, mix this desaturated color value with a saturated color value:

```
float4 grayColor(value, value, value, 1.0);
float4 outColor = mix(grayColor, inColor,
                      uniforms.saturationFactor);
outTexture.write(outColor, gid);
```

The rest of the shader creates a variable where each value is the relative luminance of the current pixel. The value is what you get when you add all the weighted values together to create a grayscale value that represents the luminance. This value is then mixed with the original red, green, and blue values of the pixel. The mix percentage is determined by the saturation factor. The result is then output to the destination texture, as shown in Figure 16.2.

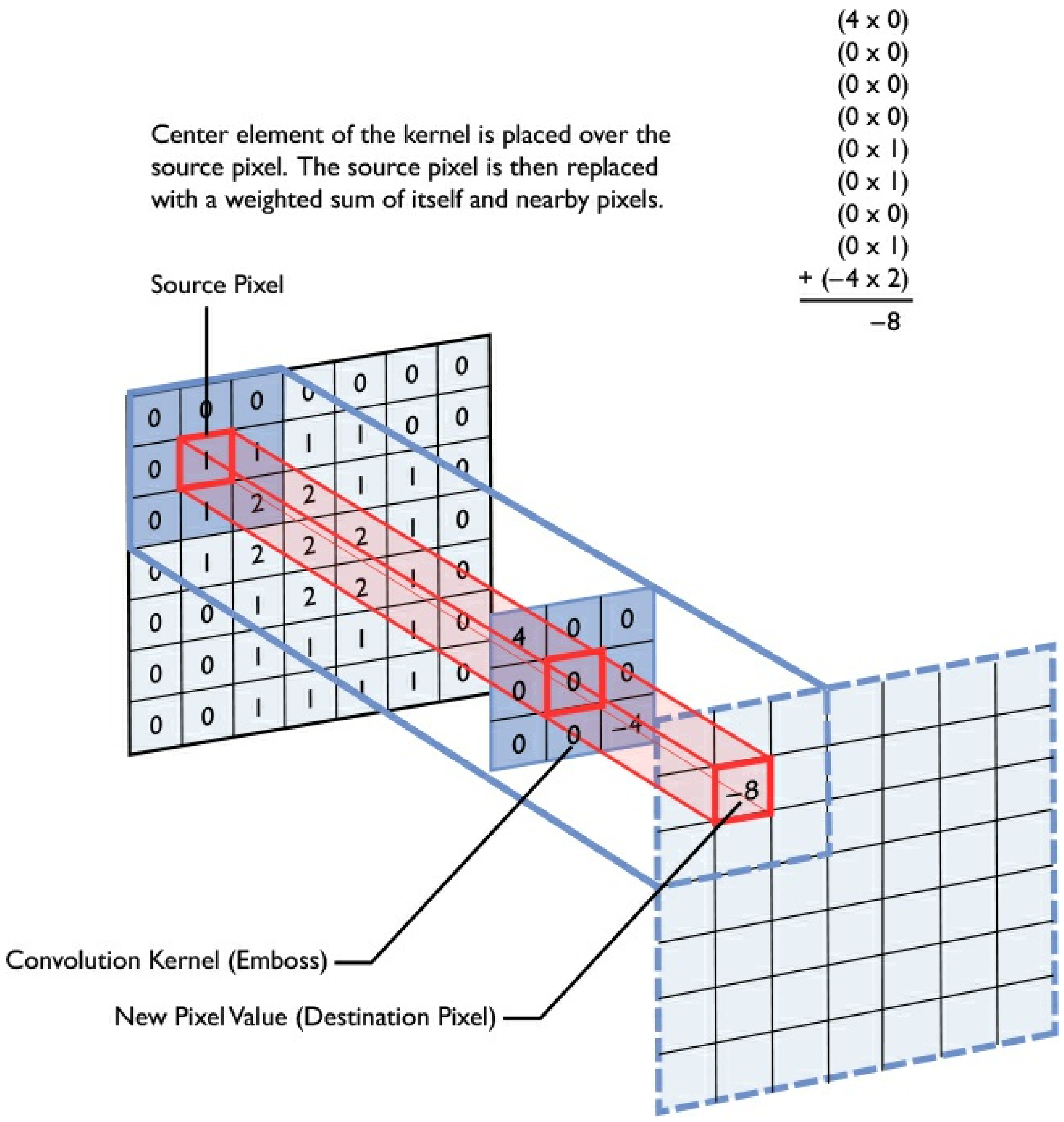
Figure 16.2. *Desaturated image*

Convolution and Dispatching a 2D Grid

The desaturation filter was interesting, but it’s constrained by being able to look at only one pixel at a time. This constraint limits the number and types of effects you can create on an image. To harness the full power of image processing, you need to look not only at the current pixel but also at the pixels around it. This process is known as *convolution*. Convolution is used in blur, sharpen, and edge detection effects. It’s a powerful concept that you must understand to fully utilize the power of Metal.

Convolution doesn’t depend on looking at just a single pixel. It depends on looking at a 2D grid of pixels. The number of pixels you need to look at to determine the value of the pixel being assessed will have row and column values, as seen in Figure 16.3.

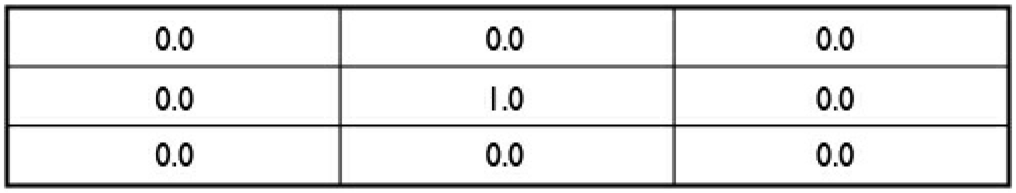
Figure 16.3. *Convolution kernel*


(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (–4 x 2)
–8
Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
Source Pixel
Convolution Kernel (Emboss)
New Pixel Value (Destination Pixel)

Convolution depends on understanding matrix operations. If you're foggy on these concepts, review the matrix section of Chapter 4, "Essential Mathematics for Graphics."

Convolution involves sampling a set of pixels around the current pixel you're processing. Even though the kernel overlaps several different pixels (or in some cases, no pixels at all), the only pixel that it ultimately changes is the source pixel underneath the center element of the kernel. The simplest convolution matrix (which is used to populate a convolution kernel) is a 3×3 matrix that looks like a tic-tac-toe game with the current pixel acting as the center square, as seen in Figure 16.4. Convolution matrices do not need to be the same dimensions, but they must have an odd number of rows and columns.

Figure 16.4. *An identity convolution kernel*



0.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	0.0

Convolution matrices, similar to other matrices, have an identity. The identity kernel serves the same purpose as the identity matrix in that if you run a value through the identity kernel, the initial value is returned unchanged. The way this works is that the value in each box of the convolution kernel is multiplied by the correlating value in a pixel in a pixel matrix. Those values are all added together to give you the final result of the operation. Say you have a matrix of pixel values that looks like this:

```
[
2.0, 5.0, 3.0,
5.0, 2.0, 1.0,
3.0, 2.5, 5.0
]
```

If you run this matrix through the identity convolution kernel, you get the following result:

```
(2.0 * 0.0) + (5.0 * 0.0) + (3.0 * 0.0) +
(5.0 * 0.0) + (2.0 * 1.0) + (1.0 * 0.0) +
(3.0 * 0.0) + (2.5 * 0.0) + (5.0 * 0.0) = 2.0
```

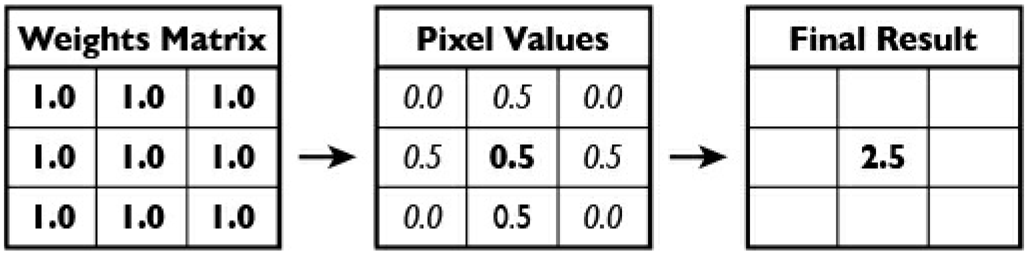
The result of this operation is known as the *weighted sum*. In this case, you are disregarding the values of all the pixels around the middle pixel value, so after running the pixel matrix through the kernel, you get the same result out as you originally put in. Keep this process in mind in the next section.

Blur Effects

The first thing to go over is what is being affected when you use convolution. In the previous section, you saw that a value was being sampled along with eight other values around it. That is slightly oversimplistic. As you saw in the desaturation example, colors are not represented by one value. They are represented by four values: percentage of red, green, blue, and the alpha value. When you are applying these weights to the pixel and the pixels around it, this operation is being run four times, once on each color channel.

Blurring an image involves mixing the color of each pixel with the colors of adjacent pixels. A *box blur* gives equal weight to all the nearby pixels, computing their average, as shown in Figure 16.5.

Figure 16.5. *A non-normalized box blur kernel*



Weights Matrix		
1.0	**1.0**	**1.0**
1.0	**1.0**	**1.0**
1.0	**1.0**	**1.0**

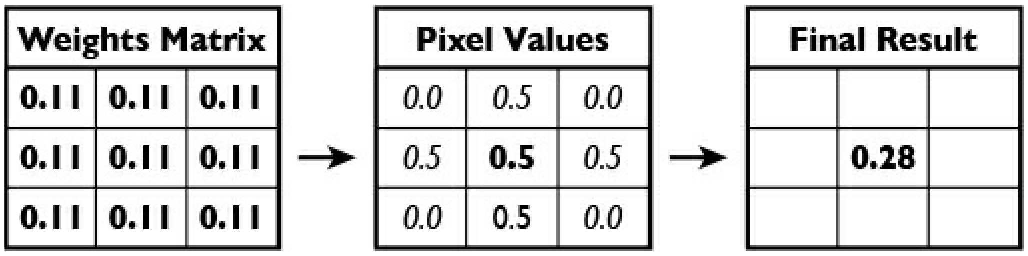
Pixel Values		
0.0	*0.5*	*0.0*
0.5	**0.5**	*0.5*
0.0	0.5	*0.0*

Final Result		
	2.5	

This effect is not quite right. The resulting pixel is five times brighter than the original pixel.

Multiplying the surrounding pixels by large whole numbers results in blowing out the image. To avoid this effect, reach back into your toolbox and use normalization. You assign the relative weights you want to each pixel, then you add those together and divide each value by that total to get the relative percentage each weight will have. Every weight added together should add up to one. This corrected kernel is shown in Figure 16.6.

Figure 16.6. *A normalized box blur kernel*



Weights Matrix		
0.11	0.11	0.11
0.11	0.11	0.11
0.11	0.11	0.11

Pixel Values		
0.0	*0.5*	*0.0*
0.5	**0.5**	*0.5*
0.0	0.5	*0.0*

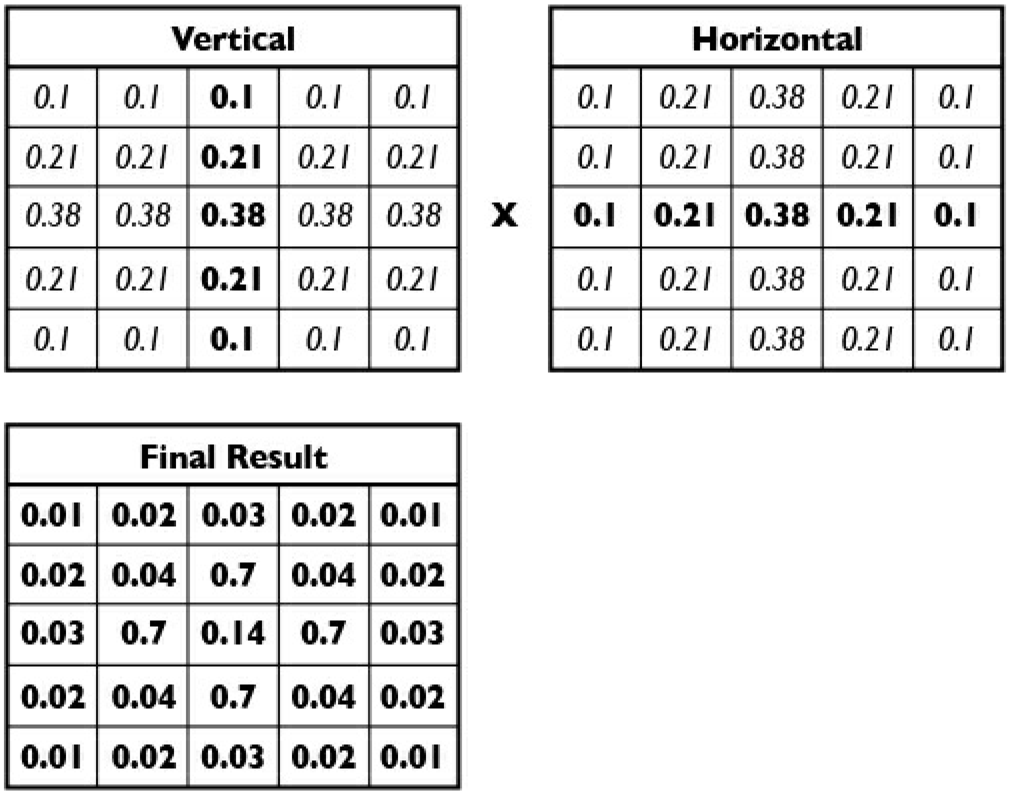
Final Result		
	0.28	

A box blur is simple, but like other simple procedures described in the book, it does not produce an optimal polished effect. A better effect can be produced using unequal weight distributions for the surrounding pixels. This type of blur is known as a *Gaussian blur*. Here is the equation behind the Gaussian blur:

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

Let’s break this down: *x* and *y* describe how far away from the sampled pixel the current coordinate is. The further away from the origin a pixel is, the less weight it has in the final calculation. The kernel is generated by determining the respective weights along the middle row and column of the kernel. This is a matrix product obtained by multiplying the weight vector with its transpose and can be seen in Figure 16.7.

Figure 16.7. *How a Gaussian convolution kernel is generated*



Vertical				
0.1	0.1	**0.1**	0.1	0.1
0.21	0.21	**0.21**	0.21	0.21
0.38	0.38	**0.38**	0.38	0.38
0.21	0.21	**0.21**	0.21	0.21
0.1	0.1	**0.1**	0.1	0.1

X

Horizontal				
0.1	0.21	0.38	0.21	0.1
0.1	0.21	0.38	0.21	0.1
0.1	**0.21**	**0.38**	**0.21**	**0.1**
0.1	0.21	0.38	0.21	0.1
0.1	0.21	0.38	0.21	0.1

Final Result				
0.01	**0.02**	**0.03**	**0.02**	**0.01**
0.02	**0.04**	**0.7**	**0.04**	**0.02**
0.03	**0.7**	**0.14**	**0.7**	**0.03**
0.02	**0.04**	**0.7**	**0.04**	**0.02**
0.01	**0.02**	**0.03**	**0.02**	**0.01**

Here is the code to implement a Gaussian blur kernel in Metal:

```
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 = weights.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);
}
```

Inside the kernel function, we iterate over the neighborhood of the current pixel, reading each pixel and its corresponding weight from the lookup 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.

Selecting an Optimal Threadgroup Size

It isn’t always the best idea to default to the deepest thread width available to you. The overhead necessary to manage your threads can offset any performance you get by having the maximum number of threads.

Many people are intimidated by multithreaded programming. When things go wrong, they go really wrong—in a way that is difficult to debug. Fortunately, one of the primary issues that causes problems in multithreaded programming is already addressed by the Metal framework. *Global mutable state* causes the dreaded race condition in which the result can change depending on which thread executes first. With Metal, the device carefully manages access to the resources shared by the CPU and the GPU to prevent the data from being modified in a way that is unexpected. That leaves choosing a good threadgroup size as your primary obstacle.

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. The number of threads per threadgroup should be less than or equal to the maximum total threads per threadgroup. For example, on a device with a maximum total threads per threadgroup of 512 and a thread execution width of 32, a suitable number of threads per threadgroup would be 32 (thread execution width) × 16 (total threads per threadgroup divided by thread execution width).

Summary

The compute command encoder allows you to do image processing with less code overhead than using a render encoder. The compute encoder also allows you to take advantage of parallelism to make the most efficient use of your GPU. You learned a few simple and common image processing algorithms to get your feet wet with image processing. It doesn't always make sense to use the maximum number of threadgroups when doing multithreaded programming due to the overhead of maintaining those threads.

17. Machine Vision

Thank you . . . motion sensor hand towel machine. You never work, so I just end up looking like I'm waving hello to a wall robot.

—Jimmy Fallon

Over the last 10 years, image processing applications, such as Instagram and Snapchat, have taken the tech world by storm. However, simple cartoon filters that give you a mustache just scratch the surface of what’s possible with image processing. Machine vision algorithms are behind many current and evolving innovations, such as self-driving cars and Amazon’s grab-and-go grocery store. This chapter covers some common machine vision algorithms and how they can change the world.

How a Computer Sees the World

Take a moment to look around. What do you see? How long does it take you to recognize the objects around you? You look at a copy of this book and you know not only that it’s a book but also that it’s a copy of *Metal Programming Guide* and not *Breeding Pugs for Fun and Profit*. You probably can’t articulate why you know this is a book because it’s simply a given that it’s a book. Our brains are incredibly sophisticated and complex. We’re able to process something visually and automatically know what it is through a complex neural network that we still don’t fully understand.

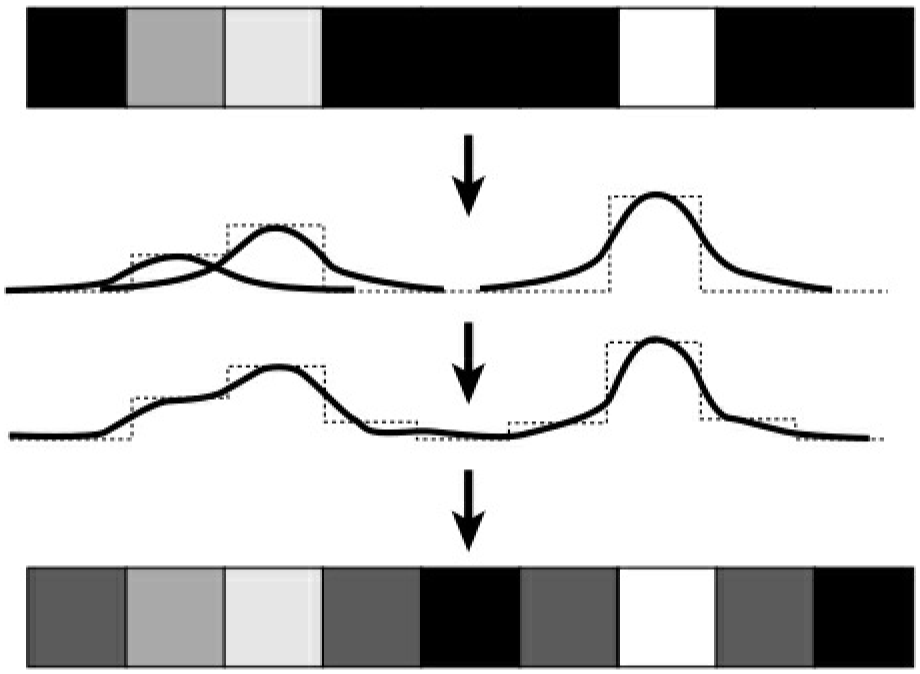
A computer sees things differently. A computer doesn’t see the different colors on a book cover or the characters that spell out the title. The computer sees a series of numerical values for the red, blue, and green intensity of an image at any given moment. The art and science of machine vision gives the computer some tools and context so that it can make sense of what those values mean.

The overall idea in machine vision is *feature detection*. Feature detection can be as simple as detecting where an edge is or as sophisticated as determining if the animal in the picture is a pug. The more sophisticated algorithms build on simpler ones, so this chapter is structured from simple machine vision concepts to more complex ones.

Noise and Smoothing

It would be great if we had perfect pictures every time we take a photo, but unfortunately, the world conspires against us. Low light, poor camera quality, and environmental factors can contribute to poor image quality. Having a noisy image can decrease the accuracy of your machine vision algorithm. If a pixel being evaluated in the convolution kernel isn’t representative of what the color should be at that given point, it can confuse the algorithm, as demonstrated in Figure 17.1.

Figure 17.1. *How smoothing works*



Chapter 16, "Image Processing in Metal," introduced you to the Gaussian blur filter. Blurs are cool effects, but they also play a vital role in machine learning. If you have a noisy image, blurs are used to smooth the image so the noise can be removed. This works because of the weighted pixel values in the convolution kernel. If you have salt-and-pepper noise, any given pixel could be randomly wrong. By smoothing the pixels together and averaging their weights, you can mitigate the impact any individual pixel has on the image.

Another algorithm to accomplish smoothing is a *median filter*. Median filters contrast with blurs because blurs utilize a mean value for the pixel by adding all the values together and dividing the values to find the mean average. Median filters choose a middle representative value of the surrounding pixels. This works better for smaller images where outlier pixels can have a disproportionate weight in a weighted kernel. The median filter would disregard those pixels as being too far outside of the median.

One final algorithm you can use to smooth images is *bilateral filtering*. Bilateral filtering has an advantage over the other smoothing algorithms in that it preserves edges. Gaussian smoothing assumes that colors will stay fairly consistent between pixels without large jumps in luminosity. That is generally true, but it does not hold up when you are dealing with edges, where their defining characteristic is large jumps in luminosity.

Bilateral filtering has a two-step process. The first step is the same Gaussian filtering mentioned earlier. The second step is also a Gaussian weighting, but it's not based on distance away from the sample pixel. The weighting in the second step is based on difference of intensity from the center pixel. Pixels that are more similar in luminosity to the sampled pixel are weighted more heavily than those with vastly different luminosity.

Sobel Edge Detection

The first machine vision algorithm you will learn about in this chapter is *Sobel edge detection*. Edge detection is a good starting point for machine vision, and Sobel edge detection is a relatively gentle introduction to feature detection.

Think critically about how you would detect an edge in an image. An edge is an area where color changes rapidly from light to dark or from dark to light. Each pixel has a set of color values. You can determine the relative luminance of a pixel by looking at those values. More critically, you can look at those values in relation to the values around them. If you have a bright pixel and every pixel to the left of it is significantly darker, that gives you an indication that this pixel is part of a feature.

Because edges can go either vertically or horizontally, it's necessary to run a Sobel operation twice. The first one checks for a horizontal edge, as shown in Figure 17.2.

Figure 17.2. *Horizontal and vertical Sobel kernels*

–1	0	+1
–2	0	+2
–1	0	+1

–1	–2	–1
0	0	0
+1	+2	+1

If the pixels have similar luminance on either side of the sample pixel, the weights cancel out one another, and you have no edge. If the pixel on either side is drastically darker or brighter, that differences is reflected in the weights. This operation is then also performed on the top and bottom rows of pixels in the kernel to check for an edge in the vertical space.

The shader creates variables to hold the intensity of the surrounding pixels in the convolution kernel. The *h* variable holds the result of the horizontal kernel operation with all the weights applied to the values. The *v* variable holds the result of the vertical kernel operation, also with the weights applied. The *h* and *v* are treated as points between which a vector length and magnitude can be calculated. This magnitude is returned as a color value.

Thresholding

Sobel edge detection is not really useful for machine vision. There is no binary yes or no as to whether a given pixel is on an edge or not. That requires a threshold.

Thresholding is fairly simple conceptually. You choose a parameter, usually luminance, to compare to a threshold value. In an 8-bit image, the threshold value can be anywhere from 0 to 255. If the current pixel is brighter than the threshold value, it is selected. If the value is lower, then the pixel is not selected. Thresholds produce binary images where each pixel is either black or white. Something is either selected or it's not. This is useful in machine vision, as you will see throughout this chapter. Creating a filter that separates pixels into one of two groups makes it easier for the GPU to deal with processing them. Either a pixel receives an effect or it does not. This is a building block that is used to produce more complex and useful effects.

There is a variation on thresholding called *adaptive thresholding*. In nonadaptive thresholding, all pixels are treated the same way and measured against the same parameter. There is no consideration about whether a pixel is part of the background or part of a large, illuminated object in the foreground. Often, you will want to treat groups of pixels differently depending on their location in relation to the pixels around them and the objects in space you represent.

Fortunately, applying adaptive thresholding utilizes a concept you're already familiar with: convolution. To create an adaptive threshold, you apply a Gaussian distribution to the surrounding pixels. This allows you increased flexibility to prevent contrast and detail from being erased from the image you are processing.

Histograms

If you've ever worked with Photoshop, you're familiar with a histogram. In Photoshop, the visual representation of an image in a histogram gives you an idea of where the darkest and the lightest parts of the image are concentrated. It allows you to enhance certain spectra of the image based on this concentration to color balance the image. This is but one use of histograms.

Histograms are a way of cataloging the composite elements of an image. An 8-bit grayscale image has 256 different values that each pixel can be. A histogram checks each pixel in the image and reports back how many pixels of each value exist in the image. These values can be bundled together in bands of values, such as 0–15, 16–30, and so on, to make it easier to analyze an image that has a lot of values. Histograms are not unique to images. Any data, such as height of girls currently in sixth grade, can be recorded and sorted into a histogram.

Histograms are useful for image processing and machine vision. You can use histograms to check for differences both between images and within images. You can check for areas of interest by finding areas that have higher histogram values. Histograms are used later in this chapter when you learn about face detection.

Facial Recognition

One major area of machine vision is facial recognition. Facial recognition can be as simple as your camera scanning the field of view for what it assumes will be the focal point of the image to being able to identify everyone present in a scene. Broadly speaking, all facial recognition algorithms fall into one of four strategies:

• **Top-down and rules-based**: The programmer creates a set of rules that a face conforms to, such as having symmetrical objects representing the eyes and nostrils. This approach is limiting because of imperfect human information. The rules are limited by the programmer's inherent understanding of what constitutes a face and what conditions the image will be created under.

• **Bottom-up and features based**: In this approach, the programmer describes regions that might constitute a face. These include lips, regions of skin color, and so on. These are extracted and used to try to reconstitute a face. This approach is susceptible to noise and shadow occlusion.

• **Template matching**: This approach creates a template of a face. The template can include a representation of a silhouette or a composite of various facial regions that are compared to the image. The weakness in this approach is that it has difficulty dealing effectively with variations in scale, poses, orientations, and shape.

• **Appearance-based modeling**: This approach is the only one that is not defined solely by humans. It combines template matching with machine learning. The computer is given a template, which it uses to search a large data set for objects that might be faces. It displays its “guesses” to the user, who rejects the objects that are not faces, thereby teaching the machine the nuances of facial recognition. This is the most effective approach and is covered in more detail in Chapter 18, “Neural Network Concepts.”

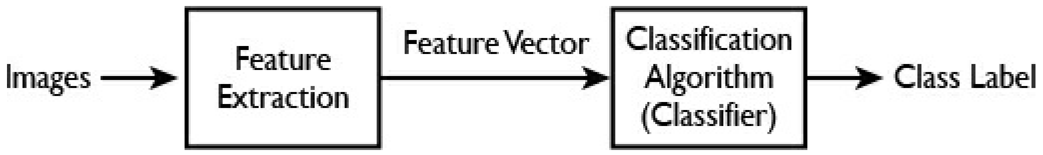
Viola-Jones Object Detection

In 2001, Paul Viola and Michael Jones proposed a new way of detecting objects that came to be known as the *Viola-Jones object detection framework*. The framework wasn’t originally targeted specifically at face detection; it was set up for general object detection. It proved to be the most effective face detection algorithm at the time and has become a foundation of the facial recognition field.

Basically, this framework utilizes feature extraction to pull out an area to analyze, as shown in Figure 17.3. It converts this area to values that can be analyzed, and it returns an answer, yes or no, about whether the feature being extracted is what you are checking it for. If you’ve seen the fourth season of *Silicon Valley*, Jian-Yang built an object-detection framework to tell you if the feature being extracted was a hot dog or not a hot dog. This is one way to utilize this algorithm, but not necessarily the best one. The algorithm has four stages:

- Haar feature selection
- Creating an integral image
- AdaBoost training
- Cascading classifiers

Figure 17.3. *How feature extraction works.*



The following sections detail how each step fits into the process. Be forewarned, these are incredibly complicated mathematical concepts. This book doesn’t go into all of this complexity but provides only a high-level overview of how this problem is approached and solved.

Haar Feature Selection

Think about a face. There are certain parts of a face that tend to be similar and to be located in similar regions of the face. Everyone has eyes (or at least a designated area of the face where they would be). Eyes tend to be about two-thirds of the way up from the chin. A person’s nose rests in the middle of the face. If you were to analyze an image to see if it’s a picture of a person, you can use this knowledge to your advantage to check areas of interest to see if they are facial features.

Haar features are rectangular areas of an image. They are shown in Figure 17.4. These rectangles have both a clear and a shaded component. The shaded component encompasses the area of interest. Subtract the sum of pixel values covered by the clear rectangles from the sum of pixel values encompassed by the shaded ones.

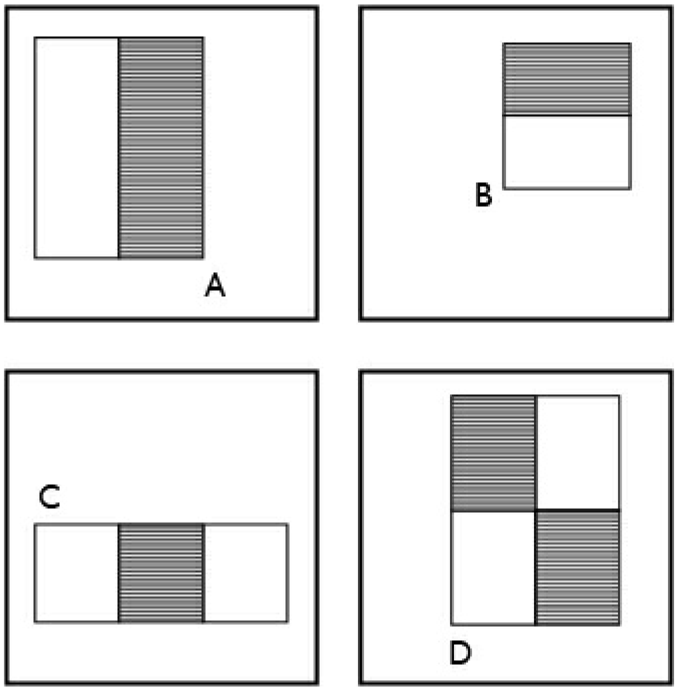
There are several different flavors of Haar features for various parts of the face:

• **Two-rectangle features**: Two rectangular regions are either horizontally (see Figure 17.4A) or vertically (see Figure 17.4B) adjacent to each other.

• **Three-rectangle features**: A central rectangular region is flanked by two other rectangular regions on either of its sides (see Figure 17.4C). This is an ideal model to check for eyes.

• **Four-rectangle features**: Four rectangular regions are arranged in a grid-like fashion, as shown in Figure 17.4D.

Figure 17.4. *Examples of Haar features*

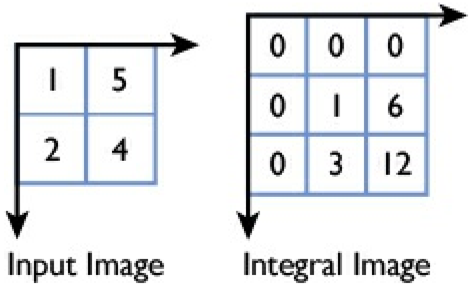


A facial detection operation uses multiple Haar features. This is all well and good to sum the values of all the shaded regions, but what happens in the common event that features overlap? It’s not efficient to sum those regions again. This case is discussed in the next section.

Creating an Integral Image

In the previous section, you learned that Haar features are made up of regions of pixels that are summed together and compared. There is no rule that says you must compute this entire region in one chunk. If the Haar feature overlaps with another Haar feature, it can make sense to break each feature up into more than one region and sum those regions together. The result is known as an *integral image* (see Figure 17.5).

Figure 17.5. *Simple integral image*



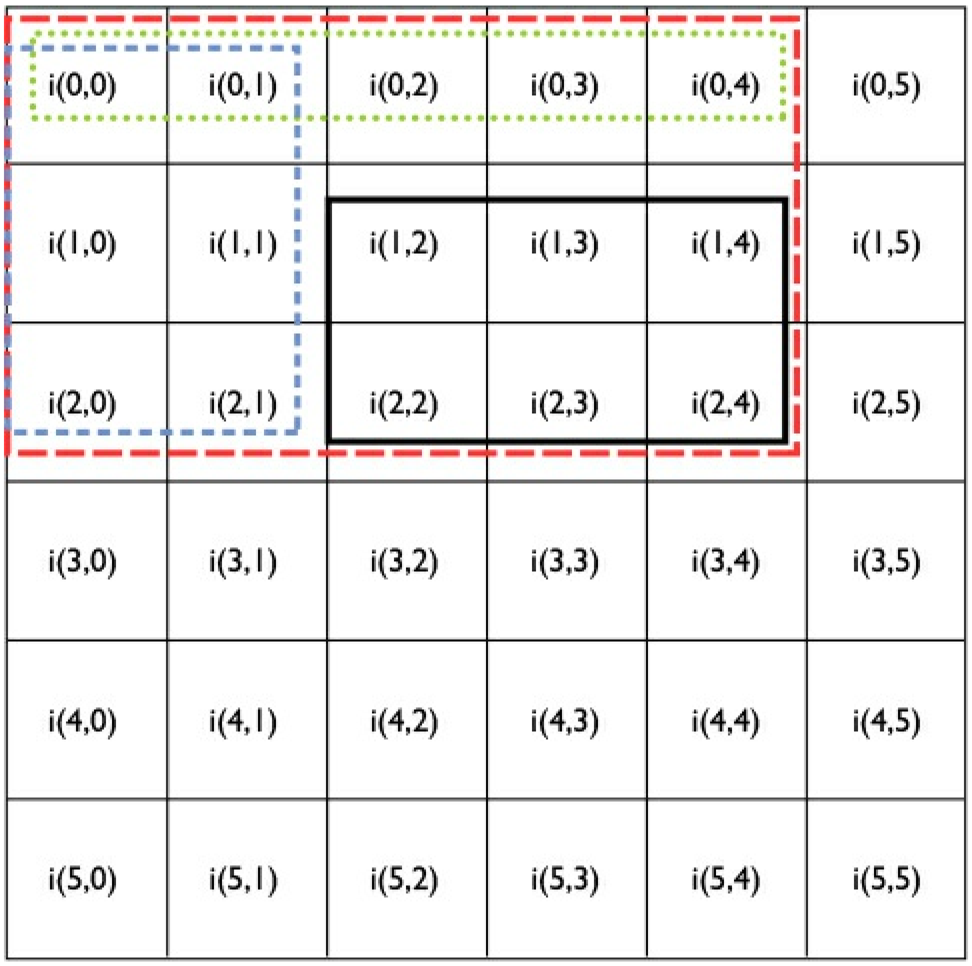
An integral image helps you rapidly calculate summations over image subregions. Every pixel in an integral image is the summation of the pixels above and to the left of it. Look at the simple integral image in the figure. There are four pixel values: 1, 5, 2, and 4. These pixels are padded by zeros to make the algorithm clearer. Because the 1 is the upper left-hand pixel, its value remains the same. It has a zero both above and to the left of it. The value next to it, 5, is the sum of itself along with all values above it and to the left of it, making its integral value 6. The 2 has a 1 above it, making its integral value 3. Finally, the 4 is in the lower right-hand corner, meaning it will be the summed total of everything above and below it.

The formula for the integral image is as follows:

$$\boxed{I(x,y) = \sum_{\substack{x' \le x \\ y' \le y}} i(x', y')}$$

Let's look at how this would work with our Haar regions. As an example, look at Figure 17.6. The region inside long-dashed rectangle encompasses the entire Haar feature that needs to be summed. The region inside the thick black rectangle overlaps with another Haar region. Since you don't want to sum that region again for the other feature, you can break it out into its own chunk and hold onto the summed value to be used later in the overlapping Haar feature. The rest of the feature is subdivided into two more rectangles to be summed. The sums of these three rectangles is summed and gives the value of the Haar feature.

Figure 17.6. *Example integral image*



The summing operation starts in the upper left-hand corner of each rectangle and ends at the lower right-hand coordinate of the feature. The last coordinate to be summed is given the compounded value of the other pixels in that region.

Let's look at the area inside the short-dashed rectangle. The rectangle starts at (0,0) and ends at (2,1). The value of this region of the feature is

```
    i(0,0) + i(0, 1) + i(1,0) + i(1, 1) + i(2,0) + i(2,1)
```

The value of this sum will be stored at (2,1). To add all the values for this Haar feature, you wind up adding only three values:

```
i(2,1) + i(0,4) + i(2,4)
```

AdaBoost Training

AdaBoost learning involves training a neural network to look at thousands of images of faces to learn what features are important and which ones are not. Each feature on its own is a *weak classifier*. A feature may or may not be indicative of a face. However, if you combine many weak classifiers, you can make a final *strong classifier*. This process informs the algorithm what to look for in an image.

Cascading Classifiers

One major issue with Haar features is that they can be calculated for any region of pixels within an image. That can result in tens of thousands of possible Haar features for any given image. No matter how efficient pulling integral images out of an image is, it’s not efficient to calculate each and every one of them. This is where *cascading classifiers* comes in.

During World War II, a group of British code breakers were responsible for cracking the German Enigma encryption machines. The basic three-rotor Enigma has 26 × 26 × 26 = 17,576 possible rotor states for each of six wheel orders, giving 6 × 17,576 = 105,456 machine states. For each of these, the plugboard (with 10 pairs of letters connected) can be in 150,738,274,937,250 possible states. The code breakers didn’t try to calculate every single one of those states because that would take far longer than the 24 hours they had until the cypher reset. Instead, they made intelligent guesses about how to eliminate various states by understanding the types of words and information that would have been encoded into the machines. They were able to cut that number down by intelligently excluding low-probability states.

For some problems in computer science, it’s easier to verify that an answer is not true than it is to solve the problem. This, to some degree, is the basis of the Millennium Prize Problem P versus NP. It’s much faster for the computer to confirm that the section it is checking does not have a face than it is to confirm that it does. Once the computer knows it can discard that region, it can save the more computationally expensive work for the sections that may possibly contain a face.

Cascading classifiers are conceptually similar to the idea of chained optionals. The cascade can abort at any point in the cascade when it knows for sure that the region does not contain a face.

The biggest limitation of Viola-Jones is that it requires the full view of a full frontal upright face, such as those seen in a mug shot. It is not effective for profile shots or images of people looking down or off to the side.

Summary

Machine vision is an important and growing part of computer science. Everything we see can be reduced to a numerical value. These numerical values can be processed and quantified for the computer. By understanding what this data conveys, we can create algorithms that describe and extract features from images.

18. Metal Performance Shaders Framework

Large increases in cost with questionable increases in performance can be tolerated only in race horses and women.

—Lord Kelvin

Machine learning and machine vision are hot topics right now. Most of the conversation around these topics involves Python and OpenCV. These frameworks get you up and running quickly because their libraries put thousands of algorithms at your fingertips, so you don't have to spend time doing work that has already been done—and done well. However, OpenCV is a clunky old C++ framework, and Python doesn't work on iOS. Wouldn't it be cool to get the advantage of those libraries in Swift? You can—through the *Metal Performance Shaders (MPS) framework.*

Overview of Metal Performance Shaders Framework

The MPS framework was introduced in 2015. Originally, it was available only on iOS, but as of 2017, it's also available on macOS. There are a few ideas behind the MPS framework. One is that most people using Metal for the compute pipeline, as opposed to rendering, will be using a lot of common operations. In Chapter 16, "Image Processing in Metal," you learned how to implement a Gaussian blur. Gaussian blurs are common steps in many image processing operations, so it makes sense to include an implementation of them in this framework so that programmers don't have to keep doing work that has already been done.

Another important aspect of Metal is that it is supported on a lot of different devices. There are multiple ways of implementing a Gaussian blur. Some work more efficiently on an A7 chip in an iPhone 5S, and others are more efficient on an A9 chip. Most programmers don't have the time to tune their Gaussian blur shader on multiple devices with different chips. Additionally, it's expensive to buy that many devices and to spend hours tuning for every single chip. Apple decided to cut out this part of the process by doing it for us. Abstracted away under the framework calls are multiple implementations of each effect. You, as the programmer, don't need to know which GPU your user is using. The framework will detect the GPU and select the most efficient implementation for that class of GPU.

Every shader in the MPS framework descends from the MPSKernel class. This class is an abstract class that is never implemented directly and is available only as a subclass. There are three general categories of MPS:

- Image processing
- Matrix multiplication operations
- Neural networks

The setup for each type of MPS follows a pattern. Each MPSKernel instance has an initializer that takes the MTLDevice instance for the program:

```
let sobel = MPSImageSobel(device: mtlDevice)
```

Each MPSKernel instance has a set of configurable properties that are unique to that shader. Check the documentation for the properties unique to the shader you are implementing. After those properties are set, you encode it to the command buffer and set the textures.

```
sobel.offset = ...
sobel.clipRect = ...
sobel.options = ...
sobel.encode(commandBuffer: commandBuffer,
             sourceTexture: inputImage,
             destinationTexture: resultImage)
```

The following sections give a detailed overview of what operations are available in each type of shader.

Image Processing with the MPS Framework

The first type of MPS this chapter covers are the ones that deal with image processing. This chapter does not focus too much on how image processing works, as that was covered in Chapter 16.

There are two abstract subclasses of MPSKernel for image processing:

• MPSUnaryImageKernel

• MPSBinaryImageKernel

The difference between these subclasses is fairly straightforward. MPSUnaryImageKernel consumes one texture and returns one texture. MPSBinaryImageKernel takes two textures and composites them to return a single texture. Neither of these classes is instantiated directly. Instead, they are subclassed depending on what effect they are trying to implement. This section goes over the capabilities of these filters.

Common Settable Properties

Regardless of the number of textures, numerous settable properties are universal to all MPS image filters:

• **Clip rect**: The clip rect describes the subrectangle of the destination texture overwritten by the filter. If the clip rectangle is larger than the destination texture, then only the area described by the texture is used. This is an optimization to avoid doing work that won’t be seen.

• **Offset**: There are two different types of offset depending on whether you are using a unary or binary image kernel. Because unary image kernels have one texture, they have only one offset. Binary image kernels have a primary offset for their primary image and a secondary offset for the secondary image. The offset represents the position of the top left corner of the clip rectangle in the source coordinate frame.

• **Edge mode**: As with offset, there are different types of edge modes. Unary images have only one edge mode, and binary images have a primary and secondary edge mode. An edge mode describes the behavior of texture reads that stray off the edge of the source image. These were detailed in Chapter 16.

Some image filters have the option to process in place. Using this option, you don’t create a new texture from the original texture; instead, you modify the original texture. The advantages of this technique are that you save memory by having one texture and you save time by not having to copy the texture over in memory. Not every kernel can process in place, and it's not always available on every GPU, so if you choose to use this option, you need to check whether it's supported before using it. To simplify error handling with failed in-place operation, the encode(commandBuffer:inPlaceTexture:fallbackCopyAllocator:) method takes an optional MPSCopyAllocator object.

In the following sections, you learn about the various categories of image processing filters and some specific examples of each.

Morphological Image Filters

Morphological operators cover a broad class of image processing algorithms. These algorithms are a branch of image processing that has been successfully used to provide tools for representing, describing, and analyzing shapes in images. They're useful for separating foreground and background features. These algorithms assume the use of a binary image and use set theory to extract image similarities and differences.

A few operations limit the image area for each source in the maximum and minimum directions:

- MPSImageAreaMax

- MPSImageAreaMin

MPSImageAreaMax is a filter that finds the maximum pixel value in a rectangular region centered around each pixel in the source image. This operator is good for extracting foreground features in a binary image. Conversely, MPSImageAreaMin is a filter that finds the minimum pixel value in a rectangular region centered around each pixel in the source image. This operator is good at separating out the background of an image.

MPSImageAreaMax and MPSImageAreaMin are fairly crude and simplistic. *Dilation* and *erosion* are more sophisticated morphological operations that exist as morphological performance shaders:

- MPSImageDilate

- MPSImageErode

An MPSImageDilate filter behaves like the MPSImageAreaMax filter, except that the intensity at each position is calculated relative to a different value before determining which is the maximum pixel value, allowing for shaped, nonrectangular morphological probes. The same goes for MPSImageErode. You can create values for these filters that emulate MPSImageAreaMax and MPSImageAreaMin, but they give you more options to customize your morphological operations.

Convolution Image Filters

Convolution is an important aspect of both machine learning and computer vision. The MPS framework provides a large variety of convolution-based image filters. The first set of convolution filters is a general convolution filter called MPSImageConvolution. Like all convolution filters, it requires an odd number for the width and the height. The number can be either 3, 5, 7, or 9. You pass in the width, height, and a pointer to an array of pixel weights to set the weights for your convolution kernel.

Metal also has built-in implementations of many common convolutional filters. One large subset of convolution involves various kinds of blurs:

• **MPSImageBox**: This is the simplest blur filter. Each pixel has the same weight in the convolution kernel.

• **MPSImageTent**: This filter's weights are strongest in the middle column and less strong as you move away from the middle, so the weights are tent-shaped.

• **MPSImageGaussianBlur**: This is the MPS implementation of the Gaussian blur filter you were introduced to in Chapter 16.

• **MPSImageGaussianPyramid**: This variation of a Gaussian blur uses mipmap levels for a more refined blur effect.

If you want to get the advantage of denoising without losing clear edges, you can use the MPSImageMedian filter.

The other big area of convolution is edge detection. One important common filter offered in the MPS framework is MPSImageSobel. Another filter for edge detection included in the framework is MPSImageLaplacian.

Histogram Image Filters

A histogram is a graphical display in which the data is grouped into ranges. In image processing, histograms represent the relative abundance of every color and luminosity in an image. The MPS class for creating a histogram is MPSImageHistogram. Here is a code snippet for creating a MPSImageHistogram:

```
var histogramInfo = MPSImageHistogramInfo(
    numberOfHistogramEntries: 256,
    histogramForAlpha: false,
    minPixelValue: vector_float4(0,0,0,0),
    maxPixelValue: vector_float4(1,1,1,1))

let calculation = MPSImageHistogram(device: device,
                                    histogramInfo: &histogramInfo)

let bufferLength = calculation.histogramSize(
    forSourceFormat: sourceTexture.pixelFormat)
let histogramInfoBuffer = device.makeBuffer(
    length: bufferLength,
    options: [.storageModePrivate])

calculation.encode(to: commandBuffer,
                   sourceTexture: sourceTexture,
                   histogram: histogramInfoBuffer,
                   histogramOffset: 0)
```

This calculation is then fed to one of two MPS histogram implementations:

• MPSImageHistogramEqualization

• MPSImageHistogramSpecification

MPSImageHistogramEqualization is a filter that equalizes the histogram of an image. This creates a privately held image transform that is used to equalize the distribution of the histogram of the source image. After the equalization object is created, it needs to encode this transform to the command buffer:

```
func encodeTransform(to commandBuffer: MTLCommandBuffer,
                     sourceTexture source: MTLTexture,
                     histogram: MTLBuffer,
                     histogramOffset: Int)
```

This method encodes a source texture and histogram with offset to a specific command buffer. The last step is to call the encode(commandBuffer:sourceTexture:destinationTexture:) method to read data from the source texture.

MPSImageHistogramSpecification is a filter that performs a histogram specification operation on an image. This has a similar implementation process, but the encodeTransform method takes more parameters:

```
func encodeTransform(to commandBuffer: MTLCommandBuffer,
                     sourceTexture source: MTLTexture,
                     sourceHistogram: MTLBuffer,
                     sourceHistogramOffset: Int,
                     desiredHistogram: MTLBuffer,
                     desiredHistogramOffset: Int)
```

The biggest difference between the two implementations is that MPSImageHistogramEqualization is a unary image operation and MPSImageHistogramSpecification is binary image operation.

Image Threshold Filters

Thresholding is an important part of edge detection and machine vision. The MPS framework offers five different flavors of thresholding:

• MPSImageThresholdBinary

• MPSImageThresholdBinaryInverse

• MPSImageThresholdToZero

• MPSImageThresholdToZeroInverse

• MPSImageThresholdTruncate

MPSImageThresholdBinary is a filter that returns a specified value for each pixel with a value greater than a specified threshold or 0 otherwise. This is the baseline thresholding filter, with no frills, that returns a binary image. If you want a filter that is the inverse and returns 0 once you cross the threshold and a different value under that threshold, you need to use MPSImageThresholdBinaryInverse.

In some situations, you'll want to hold onto the source pixel's luminosity rather than return a predetermined value. In those cases, you would use MPSImageThresholdToZero. This filter returns the original value for each pixel with a value greater than a specified threshold or 0 otherwise. The inverse of shader is MPSImageThresholdToZeroInverse.

The last thresholding shader in the framework is MPSImageThresholdTruncate. This filter clamps the return value to an upper specified value. This prevents the source pixel from going over a certain value by checking to see if the source pixel value is greater than the threshold. This shader is useful in video production. If you have a value that maxes out a color channel, it causes some issues, so you want to make sure the channel never maxes out.

Image Integral Filters

Integral images are an important part of machine vision and are explained in more detail in Chapter 17, “Machine Vision.” This section covers the MPS implementation of creating integral images.

The first implementation of an integral image in the MPS framework is MPSImageIntegral. This filter calculates the sum of pixels over a specified region in an image. This shader doesn’t have any special properties that need to be set beyond the normal MPSKernel superclass. The most important things you need to set are the offset and the origin to conform to the selection that you need to sample.

The other implementation is MPSImageIntegralOfSquares. This is a filter that calculates the sum of squared pixels over a specified region in an image. It is used in feature detection. The setup for this implementation is the same as for MPSImageIntegral.

Image Manipulation Filters

One common operation in image processing is transforming the image’s size, color, or aspect ratio. The MPS framework provides five shaders to make this task easier. The first shader in this section is the MPSImageConversion shader. This filter performs a conversion of color space, alpha, or pixel format. Sometimes, an asset needs to be reformatted to work with the other assets in your project, and this filter does that for you. There is a large list of supported pixel formats in the Metal documentation.

Scaling images is a big part of image processing, and the MPS framework provides three different shaders for scaling. MPSImageScale is a new shader in iOS 11. This filter resizes and changes the aspect ratio of an image. It has one instance property, which is an unsafe pointer to an MPSScaleTransform. This is a struct that holds all of the properties necessary to describe a scale transform, which is the scale factor and the translation factor.

The next scaling shader is MPSImageLanczosScale. This scaling filter utilizes Lanczos resampling. It also uses the same properties as the MPSImageScale filter but does slightly different calculations under the hood. The same is true of MPSImageBilinearScale, which is another new shader in iOS 11.

The last image manipulation filter is MPSImageTranspose. An MPSImageTranspose filter applies a matrix transposition to the source image by exchanging its rows with its columns. Matrix operations can be in either row-major order or column-major order. Ensuring that you are clear on which way this is set up is important, because you will get radically different results if you switch the order around.

Image Statistics Filters

Image statistics filters are used in smoothing algorithms to average out extreme variances in pixel luminosity. iOS 11 has three new shaders for this task:

• MPSImageStatisticsMean

• MPSImageStatisticsMeanAndVariance

• MPSImageStatisticsMinAndMax

The most basic one is the MPSImageStatisticsMean filter. This filter computes the mean for a given region of an image and applies it to each pixel in the kernel. This prevents outliers from throwing off the calculations. MPSImageStatisticsMeanAndVariance computes the mean and variance for a given region of an image. The mean is written to pixel location (0, 0), and the variance is written to pixel location (1, 0). The last filter is MPSImageStatisticsMinAndMax. This filter finds the minimum and maximum values within a region. The minimum value is written to pixel location (0, 0), and the maximum value is written to pixel location (1, 0).

Image Arithmetic Filters

If you’ve ever worked with Photoshop, you know that its layers have compositing modes that allow a layer on top to be blended with a layer below it. With this functionality, users can create many interesting effects. New to iOS 11, these arithmetic blend modes are now available in MPS.

Each arithmetic filter is subclassed from the MPSImageArithmetic filter. This base class has properties for things such as scale, stride, and bias. This class has four subclasses, one for each arithmetic operation:

• MPSImageAdd

• MPSImageSubtract

• MPSImageMultiply

• MPSImageDivide

Keypoints

The last section in the image processing category of the MPS framework is *keypoints*. Keypoints are areas of interest within an image that are used as landmarks when applying transforms to images. If you rotate or scale an image, you can check these keypoints to ensure the image translated properly.

The first shader you need to know about is MPSImageKeypointData. This is a data structure that contains information about a keypoint. Keypoints are found using MPSImageKeypointRangeInfo. This structure takes two parameters: the maximum number of keypoints and the minimum threshold value a keypoint’s color must have. These two structures are used by MPSImageFindKeypoints to find and store a specified number of keypoints in an image.

Matrix Operations with MPS

One major advantage of programming on the GPU is having access to fast matrix linear algebra operations. You saw examples earlier in this chapter of more complex effects, such as Sobel edge detection. Those effects are built on these matrix operations. Even if your favorite effect isn't available in the MPS framework yet, you still can take advantage of the framework. A lot of these operation are new this year, so the variety of shaders is growing each year.

Matrices and Vectors

The base data structure all of the matrices and vectors are built on is the MPSMatrix object. It is a 2D array of data that stores the data's values. Matrix data is assumed to be stored in row-major order. Besides just rows and columns, MPSMatrix has properties for the data type and the stride, in bytes, between corresponding elements of consecutive rows in the matrix and the MTLBuffer that stores the matrix data.

To create a MPSMatrix, you need to use an MPSMatrixDescriptor, which initializes the MPSMatrix. You can initialize it as one matrix with a set of rows, columns, and the data type, or you can create a matrix of matrices. In the latter case, you need to specify the number of matrices and the stride for each row in those matrices.

If you require only one dimension of data instead of two, you have the option to use an MPSVector object. You need to specify the length of the vector when it is initialized. This is similar to MPSMatrix in that you need a descriptor to create an MPSVector object. The MPSVectorDescriptor can create either a single vector or an array of vectors.

Multiplying Matrix Operations

One main matrix operation is to multiply matrix objects. This is represented by the MPSMatrixMultiplication shader. This shader computes the following equation:

```
C = alpha * op(A) * op(B) + beta * C
```

A, B, and C are matrices represented by MPSMatrix objects, and alpha and beta are scalar values of the same data type as the values of C. MPSMatrixMultiplication uses the following initializer:

```
init(device: MTLDevice,
     transposeLeft: Bool,
     transposeRight: Bool,
     resultRows: Int,
     resultColumns: Int,
     interiorColumns: Int,
     alpha: Double,
     beta: Double)
```

You have the option of transposing either the left or the right matrices. That option is passed into the initializer. You also specify the number of rows and columns in the result matrix. The last properties you are passing in are the scale factors for the product and the initial value of C. The multiplication kernel is encoded using this encoding method:

```
func encode(commandBuffer: MTLCommandBuffer,
            leftMatrix: MPSMatrix,
            rightMatrix: MPSMatrix,
            resultMatrix: MPSMatrix)
```

The left and right matrices are the input matrices. The result matrix is what you get when those two are blended together using that equation.

Another common operation is to translate an image to a matrix of image data. The operation to do this is called MPSImageCopyToMatrix. This operation stores each image as a row in the matrix. If you want to store multiple images, they will each have their own row. The number of elements in a row in the matrix must be greater than the image width multiplied by its height multiplied by the number of featureChannels in the image.

Summary

The MPS framework offers a great deal of power without a lot of setup. It includes common matrix operations that are optimized for performance. It also has a large library of common image processing operations so that you don’t have to write new code every time you want to use, say, an edge detection shader. The MPS framework’s large library makes creating and implementing neural networks easier and more powerful.

19. Neural Network Concepts

Thou shalt not make a machine in the likeness of a human mind.

—Orange Catholic Bible

One of the biggest philosophical questions we have is what makes us human. The holy grail of artificial intelligence is to create a computer system that mimics the human mind to the point that it is indistinguishable from the real thing. To begin to attempt this task, we need to understand how our brains learn and function and find a way to mimic this process in software. This is a large, immersive topic, but this chapter introduces how to approach this problem.

Overview of Neural Networks

What does it mean to think? Your brain is a sensory input device that is constantly processing how to react to the world around it. You hear a thud coming from your basement. Is it a prowler? Is it your cat knocking a box over? Is the box something important, like your grandmother's heirloom china set? You initially panic because you imagine all the worst-case scenarios, but as you process the sound, you discount the most alarming possibilities and decide it was nothing to be alarmed about. All of these thoughts and processes begin and are completed in a second.

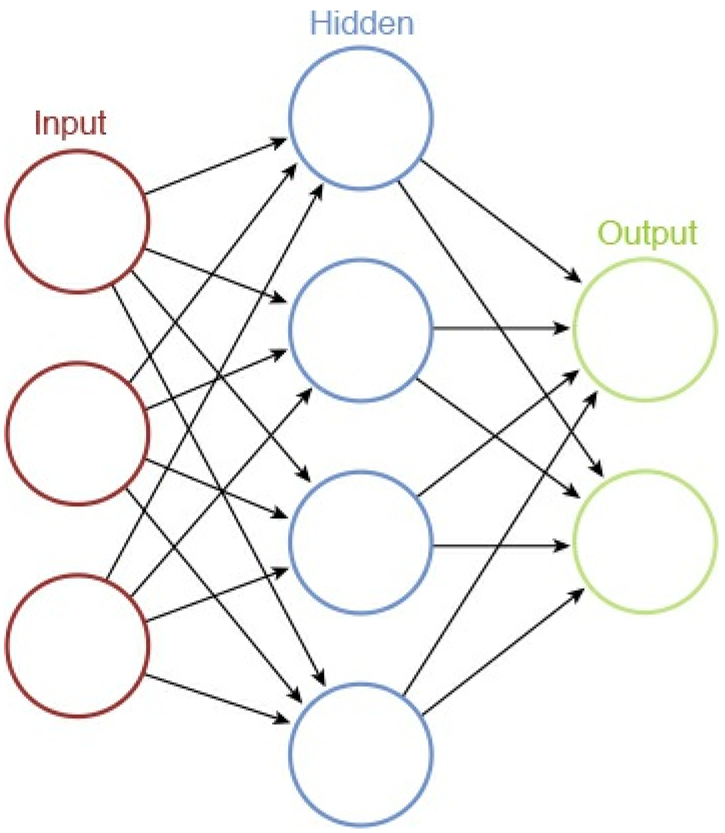
Neural network programming mimics this process. It takes a series of inputs with weighted values, evaluates a binary question, and returns either a positive or a negative. Take a common question that many computer programmers face: Should I relocate to San Francisco? There are a lot different points to consider: the cost of living is higher, it's offset by an increased salary. San Francisco has a vibrant developer community. The weather and climate are temperate, but you have to worry about earthquakes. There is a robust public transportation system. However, you may enjoy driving to work, and the commute time might be a deal breaker.

Problems such as this one can be modeled by a neural network. Each considerations is an input. Because everyone is different and has a different priority, each consideration would have a different weight. For some people, being in the middle of technical innovation is more important than any other consideration. For others, this perk isn't important, and the prospect of an hour-long commute is untenable. Considering the split between people who absolutely will not relocate to San Francisco and the large number of people who are still trying hard to get there, clearly there is no "right" answer to the question. Because different people have different priorities, people come to their own personal conclusions.

Neural Network Components

Neural networks are made up of a graph of nodes that constitute layers, as shown in Figure 19.1. These node layers are loosely categorized as one of three types: input, output, and hidden. The input layer is a collection of various inputs and weights. In the relocating to San Francisco example, each consideration would be a node in the input layer. Depending on how important each consideration is to you, each node is given a different weight. The output layer consists of a single node that represents the output, yes or no.

Figure 19.1. *General representation of a neural network*

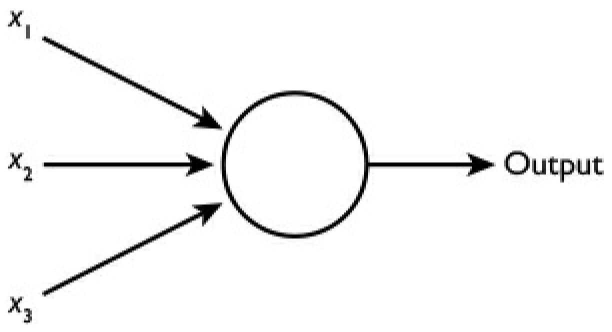


The hidden layer sounds super mysterious, but it’s simply any layer of nodes that are not direct inputs or outputs. The hidden layer is where all the neural network goodness lives. This is where algorithms are incorporated into the data inputs and where that data gets refined. Think of it like a data factory. An input is passed to the first stage of processing. That data is passed along to the next, and so on and so forth, until the output comes out the end. This type of network is called a *feedforward neural network*. A neural network that loops back is called a *recurrent neural network*. Recurrent neural networks are now available in the Metal 2 framework, but they are beyond the scope of this book. We focus on feedforward neural networks for now. Different types of hidden layers are described in greater detail later in the chapter.

Perceptrons

There are several different types of artificial neurons. One of the earliest and most primitive artificial neurons is a *perceptron*. Perceptrons have been around since the 1950s. A perceptron takes a series of inputs and weights and outputs a binary response, either 1 or 0, as shown in Figure 19.2. The neuron’s output is determined by whether the weighted sum is less than or greater than some threshold value.

Figure 19.2. *Representation of a perceptron*



Each input for the perceptron is also a binary value, either 1 or 0. Let's look at the relocating to San Francisco example to explain how perceptrons work. Let's say you have the following inputs into the perceptron:

• Is my commute less than an hour?

• Is my salary more than $160,000 a year?

• Will I have a social network of other developers to spend time with?

• Do I have to worry about a pet?

For the purposes of this example, pretend that 1s are positive indications that you want to relocate to San Francisco. In this example, your commute is an hour and a half, so the first input is 0. However, your job offer is $160,000 a year, so the second input is a 1. You already know that you will have a great social network of developers, so the third input is a 1. You don't have a pet in this example, so because you don't have to worry about trying to find a place that takes pets, the last input is also a 1.

If you were simply doing a pros and cons list, you would clearly choose to relocate because you have three pros and only one con. But that's not how this works. Pretend that you really, *really* hate commuting. For you, having a commute that is over an hour is a total deal breaker. If you give that neuron a weight of 10 and every other input a weight of one, then the result of that input will completely blow out every other consideration in the network. The sum of the inputs would be:

```
(10 * 0) + (1 * 1) + (1 * 1) + (1 * 1) = 3
```

Right now, this result has no meaning because you haven't set up a threshold yet. Let's set the threshold to 5. The sum of the inputs must be greater than 5. If the sum is exactly 5, it will return false. The threshold is set low enough so that if your commute is more than an hour, you are never going to exceed the threshold. It's possible this is what you want, but if it isn't, then the weights and the thresholds can be adjusted accordingly.

Biases

There is another way to weigh whether a set of inputs results in a 0 or a 1 besides just a threshold. This method is known as a *bias*. A bias is a measure of how easy it is to get a perceptron to fire. As in life, the larger the bias, the more likely it is to get the desired outcome—in this case, to fire and return a 1 instead of a 0.

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

This equation is slightly different than the one for thresholding. Rather than setting a threshold, the threshold is always 0. Weights can be given negative and positive weights. As before, each perceptron value is multiplied by the weight and summed, but in addition to that, a bias value is added into the equation.

Let’s go back to our relocating to San Francisco example. Even though there are a lot of good reasons not to relocate to San Francisco, you really in your soul want to relocate. So, you can adjust your weights and the bias to reflect that reality. You could theoretically just change the weights, but that results in some jarring and extreme changes in the results you get. You’ll see why this distinction is important in the next section on sigmoid neurons.

Let’s reevaluate our San Francisco example. Our new weights could look like this:

- **Commute**: −5
- **Salary**: −1
- **Social network**: −1
- **Pet**: −1

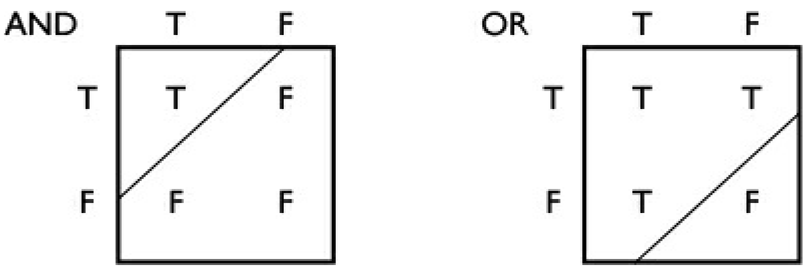
Instead of assigning a 1 to an input that will cause the neuron to fire, you will now assign a 0. With our new equation, we have this:

```
(-5 * 1) + (-1 * 0) + (-1 * 0) + (-1 * 0) = -5
```

This still doesn’t cause the neuron to fire, but if you create a bias of 6 and add it to the −5, you have a positive value, and it causes the neuron to fire. This creates a less extreme network. It can still fail if multiple neuron values change, but it does give some more flexibility to change the weights and values without the result swinging too rapidly.

So far, this is cool, but all you’ve done is create a giant decision tree structure. This already exists in computer science. You are hand coding your weights and biases. What happens if you want to input a data set of images with various values? What if you want to utilize the GPU to actually learn from a data set? The decision tree structure won’t work for you, so you need a new type of artificial neuron, as shown in Figure 19.3.

Figure 19.3. *A data set that can be linearly separated*

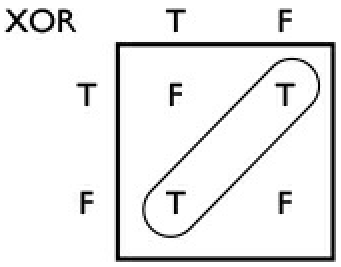


Sigmoid Neurons

Perceptrons are inherently limiting. They work well when you have linear data, but not when you have nonlinear data. Pretend you have two friends who don't get along with one another. You are deciding if you're going to go to a party. You and both of your friends are invited, but every time they're at a party together, they have a massive fight. You're happy to go to the party if only one of them is attending, but not if both are attending. If they're both going, you don't want to go because you don't want to get in the middle of their drama. If neither of them is going, you don't really know anyone else there, and you wouldn't want to go and hide out in the kitchen making friends with the local pets.

This dilemma is known as an *exclusive or* or *XOR* problem. There is no way to linearly separate these conditions in a truth table using a perceptron, as shown in Figure 19.4. This is where *sigmoid neurons* come in.

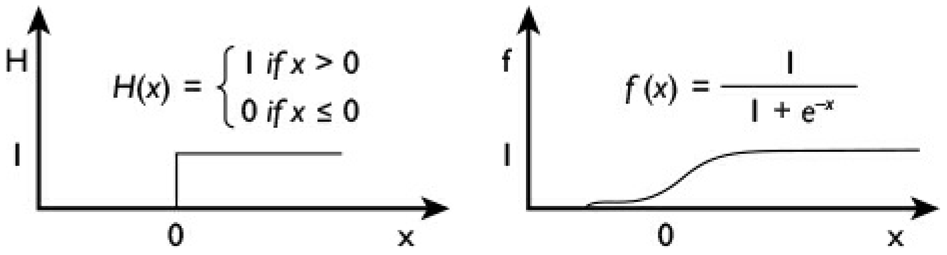
Figure 19.4. *A data set that cannot be linearly separated*



Sigmoid neurons are nonlinear. Linear simply means that an amount increases at a steady rate. If you remember back to high school algebra, when you graphed a $y = x$ equation, the slope was 1. The line always went up one unit and over one unit forever. This concept is simple, but it is not indicative of natural phenomena. The equation for a sigmoid neuron looks like this, and is shown in Figure 19.5.

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}.$$

Figure 19.5. *A comparison between a linear and a sigmoid analysis*



In many situations, you need a nuanced read of data. The perceptron example about relocating to San Francisco is a little crude. In a situation in which the offered salary is $159,000 instead of $160,000, your perceptron will return 0, even though you probably don't really care about the missing thousand dollars. The total is close enough. There is also no wiggle room in regard to the length of the commute. A commute that is an hour and five minutes has the same result and weight as one that is three hours. It's possible that even though you want less than an hour for the commute, you're willing to settle for one that is only a few minutes longer. With perceptrons, you can't produce a gradient to represent results that are close enough to or are significantly outside the range of what you want.

Learning and Training

The biggest promise of neural networks is to create a system that is capable of learning. Subtly changing weights and biases can have large impacts on the results. In practical implementation, you are training a data model to understand that a picture of a pug is a pug and not a hot dog.

In the San Francisco example, you worked under an assumption that you are personally entering in weights and biases. A theoretical, practical model capable of learning would be able to scan a large database of current job openings in Silicon Valley along with prospective living spaces and look for combinations of jobs and living spaces that suit your personal needs. To do that, the model needs to understand what a good combination looks like and what a bad combination looks like.

There are three general ways to train a neural network:

• *Supervised learning* is the learning style you probably are most familiar with and use most often, for now. Supervised learning entails taking a labeled data set of images and teaching the network to recognize the object based on multiple images of the same object. After training the model, you introduce it to images of the objects you trained it on. Based on your input as to whether or not the guess is correct, the model self adjusts.

• *Unsupervised learning* entails inferring information from a large data set. This is an existential learning style because there are no right or wrong answers. The network simply analyzes a lot of data to discover similarities. This style of learning would work for political parties looking to target the people most likely to vote for them. The network can take a large set of demographic data and look for seemingly unrelated data.

• *Reinforcement learning* is about receiving feedback based on decisions. Pretend you're an admissions person at a college. You have a limited number of people you can admit, but you have no way of knowing which students are honestly committed to attending your college and how many consider it their safety school. You want to optimize the number of students who choose to attend your college after being accepted. You can compile a list of students who were accepted to find common elements that make them most likely to attend your college. This will help you maximize the best students you can admit who are most likely to attend.

One thing that neural networks are really good at is pattern recognition. In Chapter 17, "Machine Vision," you learned a bit about face and detection. That is an example of pattern matching. A data model looking at a thousand faces can begin to recognize that they all have similar features, including two eyes and a nose. It can recognize that there is a narrow range of skin tones and that if an object is overwhelmingly green, it is unlikely to be a face.

Training Sets and Back Propagation

So far, the neural network has absolutely no idea what constitutes a correct or an incorrect answer. The neural network can't learn or adapt without feedback about whether a guess is right or wrong. In supervised learning models, you utilize a training set to teach the network how to recognize features. This training set produces a data model that is imported to a neural network.

Can I Train a Learning Set on an iPhone?

One common question about Metal is whether you can train a data model on the iPhone. The short answer is no. The more involved answer is detailing what a training set entails.

Training a data model entails a large collection of training images. One of the most popular and familiar training sets is the ImageNet training set. Besides having intellectual property permissions issues, the ImageNet training set is 190 GB. This is entirely too large for the vast majority of iOS devices on the market.

Beyond that, the GPU on the iPhone, and even on most Mac devices, is too underpowered to train a data model in a reasonable period of time. One of the most powerful GPUs on the market is the NVidia GTX 1080. This is a moving target because GPU technology is evolving very rapidly.

Apple technology is a lot of things, but customizable isn't one of them. The hardware is targeted at a general market of people who don't really want or need a $600 external GPU. This is starting to change with the introduction of the external graphics development kit in Metal 2, but generally, you're going to get more bang for your buck if you build a computer specifically to train data models rather than jury-rig your MacBook Pro to train a data model.

The training process works as follows. The network is presented with a labeled image and is asked to analyze it and return a guess as to what the image represents. There are two options. The network could answer correctly or not. If the network does not answer correctly, the result is an error. That error is fed back into the network using a method called *back propagation*. The network internally adjusts its weights and biases and tries to minimize the number of bad guesses it has within this learning set.

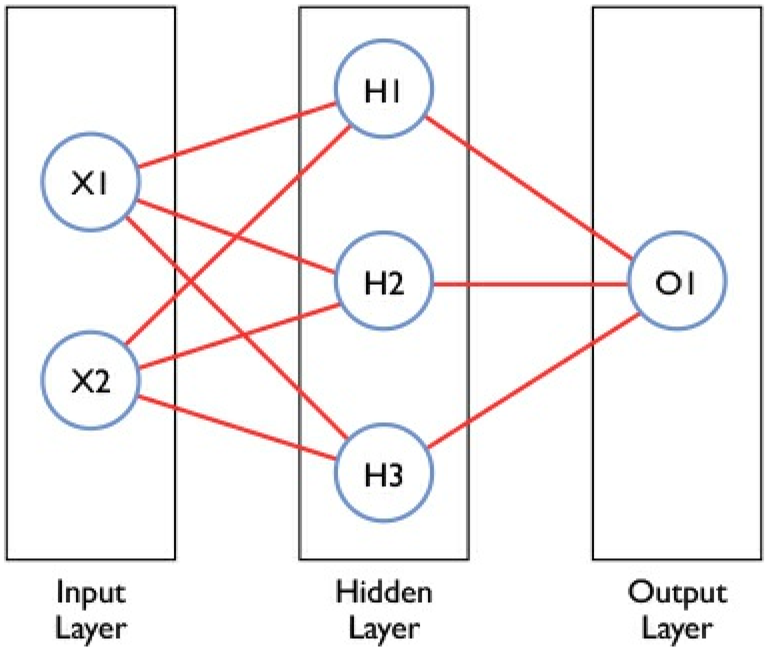
Think of back propagation like taking sample tests for the SATs. You can take practice tests over and over again, getting feedback about what questions you got wrong and what the correct answer was. After a few of these tests, you notice certain patterns emerge in regard to the common traps laid by the people who write the test. After doing enough of these practice tests, your success rate increases quite a bit, but this only works if you get feedback about the questions you got wrong and what the correct answers were.

Neural Network Architecture

The first section of this chapter went over the various components of a neural network. A neural network is more than just these components. It is how they are constructed to create a network. This section discusses how to put all of these components together to create a neural network.

Figure 19.6 illustrates a simple three-layer neural network composed of sigmoid neurons in the hidden layer. There are two inputs, three hidden nodes, and one output. One thing to notice here is that both of the input nodes are connected to all three of the hidden nodes, and all three hidden nodes are connected to the output. This is an example of a *fully connected neural network*.

Figure 19.6. *A simple three-layer neural network with two inputs and one output*



This neural network is a little like a machine. The output of the output node changes on the basis of the inputs fed into it. No values in here are stored. They are ephemeral, and they change according to the inputs.

Earlier in this chapter, you learned about weights and biases. In the crude example, you set them by hand. You also learned about trained data models. Those come into play here. To determine whether or not the output neuron fires, the inputs are multiplied by the weights and biases in the hidden layer. Those weights and biases are determined by the trained data model. This is one reason there is a joke about data scientists being unable to explain how a result occurred. The training of the neural network crunches this data in such a way that you don't necessarily see what it's doing.

Here is what the code could look like for this neural network:

```
// pseudocode
func sigmoid(x) {
  return 1 / (1 + exp(-x))
}

func f(X1, X2) {
    H1 = sigmoid( X1*Wxh[1,1] + X2*Wxh[2,1] + bh1 )
    H2 = sigmoid( X1*Wxh[1,2] + X2*Wxh[2,2] + bh2 )
    H3 = sigmoid( X1*Wxh[1,3] + X2*Wxh[2,3] + bh3 )

    O1 = sigmoid( H1*Who[1] + H2*Who[2] + H3*Who[3] +   bo1 )
    return O1
}
```

The function takes in the two node inputs and uses them to calculate the values of the hidden layers. The hidden layers are calculated using the sigmoid function at the top. This function divides 1 plus the exponent of the negative input by1, which creates the familiar S-shaped curve where the value increases rapidly at the beginning and end of the function.

Wxh[1,1] represents the weight between the first input layer and the first hidden layer. The [n, n] value represents the correlating input node and hidden node. There are different weights between the first, second, and third hidden nodes, so you need to specify which relationship you're adding to each equation. bh1 represents the bias for the first hidden node. Similarly, bh2 is the bias for the second hidden node, and bh3 is the bias for the third. Who represents the weights between the hidden layer and the output layer. Since there is only one output, there is only one node specified, which is the hidden node. The output at the bottom returns a value between 0 and 1. That value is analyzed to see whether it hits a certain threshold and whether or not the output fires.

This approach works okay, but there seems to be a lot of overlapping code. This isn't a problem with a small neural network, but when you have hundreds of nodes and dozens of layers, it becomes untenable. There is a solution for this issue, which is the topic of chapter 20, convolutional neural networks.

Summary

Neural networks model how the human brain learns. There are primitive neurons called perceptrons that are binary. There are also linear neurons known as *sigmoid neurons* that are more flexible and return a greater shade of nuance. A network architecture involves how all of these nodes are connected and how their inputs are weighted and trained against one another. This chapter focused on fully connected neural networks, which is not the only way of implementing a neural network.

20. Convolutional Neural Networks

Tiger got to hunt, bird got to fly;

Man got to sit and wonder "why, why, why?"

—Kurt Vonnegut

Chapter 18, "Neural Network Concepts," gave a high-level overview of neural networks in the context of a fully connected neural network. You may have noticed that this approach was inefficient and perhaps thought that maybe there was a better way. This chapter focuses on a newer way of creating neural networks: *convolutional neural networks*. Convolutional neural networks are the reason why neural networks went from being inefficient curiosities to the cutting edge of computer science research.

History of Convolutional Neural Networks

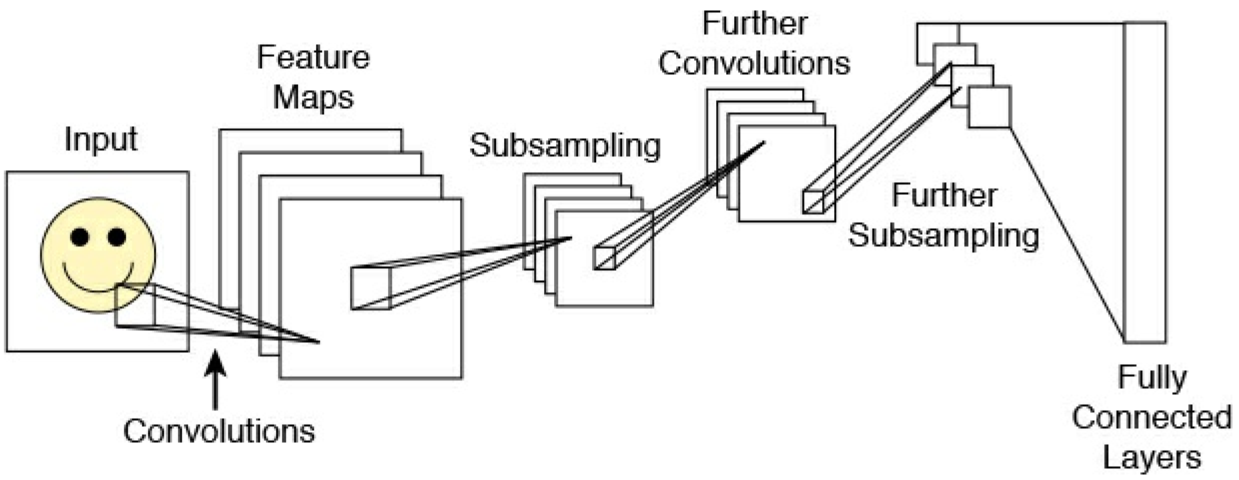
The way that your brain processes information is an iterative process. When you see an object, the top layer of your awareness takes in gross characteristics such as edges and color. The next level of this awareness refines these bits and pieces to detect individual features such as noses and eyes. At the deepest level, these refinements come together, and your brain recognizes an object. That is the foundation for how convolutional neural networks function to classify images.

In a fully connected neural network, in every layer of neurons, each neuron is fully connected to every neuron both ahead and behind it. If you have a small image with only 100 pixels square and each pixel is an input, then you have an input layer of 1,000 neurons, and each one of those thousand neurons is connected to every neuron in the first hidden layer.

As discussed in Chapter 18, the surrounding pixels are summed and processed with the selected pixel, which results in a large number of pixels being sampled. When you move on to the next pixel, you're resampling a certain number of pixels that were already sampled and summed in the previous operation.

The earliest convolutional neural networks were implemented in 1994. The first neural network, *LeNet5*, was created by Yann LeCun. The breakthrough with LeNet5 was the understanding that identifiable image features exist and can be detected throughout an image (see Figure 20.1). These features can then be extracted and used as training parameters and compared to similar features in other images.

Figure 20.1. *A visual representation of LeNet5, the first convolutional neural network*



In 1994, there were no GPUs to train neural networks, and CPUs were significantly slower. LeNet5 allowed you to save parameters and computation instead of using each pixel as a separate input. LeNet5 utilized using multiple layers to filter out relevant potential features rather than using a brute force approach to find these features.

LeNet5 contributed several advances to neural networks:

• Using a sequence of three layers for convolutional neural networking: convolution, pooling, and fully connected at the end

• Using convolution to extract spatial features

• Subsampling using spatial average of image maps

• Use of sigmoid neurons

• Using a multilayer neural network as the final classifier

• Sparse connection matrix between layers to avoid unnecessary large computational costs

Interest and research in artificial intelligence ebbs and flows. Because of lack of computing power, neural networks weren’t practical, so there was not a lot of interest or progress in neural networks until 2010. That’s when scientists realized that GPUs could be used to train data models; this realization sparked renewed interest in neural networks.

The current interest in convolutional neural networks and using the GPU for training was further boosted in 2012 by Alex Krizhevsky. He entered a convolutional neural network in the ImageNet competition and won by an incredibly large leap in accuracy over anything that was achieved before. Recent research into convolutional neural networks has resulted in the accuracy of many image identification algorithms having doubled in just 5 years.

AlexNet pushed the insights of LeNet5 to the next level, as shown in Figure 20.2. The contributions allowed much larger networks to be created that can identify more complex objects. These contributions include

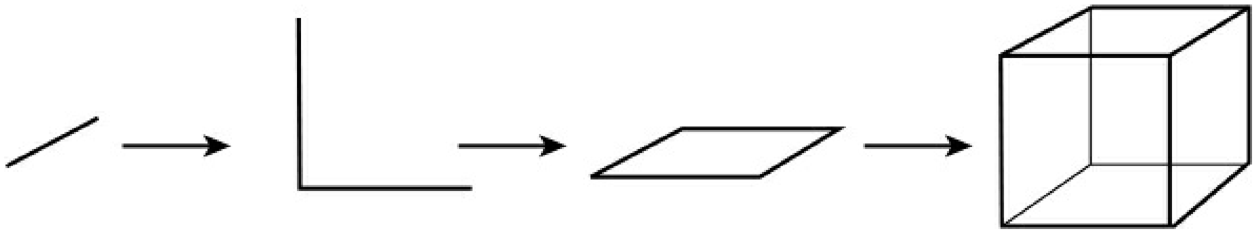
• Use of rectified linear units (ReLU) as nonlinearities

• Use of dropout technique to selectively ignore single neurons during training, a way to avoid overfitting of the model

• Overlapping max pooling, avoiding the averaging effects of average pooling

• Use of GPUs NVidia GTX 580 to reduce training time

Figure 20.2. *With AlexNet, each layer refines the image, from base components such as lines to complex images such as cubes composed of base components.*



In fully connected layers, inputs and outputs are 1D vectors of numbers. Convolution layers work on 3D data. Each color channel’s components are on their own plane of data. The convolution kernel works similarity to how it does with 1D and 2D data. It samples pixels around the input and sums them. Rather than having an input for every single pixel, the convolution kernel does not overlap.

If you had an image that was 100 pixels square, with traditional fully connected layers, you would have 1,000 inputs. With convolution layers, you would have only 34 inputs because you are not resampling the same nine pixels over and over again. The remainder of this chapter details how this works in Metal and how to set it up using the Metal Performance Shaders (MPS) framework.

MPSImage

The first object you need to be familiar with to work with convolutional neural networks is MPSImage. Convolutional neural networks work with images to extract features and do supervised learning. As you read previously, each image is broken up into multiple channels. A texture for a neural network could have as many as 64 channels. MTLTexture objects are not set up to send more than four channels, so you need a special texture to contain those channels for the network.

To create an MPSImage, you need an MPSImageDescriptor. A common pattern in Metal programming is that various objects are created using descriptor objects. There are two different methods to create an MPSImage: one for a single image and one that contains multiple images. The following properties are necessary to create an MPSImage:

• **Channel format**: Represented by an MPSImageFeatureChannelFormat enum. This property encodes the representation of a single channel within an image. The encoding can be unsigned integers or floats, depending on the precision necessary for your network.

• **Width**: The width of the image.

• **Height**: The height of the image.

• **Feature channels**: The number of feature channels per pixel.

• **Number of images**: The number of images for batch processing. This property isn't required if you have only one image.

• **Usage**: Represented by a MTLTextureUsage structure. This property contains the options that describe how a texture will be used. The texture can only be used in the ways specified by its usage values. These usage values include the shader read, shader write, and render target.

Another flavor of MPSImage is MPSTemporaryImage. This class exists to store transient data that is used and discarded immediately. The storage mode for the temporary image is private. It is used to increase performance and reduce memory footprint. It's simply an image cache to be used and discarded.

The MPSImage and MPSTemporaryImage constitute the data that is sent to the neural network. The actual processing is done by the various layers available in the MPS framework, which is detailed in the rest of this chapter.

Convolutional Neural Network Kernels

The base class for all neural network layers is MPSCNNKernel. The gist of what this class does is that it consumes an MPSImage object, processes it, and returns a new object. The areas of the image that are processed and how are specified by properties on the MPSCNNKernel class.

One important property on MPSCNNKernel is the MPSOffset. This is a signed set of *x*, *y*, and *z* coordinates specifying how far in from the upper left the processing will occur. There is also an optional property on MPSCNNKernel called clipRect that specifies a MTLRegion where the pixel data will be overwritten. If this property isn't specified, the entire region is overwritten.

You also need to specify the MPSImageEdgeMode. You want to specify the behavior of the filter when it reads beyond the bounds of the source image. This can happen because of a negative offset property, because the value of offset + clipRect.size is larger than the source image, or because the filter looks at neighboring pixels, such as a convolution filter. This is set to 0 by default.

The last property you need to set up on the MPSCNNKernel is destinationFeatureChannelOffset. It represents the number of channels in the destination image to skip before writing output data. For example, suppose a destination image has 24 channels and a kernel outputs 8 channels. If you want channels 8 to 15 of this destination image to be used for the output, you can set the value of the destinationFeatureChannelOffset property to 8.

There are many categories of layer types, but this chapter focuses on the three most common types: convolution, pooling, and fully connected.

Convolution Layer

The *convolution layer* (Figure 20.3) is the foundational and most important layer in your neural network. This layer is responsible for extracting features from the input that you provide. To create a convolution layer, you need an MPSCNNConvolutionDescriptor:

```
// create convolution descriptor with appropriate stride
let convDesc = MPSCNNConvolutionDescriptor(
    kernelWidth: Int(kernelWidth),
    kernelHeight: Int(kernelHeight),
    inputFeatureChannels: Int(inputFeatureChannels),
    outputFeatureChannels: Int(outputFeatureChannels)
)

convDesc.strideInPixelsX = Int(strideXY.0)
  convDesc.strideInPixelsY = Int(strideXY.1)
```

Here is a more detailed overview of the properties you need to create a MPSCNNConvolutionDescriptor:

• **kernelWidth**: The width of the kernel window

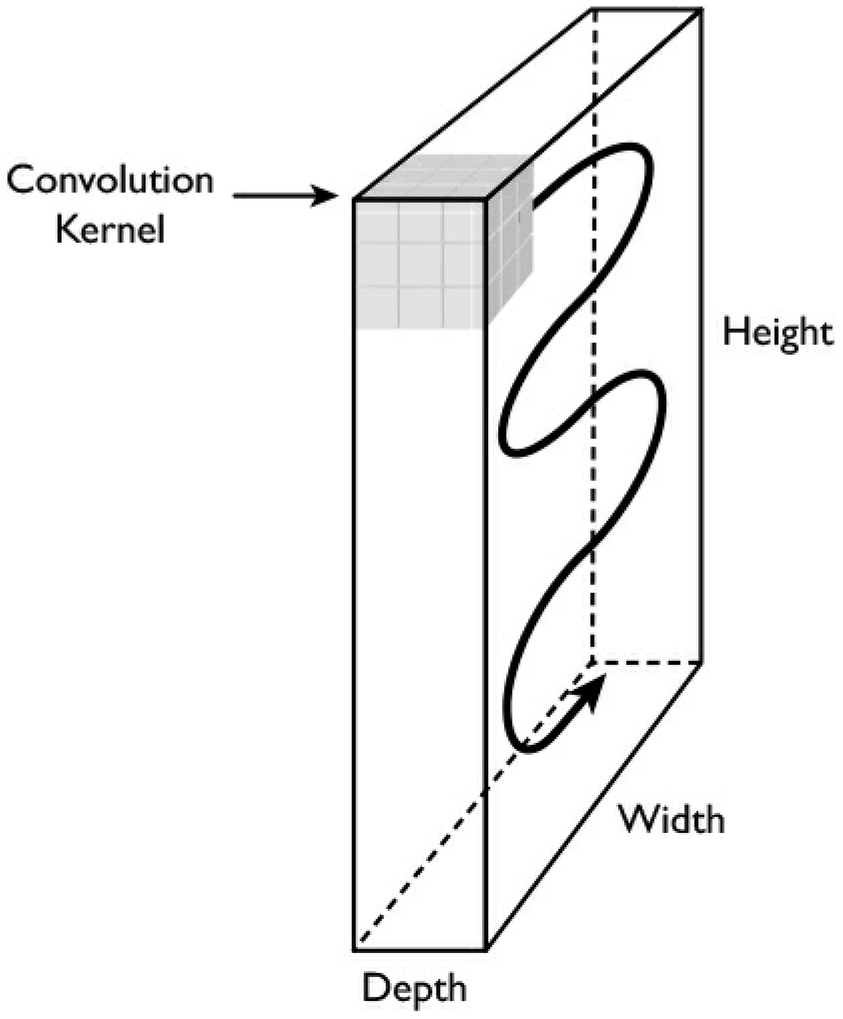
• **kernelHeight**: The height of the kernel window

• **inputFeatureChannels**: The number of feature channels per pixel in the input image

• **outputFeatureChannels**: The number of feature channels per pixel in the output image

The input and output feature channels do not need to be the same size. However, the kernel width and height do need to be the same size, as shown in Figure 20.3. The width and height do not need to be an odd number.

Figure 20.3. *Scanning of an input image using a convolution layer*



Along with the initializers, you have to set the stride for the convolution descriptor. You need to set the stride in both the X and Y directions. These specify the downsampling factor in each dimension.

Next, you use that convolution descriptor to initialize an MPSCNNConvolution kernel:

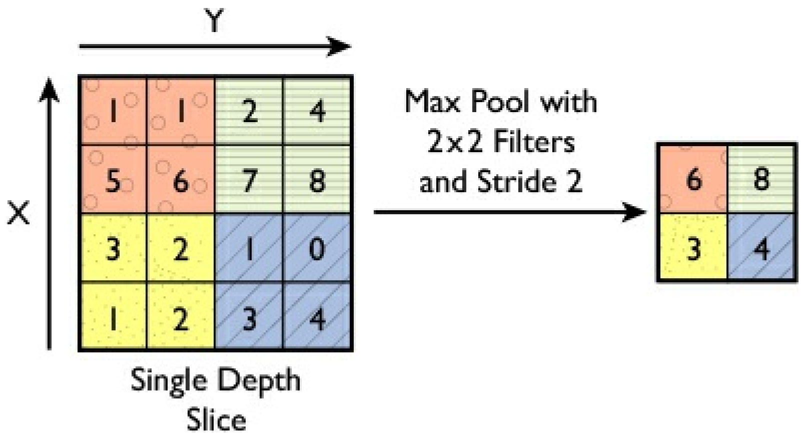
```
init(device: MTLDevice,
   convolutionDescriptor:  MPSCNNConvolutionDescriptor,
   kernelWeights: UnsafePointer<Float>,
   biasTerms: UnsafePointer<Float>?,
   flags: MPSCNNConvolutionFlags)
```

MPSCNNConvolution is a subclass of MPSCNNKernel. It takes the default MTLDevice as a parameter along with the MPSCNNConvolutionDescriptor you just created. Up next, you enter the kernel weights and biases, which you obtain from the trained data models. The MPSCNNConvolutionFlags are the options used to control how the kernel weights are stored and used. Right now, the only value for that is .none.

Pooling Layer

The role of the *pooling layer* is to reduce the spatial size of the images in the network. This layer condenses the information in a region of an image, as shown in Figure 20.4. It's common to insert a pooling layer between successive convolution layers.

Figure 20.4. *How a pooling layer condenses its input data*



The base class for pooling layers is MPSCNNPooling. This is also a subclass of MPSCNNKernel as well as every other kernel that is used to build a neural network. That base class has a method, encode(commandBuffer:sourceImage:), that is used to encode the MPSCNNPooling object to a MTLCommandBuffer object.

There are two commonly used types of pooling layers: max and average. The MPSCNNPoolingMax filter returns the maximum value of the pixels in the filter region for each pixel in the image. The MPSCNNPoolingAverage filter returns the average value of the pixels in the filter region.

An implementation of a pooling layer looks something like this:

```
pool = MPSCNNPoolingMax(device: device,
                        kernelWidth: 2,
                        kernelHeight: 2,
                        strideInPixelsX: 2,
                        strideInPixelsY: 2)
```

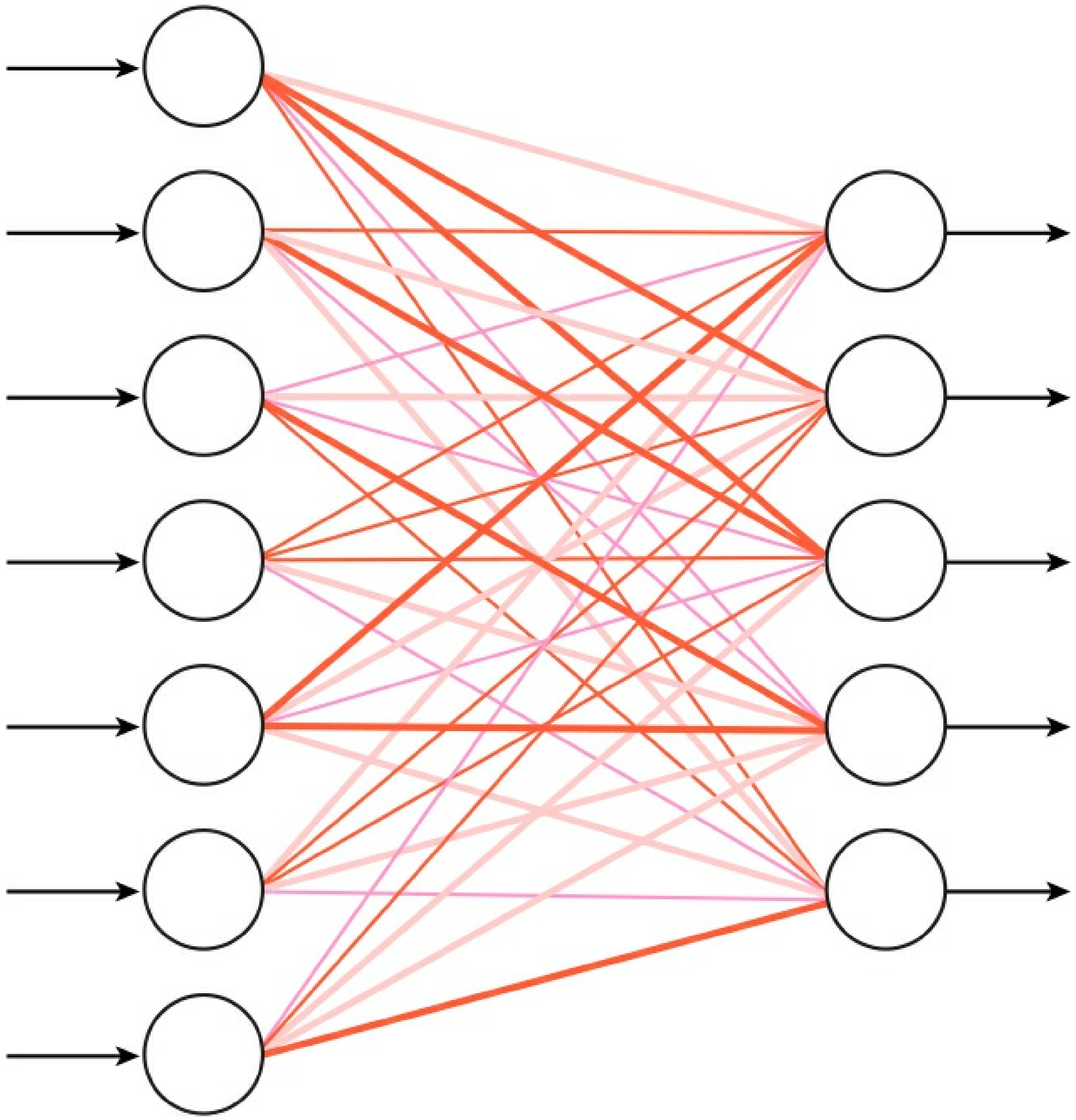
The default behavior for the kernel width and height is to set both to 2 pixels. You don't have to set it to 2, but if you don't have a good reason to make it different, 2 is a good default value.

Fully Connected Layer

The *fully connected layer* is usually the last layer in your process. Think about how hiring works: You receive dozens of resumes, which are filtered and narrowed down to a dozen people who get phone interviews. Those people are winnowed down into a couple of people who get on-site interviews. Then a final decision is made as to which person get the job offer. The previous layers were doing the narrowing and winnowing of the data, and the job of the fully connected layer is to make the final determination based on all of that processing.

Earlier, you learned about fully connected layers. Fully connected layers have the same filter size as input size. They are similar to convolution layers, except you are filtering every single input. A neural network commonly has multiple convolution and pooling layers that eventually feed into one fully connected layer, as shown in Figure 20.5. The purpose of the convolution and pooling layers is to shrink the input so that the final fully connected layer can bring everything together. It's not optimal for every layer to be fully connected, as fully connected layers are expensive. The fully connected layer is the culmination of all the processing you are doing on the neural network.

Figure 20.5. *Fully connected layers in a neural network*



```
init(device: MTLDevice,
     convolutionDescriptor fullyConnectedDescriptor:
         MPSCNNConvolutionDescriptor,
     kernelWeights: UnsafePointer<Float>,
     biasTerms: UnsafePointer<Float>?,
     flags: MPSCNNConvolutionFlags)
```

The MPSCNNFullyConnected layer is a convolution layer, so it requires an MPSCNNConvolutionDescriptor. All the other parameters should look familiar because they are the same as for a convolution layer.

Convolution Data Source

There is another initializer for a fully connected layer that has an alternative source for the data:

```
init(device: MTLDevice,
    weights: MPSCNNConvolutionDataSource)
```

The MPSCNNConvolutionDataSource protocol is an alternative way to encode the weights and biases for a convolution kernel. This protocol includes many of the convenience functions for returning objects that are necessary to set up an MPSCNNConvolution kernel. This includes the bias terms, data type, and descriptor.

Here is an example of how to do this:

```
class DataSource: NSObject, MPSCNNConvolutionDataSource {
  let name: String
  let kernelWidth: Int
  let kernelHeight: Int
  let inputFeatureChannels: Int
  let outputFeatureChannels: Int
  let useLeaky: Bool

  var data: Data?

  func load() -> Bool {
    if let url = Bundle.main.url(forResource: name,
        withExtension: "bin") {
      do {
        data = try Data(contentsOf: url)
        return true
      } catch {
        print("Error: could not load \(url): \(error)")
      }
    }
    return false
  }

  func purge() {
    data = nil
  }

  func weights() -> UnsafeMutableRawPointer {
    return UnsafeMutableRawPointer(mutating:
        (data! as NSData).bytes)
  }

  func biasTerms() -> UnsafeMutablePointer<Float>? {
    return nil
  }
}
```

This data source conforms to the MPSCNNConvolutionDataSource and thus has many methods necessary to care for and load the data.

Neural Network Graph

Prior to iOS 11, you were required to create large, complex structures using the various convolutional neural networking objects. These are mentally visualized as a graph, but this structure didn’t exist in the MPS framework to simplify the creation of neural networks. New for iOS 11 is the introduction of the MPSNNGraph class.

MPSNNGraph is an optimized representation of a graph of neural network image and filter nodes. The nodes that constitute a graph are

• MPSNNImageNode

• MPSNNFilterNode

• MPSNNStateNode

These act as base nodes for the MPSNNGraph. The MPSNNImageNode is a representation of an MPSImage that is used as the base node for a graph. The MPSNNFilterNode indicates that the following nodes will create a filter stage. Lastly, the MPSNNStateNode represents a state object.

Within the MPS framework, a set of node classes represents the classes that exist within each layer type. For example, in the pooling layer is a class called MPSCNNPoolingMax that creates a pooling layer. There is a correlating node class called MPSCNNPoolingAverageNode. Each of these nodes takes an input image from the layer above it and applies the weights and biases for the data source. Here is an example:

```
let inputImage = MPSNNImageNode(handle: nil)

let conv1 = MPSCNNConvolutionNode(source: inputImage,
    weights: DataSource("conv1", 3, 3, 3, 16))

let pool1 = MPSCNNPoolingMaxNode(source: conv1.resultImage,
    filterSize: 2)

let conv2 = MPSCNNConvolutionNode(source: pool1.resultImage,
    weights: DataSource("conv2", 3, 3, 16, 32))

let pool2 = MPSCNNPoolingMaxNode(source: conv2.resultImage,
    filterSize: 2)

// ... and so on ...

guard let graph = MPSNNGraph(device: device,
                             resultImage: conv9.resultImage) else {
    fatalError("Error: could not initialize graph")
}
```

To begin an MPSNNGraph, you need to start with one of the base node types. That node acts as the source node for the first convolution node. The result of that node feeds into a pooling node. This continues until you have declared your entire graph. The final node is used to initialize the MPSNNGraph object.

Summary

Convolutional neural networks have been around for over 20 years, but only since the development of the GPU have they become truly practical. A convolutional neural network can be composed of several different types of layers, but the most common set of layers include convolution, pooling, and fully connected. These networks can be created using the MPSImage object, or they can be connected using an MPSNNGraph object.