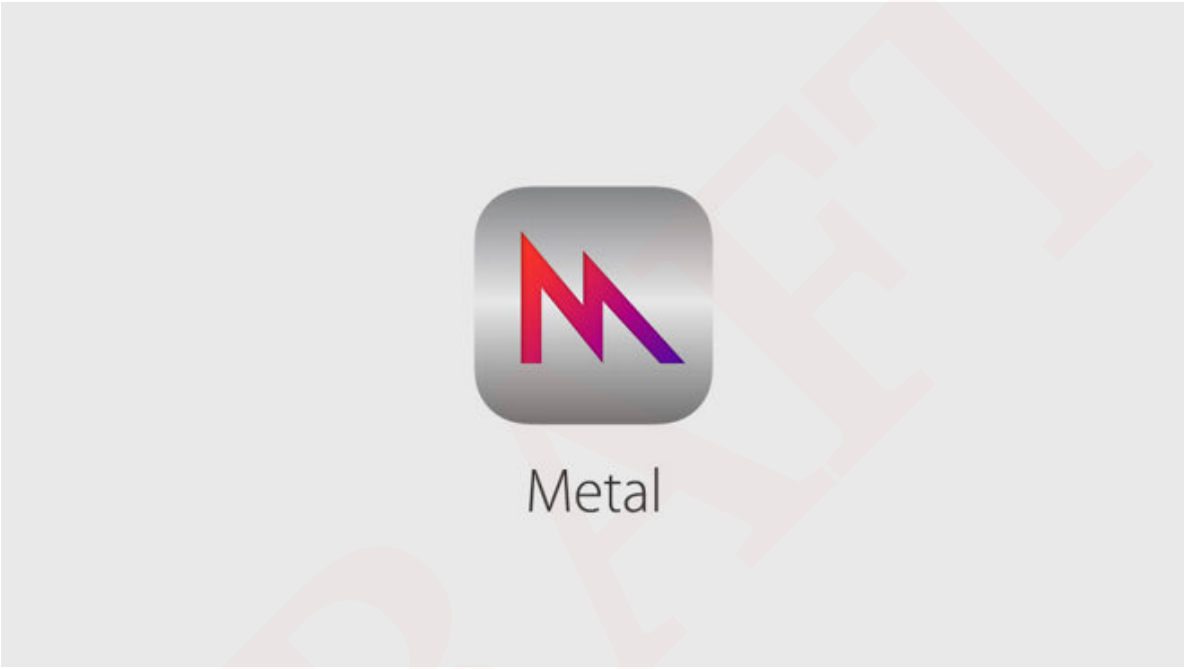
KENWRIGHT

METAL API CRASH COURSE

TECHNICAL COURSE

GETTING STARTED EDITION



Metal API: Crash Course *Kenwright*

COURSE TITLE:
Metal API: Crash Course
(Getting Started Edition)

Revision: 032902121019

First Daft, January 2019

Metal

Brief Contents

Metal

Contents

Quote

"I think the more realistic you try to make the graphics and the experience, the more you limit yourself to a single vision.

(Markus Persson)

1. Introduction and Overview

1.1 About

These notes complement the crash course to getting started with Apple's Metal API. The course focuses on the practical aspects with hands-on details, such as, simplified code snippets and step-by-step explanations. The notes have been formatted and designed, so whether or not you are currently an expert in computer graphics, actively working with an existing API (e.g., DirectX/OpenGL), or completely in the dark about this mysterious topic, this crash course has something for you. If you're an experienced developer, you'll find this crash course a light refresher to the subject, and if you're deciding whether or not to delve into graphics and the Metal API, this crash course may help you make that significant decision. This is an ambitious subject to cover in a crash course, but not unrealistic, and we know that computer graphics is a little bit of an art and involves a variety of skills and abilities. There is so much more to know than this crash course is able to present - however, it presents the essential facts of the subject with a high-level introduction to the core components and their mechanics. For the sake of practicality, this crash course discusses the important aspects of the Metal API, from a 'getting started' perspective, such as, minimum working examples, what each component does, setting up a Metal project, performance factors and real-world considerations.

The example program listings should be sufficiently simple enough for the reader to easily type in and test while working through the notes.

Figure 1.1: The Metal API is designed and maintained by Apple Ltd. The purpose of the API is to provide optimal hardware compatibility and performance for rendering and compute solutions within Apple systems.

1.2 Computer Graphics

Computer graphics is an exciting and important multi-discipline subject with applications in:

- video games,
- virtual reality
- image and video processing,
- graphical modeling,
- augmented and virtual reality,
- production/tool optimisation (CPU/GPU),
- real-time solutions,
- rendering & simulation,
- visual effects,
- user interaction
- robotics
- ...

Computer graphics covers topics from extraction and visualisation to generation and manipulation in both 2-dimensional and 3-dimensional contexts. In this course, you'll focus primarily on 3-dimensional visual solutions. However, you'll still require and apply 2-dimensional principles like texture manipulation and mapping to pixel and screen space effects (e.g., blurring, edge detection and smoothing). You'll discover that computer graphics gives you the power to create worlds of infinite possibilities (e.g., from chocolate cities 'choco-land' to real-world locations like London) or help visualise complex problems (like structural stress in buildings or the workings of internal organs in the human body). The implementations can range in complexity as well - from a simple single triangle with no lighting or texturing requiring a couple of hundred lines of code to a complete renderer engine that's able to display realistic human models accurately down to the hairs on their head (requiring thousand or more lines of code with dozens of different shaders and optimisations). What is more, these solutions may be off-line taking minutes or days to calculate or microseconds for real-time interactive virtual environments (video games).

Name: 'Metal'
Metal gets its name from its low level of hardware optimization, as it runs on 'the bare metal', rather than hovering over a large hardware abstraction layer in the model of cross platform graphics frameworks like OpenGL/DirectX, which were designed to support a wide range of processors.

1.3 Aim of this Crash Course

This course aims to introduce computer graphics programming in a practical context while addressing a number of crucial questions with regard to 'another' graphical application programming interface (API), for example:

At the end of this course, you should feel comfortable enough to work with the Metal API (i.e., create, customize and generate a variety of simple graphical applications). You should be able to explain the core components of the API, and importantly, why and how they fit together to accomplish the necessary graphical technique [3, 2].

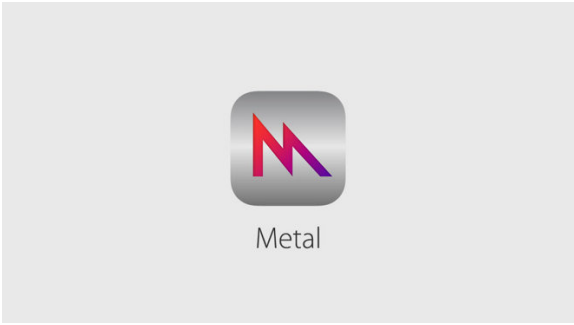


Figure 1.2: Metal has a steep learning curve initially - but over time the benefits and freedom provided by the API are rewarded compared to existing solutions (greater optimisations and customisability).

- ✔ What exactly is Computer Graphics and the Metal API?
- ✔ Why is understanding the 'differences' between the API important?
- ✔ How do you to get started programming a graphical application with Metal?
- ✔ Understanding where and why a graphical program 'fails' - e.g., perform worse than current or existing graphical API
- ✔ Dealing with problems, such as, cross-platform, memory leaks, graphical issues, rapid prototyping, versions, ...
- ✔ How to work effectively on complex projects with Metal
- ✔ Background introduction to the history of different graphical API
- ✔ Revision on basic graphical principles and techniques (shaders, lighting, transforms, triangles)
- ✔ Managing Metal API (structured modular programming)
- ✔ Implement a basic graphical application from the ground up using the native Metal API
- ✔ Essential graphical principles and how to implement them with Metal
- ✔ How to implement popular graphical effects (e.g., lighting, bump maps, instancing and texturing)

To be clear - the emphasis of the examples is to present minimal (basic) implementations from which you can extend and build upon (i.e., not to present a framework or lots of wrapper classes - but a raw taste of the core principles). Once you're up and running, there is a massive number of resources online with additional material and information to enable you to go above and beyond with Graphics and Metal.

The aims of these 'minimal working examples' within the course are:

- → have fun learning graphics and Metal
- → review and learn the underpinning graphical concepts
- → help simplify complicated ideas
- → present minimal working samples

- help make the topic interesting (simple but powerful graphical tricks)
- hands-on approach

'Not' the aim:

- develop a framework or engine
- build an entire game or library
- compare and develop cross platform libraries

1.4 Prerequisite (Setting-up Metal API)

Pre-requisites to working with the Metal API The computer graphics samples in this course are built around the Metal API - hence, to implement and run the examples you'll need to have a Apple Mac (with iOS 12), an editor/compiler (e.g., XCode), and of course, the Metal API SDK libraries on your machine.

To download and install the necessary Metal API drivers and SDK (if you don't already have them installed on your system) is very straightforward.

In addition, you'll need to have a basic understanding of core programming principles (e.g., functions, declarations, libraries and the ability to read simple computer programs written in Swift, Javascript, C, C++ or Java). While basic knowledge of computer graphics concepts would be beneficial (for example, framebuffers and refresh rate), it's not required, as you'll be guided through the process of writing a basic applications that utilizes Metal to perform simple graphical operations.

The practical examples in this crash course will be implemented using Swift.

1.5 Why Swift?

Despite its 'youth' Swift is becoming more popular among iOS developers. This can be explained, first of all, by its extreme clarity, even for beginners. To list but a few of the main advantages of Swift programming language:

- easy to learn,
- syntax is simple to understand,
- makes iOS development less complex ('smooth'),
- good error handling, object-orientated,
- good performance metrics, and

- supports dynamic linking/libraries.

1.6 Summary

These are exciting times for computer graphics. With advancements in technologies, you'll continue to see breakthroughs in realism and creativity. The resources for create amazing graphical effects is within your grasp (e.g., massive amounts of information and material online, including all sorts of free assets like 3-dimensional models). While computer graphics programming can seem daunting and difficult initially - especially if your mathematics is a bit rusty - the rewards at the end are well worth the time and effort.

2. Background (Apple and Metal API)

2.1 Overview

Apple launched the Metal 1.0 specification in February 2014.

Since Metal was first made available in 2014 on iOS devices powered by an Apple A7 or later, it has sat in the background compared to other API, such as, OpenGL and DirectX. However, the API follows the same core principles, to reduced engineering complexities and provide an intuitive interface for developers without hindering flexibility or power. The ability to accomplish stunning computer generated images is easier than ever with Metal. Computer graphics has become increasingly challenging using conventional approaches and expectations have and continue to grow, especially in areas involved with films, games and virtual reality. One specific challenge is the ability to exploit the advancements in rapidly changing technologies. For example, despite the ready availability of multiple high performance graphics cards, the limitations of existing libraries has made it difficult if not impossible to exploit the full potential of the hardware (distributing the workload for processing and rendering high fidelity images in real-time across multiple devices efficiently [1]). While parallel processing paradigms have become an attractive solution in recent years, with multiple cores and threads working together to offering tremendous performance gains, developing parallel applications that exploit these parallel speed-ups efficiently and reliably is a significant challenge.

Metal is an exciting multi-platform cross-language graphical and compute interface that exploits the latest 'parallel' hardware architectures. Metal provide you and developers with a powerful interface to create stunning visuals for a wide range of applications. Metal still follows the same original 'OpenGL' initiatives, i.e., to develop a high quality open source, cross-platform API (Mac, Windows, Linux, Android, Solaris and FreeBSD). OpenGL has come a long way and done amazingly well over the last 25 years (Figure **??**). Be that as it may, it is time for a major update. As the original OpenGL API follows a state machine architecture this ties the API to a single on-screen context. In addition the OpenGL API is blind to everything the GPU is doing (optimised and managed within the driver - and hidden from the developer). Metal takes a different approach - following an object-based API with no global state so all state concepts are localized to a Command-Buffer (you'll learn about Command-Buffers in Section **??**). What is more Metal is more explicit about what the GPU is doing (less hiding what is happening within the driver).

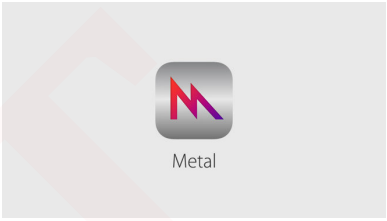


Figure 2.1: A common misconception is the naming of the Metal API. Often assumed the Metal API is named after the Vulcan race from Star Trek. However, Metal means 'volcano', and is named such as Vulcan is supposed to make low level, cross platform API erupt in the market. Volcanoes are also often regarded as being of godlike power (Roman god, as well as the Cretan god Volchanos), as they can both destroy civilizations and create new landmasses entirely. Naming the API to something that has a connotation of unparalleled power kind of pushes the idea of it being a powerful thing across. Another interesting fact - Metal is based upon the foundation work of Mantle from AMD - the Earth's mantle and plate tectonics drive volcanic activity, so the names tie into each other.

Metal was introduced in 2014 as a general purpose API for GPU-based computation. In 2018, Apple deprecated OpenGL in iOS 12 for both iOS and macOS.

API improvements:

Explicit Control
Multi-Threading Friendly
Direct State Access (DSA)
Bindless Graphics
Framebuffer Memory Info
Texture Barrier
Acceleration for applications (e.g., Browsers, WebGL, ..)

The principle of explicit control, means you promise to tell the driver every detail. So the driver doesn't have to guess or make assumptions. In return, the driver is more streamlined and efficient (does what you asked for when you asked for it quickly). For instance, memory management in Metal gives the control to the application (total memory usage is more visible and simplifies operations, such as as for streaming data). Remember, the application is in charge (so doing it correctly is your responsibility).

While the latest OpenGL graphical API (known as Metal) might seem like another iteration, it is well worth learning or even reviewing. At the same time, Metal is in its first release (revision 1.0) - and possesses a huge number of changes/improvements compared to any previous update. Importantly, these improvements should not be ignored, as they offer possibilities that were previously not feasible. These ad-

Figure 2.2: Evolution of the Metal API to the most recent release known as 'Metal 2'.

ditional features in Apple's latest update to Metal will help you get more out of GPUs, in addition to making debugging/profiling more transparent. However, to gain improvements it is important you understand the differences (i.e., applications need to be written differently to utilize these additional features and control - OpenGL ! = Metal). Metal's abstraction means your application is much closer the hardware compared to traditional APIs. Your application is driving the hardware directly, leaving just enough abstraction to reduce the burden of writing your own driver. You're not being second-guessed by the driver, while at the same time you're not being first-guessed either. You now have all the control you need to get the best out of your hardware. If it doesn't go fast in Metal, it's your fault (of course, remember, with great power comes great responsibility).

A few of the "big tick" items with Metal is:

- Explicit control,
- Support for multi-core/threading,
- Predictability, and
- Bandwidth efficiency.

2.2 History of Metal

The Metal API was designed and is maintained by the Apple Ltd. to meet current and future demands for achieving high performance rendering and compute solutions. The Metal API achieves this by allowing greater low level control (explicitly) - moving away from 'default' parameters/assumptions set within the driver. The developer has to manage the memory, resource updates, batching, scheduling, ... Hence, the Metal API initially seems verbose and complicated due

Figure 2.3: **Metal 2** is forward thinking with support for future technologies, such as, ML, AR and VR. Apple announcing Metal 2 for macOS High Sierra, with improvements including a new shader debugger and GPU dependency viewer for more efficient profiling and debugging in Xcode; support for accelerating the computationally-intensive task of training neural networks, including machine learning; lower CPU workloads via GPU-controlled pipelines, where the GPU is able to construct its own rendering commands and schedule them with little to no CPU interaction; and support for Virtual Reality.

to the large amount of initiation and management (through functions, parameters and structures), yet this is crucial for Metal's success. It should also be noted, that DirectX 12 from Microsoft follows a similar design to Metal (explicit low level control). For instance, previously, 'OpenGL' did not address multi-threading and was not designed to support the concurrent and parallel paradigm which would be a serious problem in todays multi-core multi-threaded environment. However, the Metal API is designed to exploit these multi-threaded environments (and is how it is able to outperforms previous API).

2.2.1 Mobile Devices and A11 Chip

Metal 2 supports devices with iOS using the new A11 Bionic (such as, the iPhone 8, 8 Plus, and X models). This chip also launched Apple's first independent GPU of its own design, even more tightly optimized for not just accelerating graphics but also accelerating machine learning and Augmented Reality (which includes applications such as VR and face tracking).

2.2.2 Metal Similar to OpenGL ES

In iOS 8, Apple released its own API for GPU-accelerated 3D graphics: Metal.

Metal is similar to OpenGL ES in that it's a low-level API for interacting with 3D graphics hardware.

The difference is that Metal is not cross-platform. Instead, it's designed to be extremely efficient with Apple hardware, offering improved speed and low overhead compared to using OpenGL ES.

In this tutorial, you'll get hands-on experience using the Metal API to create a bare-bones app: drawing a simple triangle. In doing so, you'll learn some of the most important classes in Metal, such as devices, command queues and more.

This tutorial is designed so that anyone can go through it, regardless of your 3D graphics background - however, things will move along fairly quickly. If you do have some prior 3D-programming or OpenGL experience, you'll find things much easier, as many of the same concepts apply to Metal.

Figure 2.4: Metal apps do not run on the iOS simulator; they require a device with an Apple A7 chip or later. To complete this tutorial, you'll need an A7 device or newer.

OpenGL ES is designed to be cross platform. That means you can write C++ OpenGL ES code, and, most of the time, with some small modifications, you can run it on other platforms, such as Android.

Apple realized that, although the cross-platform support of OpenGL ES was nice, it was missing something fundamental to how Apple designs its products: the famous Apple integration of the operating system, hardware and software as a complete package.

So Apple took a clean-room approach to see what it would look like if it were to design a graphics API specifically for Apple hardware with the goal of being extremely low overhead and performant, while supporting the latest and greatest features.

The result is Metal, which can provide up to 10*x* the number of draw calls for your app compared to OpenGL ES. This can result in some amazing effects.

2.3 Metal 'Interface'

You must remember the purpose of Metal and how it compares to higher-level frameworks like Unity.

Metal is a low-level 3D graphics API, similar to OpenGL ES, but with lower overhead meaning better performance. It's a very thin layer above the GPU, which means that, in doing just about anything, such as rendering a sprite or a 3D model to the screen, it requires you to write all of the code to do this. The trade-off is that you have full power and control.

Conversely, higher-level game frameworks like Unity are built on top

of a lower-level 3D graphics APIs like Metal or OpenGL ES. They provide much of the boilerplate code you normally need to write in a game, such as rendering a sprite or 3D model to the screen.

If all you're trying to do is make a game, you'll probably use a higher-level game framework like SpriteKit, SceneKit or Unity most of the time because doing so will make your life much easier. If this sounds like you, we have tons of tutorials to help you get started with Apple Game Frameworks or Unity.

So why learn Metal? There are two really good reasons to learn Metal:

- Enables developers (such as you) to push the hardware to its limits. Since Metal is a low level API, it allows you to really push the hardware and have full control over how your application works
- Learning Metal teaches you a lot about 3D graphics, writing your own application gives you enormous insights into graphical concepts and optimization tricks

Figure 2.5: On iOS and tvOS, Metal supports Apple-designed SoCs (System On Chip) from the Apple A7 or newer. On macOS, Metal supports Intel HD and Iris Graphics from the HD 4000 series or newer, AMD GCN-based GPUs, and Nvidia Kepler-based GPUs or newer.

2.4 Steps

Xcode's iOS game template comes with a Metal option, but you won't choose that here. This is because you're going to put together a Metal app almost from scratch, so you can understand every step of the process.

Download the files that you need for this tutorial using the Download Materials button at the top or bottom of this tutorial. Once you have the files, open HelloMetal.xcodeproj in the HelloMetal_starter folder. You'll see an empty project with a single ViewController.

There are seven steps required to set up Metal so that you can begin rendering. You need to create a:

- MTLDevice
- CAMetalLayer
- Vertex Buffer
- Vertex Shader
- Fragment Shader
- Render Pipeline
- Command Queue

2.5 Chapter Review

2.5.1 Questions

Question When was Metal first released?

Question What has changed with the latest revision of Metal (why has it changed)?

Question What is the root methodology behind Metal compared to previous graphical API?

Question Research the differences between iOS Sierra and High-Sierra in relation to the Metal API

3. Programming

3.1 Overview

This chapter is split into two parts, the first part explains how to get Metal up and running (i.e., initializing the core components through code), while the second part builds upon this to develop the graphical output and animation (emphasis on getting something on screen).

3.2 Getting Up and Running

3.2.1 Creating an MTLDevice

You'll first need to get a reference to an **MTLDevice**.

Think of **MTLDevice** as your direct connection to the GPU. You'll create all the other Metal objects you need (like command queues, buffers and textures) using this **MTLDevice**.

To do this, open **ViewController.swift** and add this import to the top of the file:

```
1 import Metal
```

This imports the Metal framework so that you can use Metal classes such as **MTLDevice** inside this file.

Next, add this property to the **ViewController**:

```
var device: MTLDevice!
```

You're going to initialize this property in **viewDidLoad()** rather than in an initializer, so it has to be an optional. Since you know you're definitely going to initialize it before you use it, you mark it as an implicitly unwrapped optional, for convenience purposes.

Finally, add **viewDidLoad()** and initialize the device property, like this:

```
override func viewDidLoad() {
  super.viewDidLoad()

  device = MTLCreateSystemDefaultDevice()
}
```

MTLCreateSystemDefaultDevice returns a references to the default **MTLDevice**, which your code will use.

3.2.2 Creating a CAMetalLayer

In iOS, everything you see on screen is backed by a **CALayer**. There are subclasses of **CALayers** for different effects, such as gradient layers, shape layers, replicator layers and more.

If you want to draw something on the screen with Metal, you need to use a special subclass of **CALayer** called **CAMetalLayer**. You'll add one of these to your view controller.

First, add this new property to the class:

```
var metalLayer: CAMetalLayer!
```

This will store a handy reference to your new layer.

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

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

Going over this line by line:

Figure 3.1: If you get a compiler error at this point, make sure that you set the app to target your Metal-compatible iOS device. As mentioned earlier, Metal is not supported on iOS Simulator at this time.

Create a new **CAMetalLayer**. You must specify the **MTLDevice** the layer should use. You simply set this to the device you obtained earlier. Set the pixel format to **bgra8Unorm**, which is a fancy way of saying "8 bytes for Blue, Green, Red and Alpha, in that order - with normalized values between 0 and 1." This is one of only two possible formats to use for a **CAMetalLayer**, so normally you'd just leave this as-is. Apple encourages you to set **framebufferOnly** to true for performance reasons unless you need to sample from the textures generated for this layer, or if you need to enable compute kernels on the layer drawable texture. Most of the time, you don't need to do this. You set the frame of the layer to match the frame of the view. Finally, you add the layer as a sub-layer of the view's main layer.

3.2.3 Creating a Vertex Buffer

Everything you draw in Metal is a 'triangle'. In this app, you're just going to draw one triangle, but even complex 3D shapes can be decomposed into a series of triangles.

In Metal, the default coordinate system is the normalized coordinate system, which means that by default you're looking at a 2x2x1 cube centered at (0, 0, 0.5).

If you consider the Z=0 plane, then (-1, -1, 0) is the lower left, (0, 0, 0) is the center, and (1, 1, 0) is the upper right. In this tutorial, you want to draw a triangle with the following three points:

You'll have to create a buffer for this. Add the following constant property to your class:

```
let vertexData: [Float] = [
   0.0,  1.0, 0.0,
  -1.0, -1.0, 0.0,
   1.0, -1.0, 0.0
]
```

This creates an array of floats on the CPU. You need to send this data to the GPU by moving it to something called a **MTLBuffer**.

Add another new property for this:

```
var vertexBuffer: MTLBuffer!
```

Then add this code to the end of **viewDidLoad()**:

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

Taking it line by line:

You need to get the size of the vertex data in bytes. You do this by multiplying the size of the first element by the count of elements in the array. You call **makeBuffer(bytes:length:options:)** on the **MTLDevice** to create a new buffer on the GPU, passing in the data from the CPU. You pass an empty array for default configuration.

3.2.4 Creating a Vertex Shader

The vertices that you created in the previous section will become the input to a little program that you'll write called a vertex shader.

A vertex shader is simply a tiny program that runs on the GPU, written in a C-like language called the Metal Shading Language.

A vertex shader is called once per vertex, and its job is to take that vertex's information, such as position - and possibly other information such as color or texture coordinate - and return a potentially modified position and possibly other data.

To keep things simple, your simple vertex shader will return the same position as the position passed in.

The easiest way to understand vertex shaders is to see it yourself. Go to File > New > File, choose iOS > Source > Metal File, and click Next. Enter **Shaders.metal** for the filename and click Create.

Add the following code to the bottom of **Shaders.metal**:

```
1  vertex float4 basic_vertex(
2    const device packed_float3* vertex_array [[ buffer(0) ]],
3    unsigned int vid [[ vertex_id ]]) {
4    return float4(vertex_array[vid], 1.0);
5  }
```

Figure 3.2: In Metal, you can include multiple shaders in a single Metal file. You can also split your shaders across multiple Metal files if you would like, as Metal will load shaders from any Metal file included in your project.

Here's what's going on in the code above:

All vertex shaders must begin with the keyword vertex. The function must return (at least) the final position of the vertex. You do this here by indicating **float4** (a vector of four floats). You then give the name of the vertex shader; you'll look up the shader later using this name. The first parameter is a pointer to an array of **packed_float3** (a packed vector of three floats) - i.e., the position of each vertex. Use the **[[...]]** syntax to declare attributes, which you can use to specify additional information such as resource locations, shader inputs and built-in variables. Here, you mark this parameter with **[[buffer(0)]]** to indicate that the first buffer of data that you send to your vertex

shader from your Metal code will populate this parameter. The vertex shader also takes a special parameter with the **vertex_id** attribute, which means that the Metal will fill it in with the index of this particular vertex inside the vertex array. Here, you look up the position inside the vertex array based on the vertex id and return that. You also convert the vector to a float4, where the final value is 1.0 - long story short, this is required for 3D math.

3.2.5 Creating a Fragment Shader

After the vertex shader completes, Metal calls another shader for each fragment (think pixel) on the screen: the fragment shader.

The fragment shader gets its input values by interpolating the output values from the vertex shader. For example, consider the fragment between the bottom two vertices of the triangle:

The input value for this fragment will be a 50/50 blend of the output value of the bottom two vertices.

The job of a fragment shader is to return the final color for each fragment. To keep things simple, you'll make each fragment white.

Add the following code to the bottom of **Shaders.metal**:

```
1  fragment half4 basic_fragment() {
2    return half4(1.0);    // return (1,1,1,1)
3  }
```

Reviewing line by line:

All fragment shaders must begin with the keyword fragment. The function must return (at least) the final color of the fragment. You do so here by indicating **half4** (a four-component color value RGBA). Note that **half4** is more memory efficient than **float4** because you're writing to less GPU memory. Here, you return (1, 1, 1, 1) for the color, which is white.

3.2.6 Creating a Render Pipeline

Now that you've created a vertex and fragment shader, you need to combine them âĂŤ along with some other configuration data - into a special object called the render pipeline.

One of the cool things about Metal is that the shaders are precompiled, and the render pipeline configuration is compiled after you first set it up. This makes everything extremely efficient.

First, add a new property to **ViewController.swift**:

```
var pipelineState: MTLRenderPipelineState!
```

This will keep track of the compiled render pipeline you're about to create.

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

```
let defaultLibrary  = device.makeDefaultLibrary()!                    1
let fragmentProgram = defaultLibrary.makeFunction(name: "basic_fragment")
let vertexProgram   = defaultLibrary.makeFunction(name: "basic_vertex")

let pipelineStateDescriptor = MTLRenderPipelineDescriptor()           2
pipelineStateDescriptor.vertexFunction   = vertexProgram
pipelineStateDescriptor.fragmentFunction = fragmentProgram
pipelineStateDescriptor.colorAttachments[0].pixelFormat = .bgra8Unorm

pipelineState = try! device.makeRenderPipelineState(descriptor:
      pipelineStateDescriptor)                                        3
```

Taking it line by line:

You can access any of the precompiled shaders included in your project through the **MTLLibrary** object you get by calling **device.makeDefaultLibrary()!**. Then, you can look up each shader by name. You set up your render pipeline configuration here. It contains the shaders that you want to use, as well as the pixel format for the color attachment - i.e., the output buffer that you're rendering to, which is the **CAMetalLayer** itself. Finally, you compile the pipeline configuration into a pipeline state that is efficient to use here on out.

3.2.7 Creating a Command Queue

The final one-time-setup step that you need to do is to create an **MTLCommandQueue**.

Think of this as an ordered list of commands that you tell the GPU to execute, one at a time.

To create a command queue, simply add a new property:

```
var commandQueue: MTLCommandQueue!
```

Then, add the following line at the end of **viewDidLoad()**:

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

Congratulations - you've managed to write the fundamental code for setting up Metal.

3.3 Getting Something on Screen

3.3.1 Rendering the Triangle

Now, it's time to move on to the code that executes each frame - to render the triangle!

This is done in five steps:

1. Create a Display Link
2. Create a Render Pass Descriptor
3. Create a Command Buffer
4. Create a Render Command Encoder
5. Commit your Command Buffer

Figure 3.3: In theory, the app doesn't actually need to render things once per frame, because the triangle doesn't move after it's drawn. However, most apps do have moving pieces, so you'll do things this way to learn the process. This also gives a nice starting point for future tutorials.

3.3.2 Creating a Display Link

You need a way to redraw the screen every time the device screen refreshes.

CADisplayLink is a timer synchronized to the displays refresh rate. The perfect tool for the job! To use it, add a new property to the class:

```
1  var timer: CADisplayLink!
```

Initialize it at the end of **viewDidLoad()** as follows:

```
1  timer = CADisplayLink(target: self, selector: #selector(gameloop))
2  timer.add(to: RunLoop.main, forMode: .default)
```

This sets up your code to call a method named **gameloop()** every time the screen refreshes.

Finally, add these stub methods to the class:

```
1  func render() {                                    1
2    // TODO
3  }
4
5  @objc func gameloop() {                            2
6    autoreleasepool {
7      self.render()
8    }
9  }
```

Here, **gameloop()** simply calls **render()** each frame, which, right now, just has an empty implementation. Time to flesh this out.

3.3.3 Creating a Render Pass Descriptor

The next step is to create an **MTLRenderPassDescriptor**, which is an object that configures which texture is being rendered to, what the clear color is and a bit of other configuration.

Add these lines inside **render()**:

```
guard let drawable = metalLayer?.nextDrawable() else { return }
let renderPassDescriptor = MTLRenderPassDescriptor()
renderPassDescriptor.colorAttachments[0].texture = drawable.texture
renderPassDescriptor.colorAttachments[0].loadAction = .clear
renderPassDescriptor.colorAttachments[0].clearColor = MTLClearColor(
  red: 0.0,
  green: 104.0/255.0,
  blue: 55.0/255.0,
  alpha: 1.0)
```

First, you call **nextDrawable()** on the Metal layer you created earlier, which returns the texture in which you need to draw in order for something to appear on the screen.

Next, you configure the render pass descriptor to use that texture. You set the load action to Clear, which means "set the texture to the clear color before doing any drawing," and you set the clear color to the green color used on the site.

3.3.4 Creating a Command Buffer

The next step is to create a command buffer. Think of this as the list of render commands that you wish to execute for this frame. The cool thing is that nothing actually happens until you commit the command buffer, giving you fine-grained control over when things occur.

Creating a command buffer is easy. Simply add this line to the end of render():

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

A command buffer contains one or more render commands. You'll create one of these next.

3.3.5 Creating a Render Command Encoder

To create a render command, you use a helper object called a render command encoder. To try this out, add these lines to the end of render():

```
let renderEncoder = commandBuffer
  .makeRenderCommandEncoder(descriptor: renderPassDescriptor)!
renderEncoder.setRenderPipelineState(pipelineState)
renderEncoder.setVertexBuffer(vertexBuffer, offset: 0, index: 0)
renderEncoder
  .drawPrimitives(type: .triangle, vertexStart: 0, vertexCount: 3, instanceCount: 1)
renderEncoder.endEncoding()
```

Here, you create a command encoder and specify the pipeline and vertex buffer that you created earlier.

The most important part is the call to **drawPrimitives(type:vertexStart:vertexCount:instanceCount:)**. Here, you're telling the GPU to draw a set of triangles, based on the vertex buffer. To keep things simple, you are only drawing one. The method arguments tell Metal that each triangle consists of three vertices, starting at index 0 inside the vertex buffer, and there is one triangle total.

When you're done, you simply call **endEncoding()**.

3.3.6 Committing Your Command Buffer

The final step is to commit the command buffer. Add these lines to the end of render():

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

The first line is needed to make sure that the GPU presents the new texture as soon as the drawing completes. Then you commit the transaction to send the task to the GPU.

Phew! That was a ton of code, but, at long last, you are done! Build and run the app and bask in your triangular glory. If you're successful, you should see a beautiful metal triangle on screen.

3.3.7 What Next?

You have learned the 'core' elements for setting up an application that uses the Metal API. This gives you a starting point, from which you can experiment and build on (e.g., once you've got one triangle on the screen, you can add more, constructing entire cities from triangles). Expand your understanding further on some of the important concepts in Metal, such as shaders, devices, command buffers, pipelines and more.

Only the beginning, every journey starts with that first step, so continue to read around the subject and consult some of the many fantastic resources available from Apple, such as:

- Apple's Metal for Developers page, with tons of links to documentation, videos and sample code
- Apple's Metal Programming Guide
- Apple's Metal Shading Language Guide
- The Metal videos from WWDC 2014

4. Moving from OpenGL to Metal

4.1 Overview

This chapter explains some similarities and differences between other popular API (e.g., OpenGL). The chapter shows how you would convert a simple OpenGL program to Metal (or vice versa).

4.2 'Similar' API

Before getting started, you may want to check out these great resources on Metal and OpenGL.

If you don't have experience with 3D graphics, don't worry! You'll still be able to follow along. If you do have some experience with 3D programming or OpenGL, you may find this tutorial easier. Many of the same concepts apply in Metal.

Figure 4.1: Metal apps don't run on the iOS simulator. They require a device with an Apple A7 chip or later. To complete this tutorial, you'll need an A7 device or newer.

4.2.1 OpenGL ES vs. Metal

OpenGL ES is designed to be a cross-platform framework. That means, with a few small modifications, you can run C++ OpenGL ES code on other platforms, such as Android.

The cross-platform support of OpenGL ES is nice, but Apple real-

ized it was missing the signature integration of the operating system, hardware, and software that all good Apple products have. So, it took a clean-room approach and designed a graphics API specifically for Apple hardware. The goal was to have low overhead and high performance while supporting the latest and greatest features.

The result is Metal, which can provide up to 10*x* the number of draw calls for your app compared to OpenGL ES.

4.3 Understanding Conceptual Differences

From a development perspective, OpenGL and Metal are similar. In both, you set up buffers with data to pass to the GPU and specify vertex and fragment shaders. In OpenGL projects, there is a **GLKBaseEffect**, which is an abstraction on top of shaders. There's no such API for Metal, so you need to write shaders yourself. But don't worry - it's not too complicated.

The biggest difference between OpenGL and Metal is that in Metal, you usually operate with two types of objects:

4.3.1 Descriptor objects and Compiled-state objects

The idea is simple. You create a descriptor object and compile it. The compiled-state object is a GPU-optimized resource. The creation and compilation are both expensive operations, so the idea is to do them as rarely as possible, and later to operate with compiled-state objects.

This approach means that when using Metal, you don't need to do a lot of setup operations on the render loop. This makes it much more efficient than OpenGL which can't do the same due to architecture restrictions.

Time to explore the code differences.

4.4 Integrating Metal

Let us assume you have an OpenGL project written in SWIFT. The different sections, will point out functions and definitions that you need to replace (i.e., convert to equivalent ones defined for Metal). For example, if you open **ViewController.swift** file, and change **ViewController** to be a subclass of **UIViewController** instead of **GLKViewController**. In Metal, there's no such thing as **MetalViewController**. Instead, you have to use **MTKView** inside the **UIViewController**.

MTKView is a part of the MetalKit framework. To access this API, add the following at the top of the file:

```
import MetalKit
```

4.5 Switching from OpenGL

Now it's time to do some OpenGL cleanup. Follow these steps:

(1) Rename both occurrences of **setupGL()** to **setupMetal()**

(2) Remove **tearDownGL()** and **deinit()** methods (with Metal, there's no need for explicit cleanup like this)

(3) Find and remove the whole extension **GLKViewControllerDelegate**, since this view controller is no longer a **GLKViewController** (note that **glkViewControllerUpdate** contains the logic for spinning. This is useful, but for now, remove it)

(4) Remove the following code from the top of **setupMetal()**:

```
context = EAGLContext(api: .openGLES3)
EAGLContext.setCurrent(context)

if let view = self.view as? GLKView, let context = context {
  view.context = context
  delegate = self
}
```

(5) Remove the following properties from the top of **ViewController**:

```
private var context: EAGLContext?
private var effect = GLKBaseEffect()
private var ebo    = GLuint()
private var vbo    = GLuint()
private var vao    = GLuint()
```

(6) Finally, at the top of the **ViewController** class declaration, add an outlet to a **MTKView**:

```
@IBOutlet weak var metalView: MTKView!
```

4.6 Setting-up the Storyboard

The **ViewController** is no longer **GLKViewController**, so you need to make some changes in the storyboard.

Open **Main.storyboard**. In this example, the storyboard contains two scenes, both named View Controller Scene. One has a **GLKView**, and the other one contains a **MTKView** and a connection to the outlet that you've just added to the source code.

All you need to do is set the scene with the **MTKView** as the initial View Controller. Find the scene which doesn't currently have the arrow pointing to it. Click on the bar at the top to select the view controller. Alternatively you can select it in the document outline pane. Then open the attributes inspector and check Is Initial View Controller.

Once that's done, you can delete the first scene.

4.7 Adding Metal

Are you ready? It's time to use some Metal!

In Metal, the main object that you'll use to access the GPU is **MTLDevice**. The next most important object is **MTLCommandQueue**. This object is a queue to which you'll pass encoded frames.

Open **ViewController.swift** and add these properties:

```
1  private var metalDevice: MTLDevice!
2  private var metalCommandQueue: MTLCommandQueue!
```

Now, go to **setupMetal()**. Replace the contents of it with the following:

```
1  metalDevice        = MTLCreateSystemDefaultDevice()
2  metalCommandQueue  = metalDevice.makeCommandQueue()
3  metalView.device   = metalDevice
4  metalView.delegate = self
```

That's a lot shorter than what was there before right!

This grabs the system default Metal device, then makes a command queue from the device. Then it assigns the device to the Metal view. Finally it sets the view controller as the view's delegate to receive callbacks when to draw and resize.

Now you need to to implement the **MTKViewDelegate** protocol.

At the bottom of **ViewController.swift**, add this extension:

```
1  extension ViewController: MTKViewDelegate {
2
3    func mtkView(_ view: MTKView, drawableSizeWillChange size: CGSize) {
4    }
5
```

```
6
7   func draw(in view: MTKView) {                                    2
8   }
9 }
```

This extension implements two methods.

This method is called when the drawable size changes, such as when the screen rotates. This method is called to perform the actual drawing.

4.8 Basic Drawing

You're all set to draw! To keep things simple, you'll draw a gray background first.

For each frame you want to draw in Metal, you must create a command buffer in which you specify what and how you want to draw. Then, this buffer is encoded on the CPU and sent to the GPU through the command queue.

Add the following code inside **draw(in:)**:

```
1
2  guard let drawable = view.currentDrawable else {                  1
3    return
4  }
5
6  let renderPassDescriptor = MTLRenderPassDescriptor()              2
7  renderPassDescriptor.colorAttachments[0].texture = drawable.texture
8  renderPassDescriptor.colorAttachments[0].loadAction = .clear
9  renderPassDescriptor.colorAttachments[0]
10   .clearColor = MTLClearColor(red: 0.65, green: 0.65, blue: 0.65, alpha: 1.0)
11
12
13 guard let commandBuffer = metalCommandQueue.makeCommandBuffer() else {   3
14   return
15 }
16
17
18 guard let renderEncoder = commandBuffer                           4
19   .makeRenderCommandEncoder(descriptor: renderPassDescriptor) else {
20     return
21 }
22
23 // Frame drawing goes here                                         5
24 renderEncoder.endEncoding()
25
26 commandBuffer.present(drawable)                                   6
27 commandBuffer.commit()
```

This is a big one. Here's what is going on in the code above:

Ensure there's a valid drawable to be used for the current frame. **MTLRenderPassDescriptor** contains a collection of attachments that

are the rendering destination for pixels generated by a rendering pass. Set the texture from the view as a destination for drawing. Clear every pixel at the start of a rendering. Specify the color to use when the color attachment is cleared. In this case, it's a lovely 65% gray. Ask the command queue to create a new command buffer. Create an encoder object that can encode graphics rendering commands into the command buffer. You'll add your actual drawing code after this statement later. Declare that all command generations from this encoder are complete. Register a drawable presentation to occur as soon as possible. Commit this command buffer for execution in the command queue. In summary, you create a command buffer and a command encoder. Then, you do your drawing in the command encoder and commit the command buffer to the GPU through the command queue. These steps are then repeated each time the frame is drawn.

Figure 4.2: If you're using the simulator, at this point you'll get a build error. You need a compatible device to build and run the app, hence, build and run the application.

4.8.1 Build and Running Application

The output will be a clear color (i.e., a Metal grey color). This might not be much, but this shows implementation is running. If you want to experiment, try changing the **MTLClearColor** values and rerunning the program.

4.9 Drawing Primitives

It will take some time to draw something more meaningful. So, take a deep breath and dive right in!

To draw something, you must pass data that represents the object to the GPU. You already have Vertex structure to represent the vertices data, but you need to make a small change to use it with Metal.

Open **ViewController.swift** and change the Indices property from this:

```
1  var Indices: [GLubyte]
```

To this:

```
1  var Indices: [UInt32]
```

You'll see why this change is required when drawing primitives.

4.10 Data Buffers

To pass vertices data to the GPU, you need to create two buffers: one for vertices and one for indices. This is similar to OpenGL's element buffer object (EBO) and vertex buffer object (VBO).

Add these properties to ViewController:

```
1  private var vertexBuffer:  MTLBuffer!
2  private var indicesBuffer: MTLBuffer!
```

Now, inside **setupMetal()**, add the following at the bottom:

```
1  let vertexBufferSize = Vertices.size()                                    (1)
2  vertexBuffer = metalDevice
3    .makeBuffer(bytes: &Vertices, length: vertexBufferSize, options:
           .storageModeShared)
4
5  let indicesBufferSize = Indices.size()                                    (2)
6  indicesBuffer = metalDevice
7    .makeBuffer(bytes: &Indices, length: indicesBufferSize, options:
           .storageModeShared)
```

This asks **metalDevice** to create the vertices and indices buffers initialized with your data.

Now, go to **draw(in:)**, and right before:

```
1  renderEncoder.endEncoding()
```

Add the following:

```
1  renderEncoder.setVertexBuffer(vertexBuffer, offset: 0, index: 0)
2  renderEncoder.drawIndexedPrimitives(
3    type:        .triangle,
4    indexCount:  Indices.count,
5    indexType:   .uint32,
6    indexBuffer: indicesBuffer,
7    indexBufferOffset: 0)
```

First this passes the vertex buffer to the GPU, setting it at index 0. Then, it draws triangles using **indicesBuffer**. Note that you need to specify the index type which is **uint32**. That's why you changed the indices type earlier.

4.11 Building and Running the Application

Crash! That's not good. You passed data from the CPU to the GPU. It crashed because you didn't specify how the GPU should use this data. You'll need to add some shaders! Fortunately, that's the next step.

4.12 Adding Shaders

Create a new file. Click File > New > File.., choose iOS > Source > Metal File. Click Next. Name it **Shaders.metal** and save it wherever you want.

Metal uses C-like language for the shaders. In most cases, it's similar to the GLSL used for OpenGL.

For more on shaders, check the references at the bottom of this chapter.

Add this to the bottom of the file:

```
1  struct VertexIn {                                    1
2      packed_float3 position;
3      packed_float4 color;
4  };
5
6  struct VertexOut {                                   2
7      float4 computedPosition [[position]];
8      float4 color;
9  };
```

You'll use these structures as input and output data for the vertex shader.

4.12.1 Writing a Vertex Shader

A vertex shader is a function that runs for each vertex that you draw.

Below the structs, add the following code:

```
1  vertex VertexOut basic_vertex(
2    const device VertexIn* vertex_array [[ buffer(0) ]],
3    unsigned int vid [[ vertex_id ]]) {
4
5      VertexIn v = vertex_array[vid];                  1
6
7
8      VertexOut outVertex = VertexOut();               2
9      outVertex.computedPosition = float4(v.position, 1.0);
10     outVertex.color = v.color;
11     return outVertex;
12 }
```

The vertex indicates this is a vertex shader function. The return type for this shader is **VertexOut**. Here, you get the vertex buffer that you passed to the command encoder. This parameter is a vertex id for which this shader was called. Grab the input vertex for the current vertex id. Here, you create a VertexOut and pass data from the current VertexIn. This is simply using the same position and color as the input. At this point, the vertex shader's job is done.

4.12.2 Writing a Fragment Shader

After the vertex shader finishes, the fragment shader runs for each potential pixel.

Below the vertex shader, add the following code:

```
fragment float4 basic_fragment(VertexOut interpolated [[stage_in]]) {
  return float4(interpolated.color);
}
```

The fragment shader receives the output from the vertex shader - the **VertexOut**. Then, the fragment shader returns the color for the current fragment.

4.12.3 Hooking up the Shaders to the Pipeline

The shaders are in place, but you haven't hooked them to your pipeline yet. In order to do that, go back to **ViewController.swift**, and add this property to the class:

```
private var pipelineState: MTLRenderPipelineState!
```

This property will contain shaders data.

Now, find **setupMetal()**. Add the following at the bottom of the method:

```

let defaultLibrary = metalDevice.makeDefaultLibrary()!                    1
let fragmentProgram = defaultLibrary.makeFunction(name: "basic_fragment")
let vertexProgram = defaultLibrary.makeFunction(name: "basic_vertex")


let pipelineStateDescriptor = MTLRenderPipelineDescriptor()               2
pipelineStateDescriptor.vertexFunction = vertexProgram
pipelineStateDescriptor.fragmentFunction = fragmentProgram
pipelineStateDescriptor.colorAttachments[0].pixelFormat = .bgra8Unorm


pipelineState = try! metalDevice                                          3
  .makeRenderPipelineState(descriptor: pipelineStateDescriptor)
```

This is what the code does:

Finds the vertex and fragment shaders by their name in all **.metal** files. Creates a descriptor object with the vertex shader and the fragment shader. The pixel format is set to a standard BGRA (Blue Green Red Alpha) 8-bit unsigned. Asks the GPU to compile all that into the GPU-optimized object. Now, the pipeline state is ready. It's time to use it. For this, go to draw(in:), and right before:

```
renderEncoder.drawIndexedPrimitives(
  type: .triangle,
  indexCount: Indices.count,
  indexType: .uint32,
  indexBuffer: indicesBuffer,
  indexBufferOffset: 0)
```

Add:

```
renderEncoder.setRenderPipelineState(pipelineState)
```

Build and run. You should get this colorful screen.

4.13 Gradient Background

4.13.1 Matrices

In order to manipulate the scene, you need to pass the projection and model-view matrices to the GPU. The projection matrix allows you to manipulate the perception of the scene to make closer objects appear bigger than farther ones. The model view matrix allows you to manipulate the position, rotation, and scale of an object or the whole scene.

In order to use those matrices, you'll create a new struct. Open **Vertex.swift**. At the top of the file, add:

```
struct SceneMatrices {
    var projectionMatrix: GLKMatrix4 = GLKMatrix4Identity
    var modelviewMatrix: GLKMatrix4 = GLKMatrix4Identity
}
```

Note that you'll still use **GLKMatrix4**. This part of **GLKit** is not deprecated, so you can use it for matrices in Metal.

Now, open **ViewController.swift**, and add two new properties:

```
private var sceneMatrices = SceneMatrices()
private var uniformBuffer: MTLBuffer!
```

Then, go to **draw(in:)**, and right before:

```
renderEncoder.setRenderPipelineState(pipelineState)
```

Add:

```

let modelViewMatrix = GLKMatrix4MakeTranslation(0.0, 0.0, -6.0)
sceneMatrices.modelviewMatrix = modelViewMatrix
```

```
4
5  let uniformBufferSize = MemoryLayout.size(ofValue: sceneMatrices)                2
6  uniformBuffer = metalDevice.makeBuffer(
7    bytes: &sceneMatrices,
8    length: uniformBufferSize,
9    options: .storageModeShared)
10
11 renderEncoder.setVertexBuffer(uniformBuffer, offset: 0, index: 1)               3
```

Here's what the code above does:

Creates a matrix to shift the object backwards by 6 units, to make it look smaller. Creates a uniform buffer like you did with vertex buffer before, but with matrices data. Hooks the uniform buffer to the pipeline and sets its identifier to 1.

4.13.2 Projection Matrix

While you're still in **ViewController.swift**, inside **mtkView(_:drawableSizeWillChange:)**, add the following:

```
1  let aspect = fabsf(Float(size.width) / Float(size.height))
2  let projectionMatrix = GLKMatrix4MakePerspective(
3    GLKMathDegreesToRadians(65.0),
4    aspect,
5    4.0,
6    10.0)
7  sceneMatrices.projectionMatrix = projectionMatrix
```

This code creates a projection matrix based on the aspect ratio of the view. Then, it assigns it to the scene's projection matrix.

With this in place, your square will now look square and not stretched out to the whole screen.

4.13.2.1 Matrices in Shaders

You're almost there! Next, you'll need to receive matrices data in shaders. Open **Shaders.metal**. At the very top, add a new struct:

```
1  struct SceneMatrices {
2      float4x4 projectionMatrix;
3      float4x4 viewModelMatrix;
4  };
```

Now, replace the **basic_vertex** function with the following:

```
1  vertex VertexOut basic_vertex(                                                  1
2    const device VertexIn* vertex_array [[ buffer(0) ]],
3    const device SceneMatrices& scene_matrices [[ buffer(1) ]],
4    unsigned int vid [[ vertex_id ]]) {
```

```
5
6       float4x4 viewModelMatrix = scene_matrices.viewModelMatrix;          2
7       float4x4 projectionMatrix = scene_matrices.projectionMatrix;
8
9       VertexIn v = vertex_array[vid];                                     3
10
11
12      VertexOut outVertex = VertexOut();                                  4
13      outVertex
14        .computedPosition = projectionMatrix * viewModelMatrix * float4(v.position,
              1.0);
15      outVertex.color = v.color;
16      return outVertex;
17  }
```

Here's what has changed:

Receives matrices as a parameter inside the vertex shader. Extracts the view model and projection matrices. Multiplies the position by the projection and view model matrices. Build and run the app. You should see this:

You should see a square on screen!

4.14 Making the Shape Spin

In the OpenGL implementation, **GLViewController** provided **lastUpdateDate** which would tell you when the last render was performed. In Metal, you'll have to create this yourself.

First, in **ViewController**, add a new property:

```
1   private var lastUpdateDate = Date()
```

Then, go to **draw(in:)**, and just before:

```
1   commandBuffer.present(drawable)
```

Add the following code:

```
1   commandBuffer.addCompletedHandler { _ in
2     self.lastUpdateDate = Date()
3   }
```

With this in place, when a frame drawing completes, it updates **lastUpdateDate** to the current date and time.

Now, it's time to spin! In **draw(in:)**, replace:

```
1   let modelViewMatrix = GLKMatrix4Translate(GLKMatrix4Identity, 0, 0, -6.0)
```

With:

```
var modelViewMatrix = GLKMatrix4MakeTranslation(0.0, 0.0, -6.0)                    1
let timeSinceLastUpdate = lastUpdateDate.timeIntervalSince(Date())

rotation += 90 * Float(timeSinceLastUpdate)                                        2

modelViewMatrix = GLKMatrix4Rotate(                                                3
  modelViewMatrix,
  GLKMathDegreesToRadians(rotation), 0, 0, 1)
```

This increments the rotation property by an amount proportional to the time between the last render and this render. Then, it applies a rotation around the Z-axis to the model-view matrix.

Build and run the app. You'll see the cube spinning. Success! You have a Metal spinning cube.

4.15 Where to Go From Here?

Congrats! You've learned a ton about the Metal API! Now you understand some of the most important concepts in Metal, such as shaders, devices, command buffers, and pipelines, and you have some useful insights into the differences between OpenGL and Metal.

For more, be sure to check out these great resources from Apple:

- Apple's Metal for OpenGL Developers is an interesting video for anyone with OpenGL experience
- Apple's Metal for Developers page has links to documentation, videos and sample code
- Apple's Metal Programming Guide
- Apple's Metal Shading Language Guide
- Metal videos from WWDC 2014
- Metal videos from WWDC 2015
- Metal videos from WWDC 2016
- Metal videos from WWDC 2017
- Metal videos from WWDC 2018

Metal

Bibliography

[1] Alan Chalmers and Kirsten Cater. Realistic rendering in real-time. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 21–28, London, UK, UK, 2002. Springer-Verlag.

[2] Abhishek Nandy and Debashree Chanda. Introduction to the game engine. In *Beginning Platino Game Engine*, pages 1–17. Springer, 2016.

[3] Joseph A Shiraef. An exploratory study of high performance graphics application programming interfaces. 2016.

Metal

Index