

Third Edition — Xcode 13 · Metal 2.4



# Metal by Tutorials

Beginning Game Engine Development With Metal

By the raywenderlich Tutorial Team

Caroline **Begbie** & Marius **Horga**

# Metal by Tutorials

By Caroline Begbie & Marius Horga

Copyright ©2022 Razeware LLC.

## Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

## Notice of Liability

This book and all corresponding materials (such as source code) are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

## Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.



# Table of Contents: Overview

Book License .....	17
<b>Before You Begin .....</b>	<b>18</b>
What You Need.....	19
Book Source Code & Forums .....	20
Acknowledgments.....	23
Introduction .....	24
<b>Section I: Beginning Metal.....</b>	<b>28</b>
Chapter 1: Hello, Metal! .....	29
Chapter 2: 3D Models.....	44
Chapter 3: The Rendering Pipeline .....	69
Chapter 4: The Vertex Function.....	97
Chapter 5: 3D Transformations .....	117
Chapter 6: Coordinate Spaces .....	134
Chapter 7: The Fragment Function.....	156
Chapter 8: Textures.....	179
Chapter 9: Navigating a 3D Scene .....	212
Chapter 10: Lighting Fundamentals .....	239
<b>Section II: Intermediate Metal .....</b>	<b>267</b>
Chapter 11: Maps & Materials .....	268
Chapter 12: Render Passes.....	300
Chapter 13: Shadows .....	321

Chapter 14: Deferred Rendering .....	343
Chapter 15: Tile-Based Deferred Rendering.....	371
Chapter 16: GPU Compute Programming.....	395
Chapter 17: Particle Systems.....	411
Chapter 18: Particle Behavior .....	437
<b>Section III: Advanced Metal .....</b>	<b>458</b>
Chapter 19: Tessellation & Terrains .....	459
Chapter 20: Fragment Post-Processing.....	491
Chapter 21: Imaged-Based Lighting.....	510
Chapter 22: Reflection & Refraction.....	544
Chapter 23: Animation .....	568
Chapter 24: Character Animation.....	593
Chapter 25: Managing Resources .....	619
Chapter 26: GPU-Driven Rendering.....	637
<b>Section IV: Ray Tracing.....</b>	<b>665</b>
Chapter 27: Rendering With Rays .....	666
Chapter 28: Advanced Shadows.....	694
Chapter 29: Advanced Lighting.....	722
Chapter 30: Metal Performance Shaders .....	738
Chapter 31: Performance Optimization.....	755
Chapter 32: Best Practices .....	777
Conclusion .....	788

# Table of Contents: Extended

Book License .....	17
<b>Before You Begin.....</b>	<b>18</b>
What You Need .....	19
Book Source Code & Forums .....	20
About the Authors .....	22
About the Editors .....	22
Acknowledgments .....	23
Introduction .....	24
About This Book .....	24
How Did Metal Come to Life?.....	25
Why Would You Use Metal? .....	25
When Should You Use Metal? .....	26
Who This Book Is For? .....	27
How to Read This Book?.....	27
<b>Section I: Beginning Metal .....</b>	<b>28</b>
Chapter 1: Hello, Metal! .....	29
What is Rendering? .....	30
What is a Frame?.....	32
Your First Metal App.....	32
The Metal View .....	33
Rendering.....	38
Challenge .....	42
Key Points.....	43
Chapter 2: 3D Models .....	44
What Are 3D Models? .....	45
Creating Models With Blender .....	49

3D File Formats.....	50
Exporting to Blender.....	51
The .obj File Format.....	53
The .mtl File Format.....	54
Material Groups .....	57
Vertex Descriptors.....	59
Metal Coordinate System.....	62
Submeshes.....	65
Challenge .....	67
Key Points.....	68
Chapter 3: The Rendering Pipeline .....	69
The GPU and CPU .....	70
The Metal Project.....	71
The Render Pipeline .....	81
Challenge .....	95
Key Points.....	96
Chapter 4: The Vertex Function .....	97
Shader Functions .....	98
The Starter Project .....	98
Rendering a Quad.....	99
Calculating Positions .....	102
More Efficient Rendering .....	104
Vertex Descriptors .....	107
Adding Another Vertex Attribute .....	110
Rendering Points .....	113
Challenge.....	115
Key Points .....	116
Chapter 5: 3D Transformations .....	117
Transformations.....	118
The Starter Project & Setup.....	119

Translation .....	122
Vectors & Matrices .....	122
Creating a Matrix.....	125
Scaling.....	127
Rotation .....	129
Key Points .....	133
<b>Chapter 6: Coordinate Spaces.....</b>	<b>134</b>
The Starter Project .....	139
Uniforms .....	141
Projection .....	146
Refactoring the Model Matrix.....	152
Key Points .....	155
Where to Go From Here?.....	155
<b>Chapter 7: The Fragment Function.....</b>	<b>156</b>
The Starter Project .....	157
Screen Space .....	158
Metal Standard Library Functions.....	161
Normals .....	167
Loading the Train Model With Normals .....	169
Depth.....	172
Hemispheric Lighting.....	175
Challenge.....	177
Key Points .....	178
Where to Go From Here?.....	178
<b>Chapter 8: Textures .....</b>	<b>179</b>
Textures and UV Maps .....	180
The Starter App .....	183
sRGB Color Space .....	190
Capture GPU Workload.....	192
Samplers.....	195

Mipmaps.....	200
The Asset Catalog .....	204
Texture Compression.....	209
Key Points .....	211
<b>Chapter 9: Navigating a 3D Scene.....</b>	<b>212</b>
The Starter Project.....	213
Scenes .....	213
Cameras .....	215
Input.....	219
Delta Time.....	221
Mouse and Trackpad Input .....	227
Arcball Camera.....	228
Orthographic Projection .....	232
Challenge.....	237
Key Points .....	238
<b>Chapter 10: Lighting Fundamentals.....</b>	<b>239</b>
The Starter Project.....	240
Representing Color.....	241
Normals .....	243
Light Types .....	244
Directional Light.....	244
The Phong Reflection Model.....	247
The Dot Product.....	248
Creating Shared Functions in C++ .....	252
Point Lights.....	260
Spotlights.....	263
Key Points .....	266
Where to Go From Here?.....	266
<b>Section II: Intermediate Metal.....</b>	<b>267</b>
<b>Chapter 11: Maps &amp; Materials .....</b>	<b>268</b>

Normal Maps .....	269
The Starter App .....	275
Using Normal Maps .....	275
Materials .....	285
Physically Based Rendering (PBR) .....	291
Channel Packing.....	296
Challenge.....	298
Where to Go From Here?.....	299
<b>Chapter 12: Render Passes.....</b>	<b>300</b>
Render Passes .....	301
Object Picking.....	302
The Starter App .....	303
Setting up Render Passes.....	304
Creating a UInt32 Texture.....	305
Adding the Render Pass to Renderer.....	308
Adding the Shader Function.....	309
Adding the Depth Attachment .....	312
Load & Store Actions .....	315
Reading the Object ID Texture .....	316
Key Points .....	320
<b>Chapter 13: Shadows.....</b>	<b>321</b>
Shadow Maps.....	322
The Starter Project.....	323
Identifying Problems .....	334
Visualizing the Problems .....	336
Solving the Problems .....	339
Cascaded Shadow Mapping .....	341
Key Points .....	342
<b>Chapter 14: Deferred Rendering.....</b>	<b>343</b>
The Starter Project .....	346

The G-buffer Pass .....	347
Updating Renderer .....	357
The Lighting Shader Functions .....	358
Adding Point Lights.....	360
Blending .....	367
Key Points .....	370
<b>Chapter 15: Tile-Based Deferred Rendering.....</b>	<b>371</b>
Programmable Blending.....	373
Tiled Deferred Rendering .....	373
The Starter Project .....	374
Stencil Tests .....	384
Create the Stencil Texture.....	387
Configure the Stencil Operation .....	388
Masking the Sky .....	392
Challenge.....	394
Key Points .....	394
Where to Go From Here?.....	394
<b>Chapter 16: GPU Compute Programming.....</b>	<b>395</b>
The Starter Project.....	396
Winding Order and Culling.....	397
Reversing the Model on the CPU.....	398
Compute Processing .....	400
Reversing the Warrior Using GPU Compute Processing.....	404
The Kernel Function.....	406
Atomic Functions.....	408
Key Points .....	410
<b>Chapter 17: Particle Systems.....</b>	<b>411</b>
Particle .....	412
Emitter .....	413
The Starter Project .....	413

Creating a Particle and Emitter .....	414
The Fireworks Pass.....	417
Particle Dynamics .....	421
Implementing Particle Physics .....	422
Particle Systems.....	423
Rendering a Particle System .....	426
Configuring Particle Effects.....	432
Fire .....	434
Key Points .....	436
Where to Go From Here?.....	436
<b>Chapter 18: Particle Behavior.....</b>	<b>437</b>
Behavioral Animation .....	438
Swarming Behavior.....	439
The Starter Project.....	440
Velocity.....	443
Behavioral Rules.....	445
Key Points .....	457
Where to Go From Here?.....	457
<b>Section III: Advanced Metal .....</b>	<b>458</b>
<b>Chapter 19: Tessellation &amp; Terrains .....</b>	<b>459</b>
Tessellation.....	460
The Starter Project .....	460
The Tessellation Kernel.....	465
Multiple Patches .....	472
Tessellation By Distance .....	474
Displacement.....	479
Shading By Slope .....	485
Challenge.....	488
Key Points .....	489
Where to Go From Here?.....	490

<b>Chapter 20: Fragment Post-Processing .....</b>	<b>491</b>
The Starter App .....	492
Alpha Testing .....	493
Depth Testing .....	495
Stencil Testing .....	495
Scissor Testing .....	496
Alpha Blending .....	497
Opacity .....	498
Blending .....	499
Antialiasing .....	505
Fog .....	507
Key Points .....	509
Where to Go From Here? .....	509
<b>Chapter 21: Imaged-Based Lighting .....</b>	<b>510</b>
The Starter Project .....	511
The Skybox .....	511
Procedural Skies .....	518
Reflection .....	525
Image-Based Lighting .....	528
Challenge .....	542
Key Points .....	543
Where to Go From Here? .....	543
<b>Chapter 22: Reflection &amp; Refraction .....</b>	<b>544</b>
The Starter Project .....	545
Rendering Rippling Water .....	546
1. Creating the Water Surface .....	547
2. Rendering the Reflection .....	549
3. Creating Clipping Planes .....	555
4. Rippling Normal Maps .....	557
5. Adding Refraction .....	560
6. The Fresnel Effect .....	563

7. Adding Smoothness Using a Depth Texture .....	565
Key Points .....	567
Where to Go From Here?.....	567
Chapter 23: Animation .....	568
The Starter Project.....	569
Animation .....	569
Procedural Animation.....	570
Animation Using Physics .....	571
Keyframes .....	574
Quaternions .....	581
USD and USDZ Files.....	585
Animating Meshes.....	586
Challenge.....	592
Key Points .....	592
Chapter 24: Character Animation.....	593
Skeletal Animation .....	594
The Starter App .....	600
Implementing Skeletal Animation.....	601
Loading the Animation .....	604
The Joint Matrix Palette.....	605
The Inverse Bind Matrix.....	607
Updating the Vertex Shader.....	611
Function Specialization.....	613
Key Points .....	617
Where to Go From Here?.....	618
Chapter 25: Managing Resources .....	619
The Starter Project.....	620
Argument Buffers .....	622
Resource Heaps .....	629
Key Points .....	636

<b>Chapter 26: GPU-Driven Rendering .....</b>	<b>637</b>
The Starter Project .....	638
Indirect Command Buffers .....	639
GPU-Driven Rendering.....	651
Challenge.....	663
Key Points .....	663
Where to Go From Here?.....	664
<b>Section IV: Ray Tracing.....</b>	<b>665</b>
<b>Chapter 27: Rendering With Rays.....</b>	<b>666</b>
Getting Started.....	667
Ray Casting.....	668
Ray Tracing .....	670
Path Tracing .....	671
Raymarching .....	673
Signed Distance Functions .....	675
The Starter Playground .....	676
Using a Signed Distance Function.....	676
The Raymarching Algorithm .....	677
Creating Random Noise .....	684
Marching Clouds .....	691
Key Points .....	693
<b>Chapter 28: Advanced Shadows .....</b>	<b>694</b>
The Starter Playground .....	695
Hard Shadows.....	695
Soft Shadows .....	703
Ambient Occlusion .....	712
Key Points .....	721
Where to Go From Here?.....	721
<b>Chapter 29: Advanced Lighting .....</b>	<b>722</b>
The Rendering Equation.....	723

Reflection .....	724
Getting Started.....	725
Refraction .....	731
Raytraced Water .....	735
Key Points .....	737
Where to Go From Here?.....	737
<b>Chapter 30: Metal Performance Shaders .....</b>	<b>738</b>
Overview .....	739
The Sobel Filter.....	739
Image Processing.....	740
The Starter Project.....	744
The Blit Command Encoder .....	746
Gaussian Blur.....	747
Matrix / Vector Mathematics .....	750
Challenge.....	754
Key Points .....	754
<b>Chapter 31: Performance Optimization.....</b>	<b>755</b>
The Starter App .....	756
Profiling .....	756
GPU Workload Capture.....	758
GPU Timeline.....	764
Instancing .....	768
Removing Duplicate Textures .....	770
CPU-GPU Synchronization.....	771
Key Points .....	776
<b>Chapter 32: Best Practices .....</b>	<b>777</b>
General Performance Best Practices.....	778
Memory Bandwidth Best Practices .....	780
Memory Footprint Best Practices.....	784
Where to Go From Here?.....	787

Conclusion.....	788
-----------------	-----



# Book License

By purchasing *Metal by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *Metal by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Metal by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: “Artwork/images/designs: from *Metal by Tutorials*, available at [www.raywenderlich.com](http://www.raywenderlich.com)”.
- The source code included in *Metal by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Metal by Tutorials* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

# Before You Begin

This section tells you a few things you need to know before you get started, such as what you'll need for hardware and software, where to find the project files for this book and more.





# What You Need

To follow along with the tutorials in this book, you need the following:

- A **Metal-capable Mac running macOS Monterey 12.0 or later.** All Macs built since 2012 should run Metal, although not all of them will be able to run the most recent features in Metal 2. Nvidia GPUs will have issues, in some cases serious, as drivers have not been updated since macOS High Sierra. One chapter includes Apple GPU-specific code, for which you'll need either an M1 GPU or a recent iPhone or iPad.
- **Xcode 13.3 or later.**
- [optional] A **Metal-capable iPhone or iPad running iOS 15 or later.** Any iOS device running the A7 chip or later will run Metal. The latest features, such as tile shading and imageblocks, will only run on the A11 (or later) chipset. The projects will build and run on macOS, and most of them will run on the iOS Simulator, so using an iOS device is optional. If you wish to make an iOS game, the game engine you build while reading this book will have an iOS target as well.





# Book Source Code & Forums

## Where to download the materials for this book

The materials for this book can be cloned or downloaded from the GitHub book materials repository:

- <https://github.com/raywenderlich/met-materials/tree/editions/3.0>

## Forums

We've also set up an official forum for the book at <https://forums.raywenderlich.com/c/books/metal-by-tutorials>. This is a great place to ask questions about the book or to submit any errors you may find.



“To Warren Moore, who first made it possible for me to learn Metal, to my wonderful children Robin and Kayla, and to my best friends who patiently waited for me to indulge my dream.”

— *Caroline Beagie*

“To my wife, Adina, and my son, Victor Nicholas, without whose patience, support and understanding I could not have made it. To Warren Moore who first whet my appetite for Metal, offered his advice when needed and motivated me to get involved with Metal too. To Chris Wood who taught me that most of the times all you need to render is a ray, a camera and a few distance fields. To Simon Gladman whose amazing work with compute kernels inspired me to write more about particles and fluid dynamics. To Jeff Biggus who keeps the GPU programming community in Chicago alive. Our daily conversations motivate me to stay hungry for more. To everyone else who believes in me. A huge Thanks to all of you!”

— *Marius Horga*

## About the Authors



**Caroline Begbie** is a co-author of this book. Caroline is an indie iOS developer. When she's not developing, she's playing around with 2D and 3D animation software, or planning The Big Lap around Australia. She has previously taught the elderly how to use their computers, done marionette shows for pre-schools, and created accounting and stock control systems for mining companies.



**Marius Horga** is a co-author of this book. Marius is an iOS developer and Metal API blogger. He is also a computer scientist. He has more than a decade of experience with systems, support, integration and development. You can often see him on Twitter talking about Metal, GPGPU, games and 3D graphics. When he's away from computers, he enjoys music, biking or stargazing.

## About the Editors



**Adrian Strahan** is the technical editor of this book. Adrian is a lead iOS developer working for a leading UK bank. When he's not sat in front of a computer, he enjoys watching sport, listening to music and trying to keep fit and healthy.



**Tammy Coron** is the final pass editor of this book. Tammy is an independent creative professional, author of Apple Game Frameworks and Technologies, and the maker behind the AdventureGameKit — a custom SpriteKit framework for building point and click adventure games. Find out more at [tammycoron.com](http://tammycoron.com).

# Acknowledgments

Many of the models and images used in this book were made by the authors and the raywenderlich.com team. In some cases, the authors used public domain or CC By 4.0, commercial allowed models and images and included links and licences in either the projects' **Models** folder or **references.markdown** for the chapter.





# Introduction

Welcome to *Metal by Tutorials!*

Metal is a unified, low-level, low-overhead application programming interface (API) for the graphics processing unit, or GPU. It's unified because it applies to both 3D graphics and data-parallel computation paradigms. Metal is a low-level API because it provides programmers near-direct access to the GPU. Finally, Metal is a low-overhead API because it reduces the runtime cost by multi-threading and pre-compiling of resources.

But beyond the technical definition, Metal is the most appropriate way to use the GPU's parallel processing power to visualize data or solve numerical challenges. It's also tailored to be used for machine learning, image/video processing or, as this book describes, graphics rendering.

## About This Book

This book introduces you to low-level graphics programming in Metal — Apple's framework for programming on the graphics processing unit (GPU). As you progress through this book, you'll learn many of the fundamentals that go into making a game engine and gradually put together your own engine. Once your game engine is complete, you'll be able to put together 3D scenes and program your own simple 3D games. Because you'll have built your 3D game engine from scratch, you'll be able to customize every aspect of what you see on your screen.



## How Did Metal Come to Life?

Historically, you had two choices to take advantage of the power of the GPU: OpenGL and the Windows-only DirectX. In 2013, the GPU vendor AMD announced the Mantle project in an effort to revamp GPU APIs and come up with an alternative to Direct3D (which is part of DirectX) and OpenGL. AMD were the first to create a true low-overhead API for low-level access to the GPU. Mantle promised to be able to generate up to 9 times more draw calls (the number of objects drawn to the screen) than similar APIs and also introduced asynchronous command queues so that graphics and compute workloads could be run in parallel. Unfortunately, the project was terminated before it could become a mainstream API.

Metal was announced at the Worldwide Developers Conference (WWDC) on June 2, 2014 and was initially made available only on A7 or newer GPUs. Apple created a new language to program the GPU directly via shader functions. This is the **Metal Shading Language** (MSL) based on the C++11 specification. A year later at WWDC 2015, Apple announced two Metal sub-frameworks: **MetalKit** and **Metal Performance Shaders** (MPS). In 2018, MPS made a spectacular debut as a Ray Tracing accelerator.

The API continues to evolve to work with the exciting features of the new Apple GPUs designed in-house by Apple. **Metal 2** adds support for Virtual Reality (VR), Augmented Reality (AR) and accelerated machine learning (ML), among many new features, including image blocks, tile shading and threadgroup sharing. MSL was also updated to version 2.0 in Fall 2017 and is now based on the C++14 specification.

## Why Would You Use Metal?

Metal is a top-notch graphics API. That means Metal can empower graphics pipelines and, more specifically, game engines such as the following:

- **Unity and Unreal Engine:** The two leading cross-platform game engines today are ideal for game programmers who target a range of console, desktop and mobile devices. However, these engines haven't always kept pace with new features in Metal. For example, Unity announced that tessellation on iOS was to be released in 2018, despite it being demonstrated live at WWDC 2016. If you to use cutting-edge Metal developments, you can't always depend on third-party engines.

- **Divinity - Original Sin 2:** Larian Studios worked closely with Apple to bring their amazing AAA game to iPad, taking advantage of Metal and the Apple GPU hardware. It truly is a stunning visual experience.
- **The Witness:** This award-winning puzzle game has a custom engine that runs on top of Metal. By taking advantage of Metal, the iPad version is every bit as stunning as the desktop version and is highly recommended for puzzle game fans.
- **Many Others:** From notable game titles such as *Hitman*, *BioShock*, *Deus Ex*, *Mafia*, *Starcraft*, *World of Warcraft*, *Fortnite*, *Unreal Tournament*, *Batman* and even the beloved *Minecraft*.

But Metal isn't limited to the world of gaming. There are many apps that benefit from GPU acceleration for image and video processing:

- **Procreate:** An app for sketching, painting and illustrating. Since converting to Metal, it runs four times faster than it did before.
- **Pixelmator:** A Metal-based app that provides image distortion tools. In fact, they were able to implement a new painting engine and dynamic paint blending technology powered by Metal 2.
- **Affinity Photo:** Available on the iPad. According to the developer Serif, "Using Metal allows users to work easily on large, super high-resolution photographs, or complex compositions with potentially thousands of layers."

Metal, and in particular, the MPS sub-framework, is incredibly useful in the realm of machine and deep learning on convolutional neural networks. Apple presented a practical machine learning application at WWDC 2016 that demonstrated the power of CNNs in high-precision image recognition.

## When Should You Use Metal?

GPUs belong to a special class of computation that Flynn's taxonomy terms Single Instruction Multiple Data (SIMD). Simply, GPUs are processors that are optimized for throughput (how much data can be processed in one unit of time), while CPUs are optimized for latency (how much time it takes a single unit of data to be processed). Most programs execute serially: they receive input, process it, provide output and then the cycle repeats.

Those cycles sometimes perform computationally-intensive tasks, such as large matrix multiplication, which would take CPUs a lot of time process serially, even in a multithreaded manner on a handful of cores.

In contrast, GPUs have hundreds or even thousands of cores which are smaller and have less memory than CPU cores, but perform fast parallel mathematical calculations.

Choose Metal when:

- You want to render 3D models as efficiently as possible.
- You want your game to have its own unique style, perhaps with custom lighting and shading.
- You will be performing intensive data processes, such as calculating and changing the color of each pixel on the screen every frame, as you would when processing images and video.
- You have large numerical problems, such as scientific simulations, that you can partition into independent sub-problems to be processed in parallel.
- You need to process multiple large datasets in parallel, such as when you train models for deep learning.

## Who This Book Is For?

This book is for intermediate Swift developers interested in learning 3D graphics or gaining a deeper understanding of how game engines work. If you don't know Swift, you can still follow along, as all the code instructions are included in the book. You'll gain general graphics knowledge, but it would be less confusing if you cover Swift basics first. We recommend the *Swift Apprentice* book, available in our catalogue:

<https://www.raywenderlich.com/books/swift-apprentice>

A smattering of C++ knowledge would be useful too. The Metal Shader Language that you'll use when writing GPU shader functions is based on C++. But, again, all the code you'll need is included in the book.

## How to Read This Book?

If you're a beginner to iOS/macOS development or Metal, you should read this book from cover to cover.

If you're an advanced developer, or already have experience with Metal, you can skip from chapter to chapter or use this book as a reference.

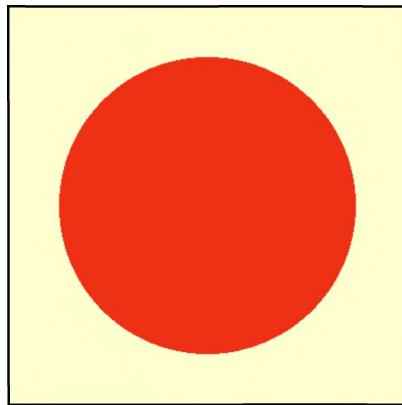
# Section I: Beginning Metal

It takes a wealth of knowledge to render a simple triangle on the screen or animate game characters. This section will guide you through the necessary basics of vertex wrangling, lighting, textures and creating a game scene. If you're worried about the math, don't be! Although computer graphics is highly math-intensive, each chapter explains everything you need, and you'll get experience creating and rendering models.



# Chapter 1: Hello, Metal!

You've been formally introduced to Metal and discovered its history and why you should use it. Now you're going to try it out for yourself in a Swift playground. To get started, you'll render this sphere on the screen:



*The final result*

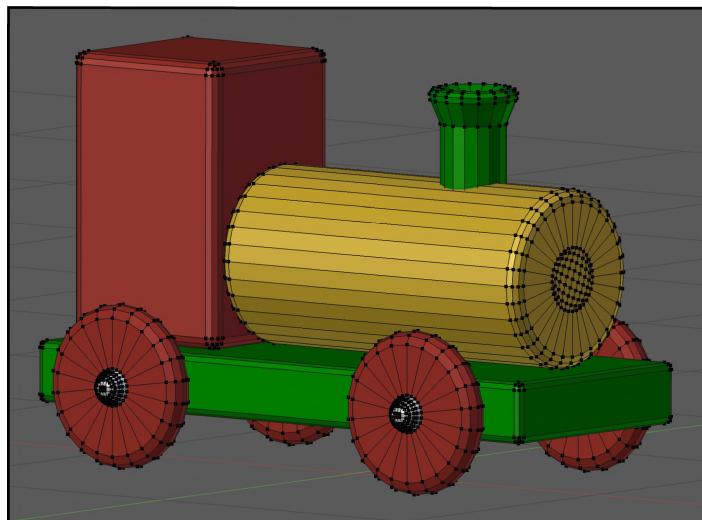
It may not look exciting, but this is a great starting point because it lets you touch on almost every part of the rendering process. But before you get started, it's important to understand the terms **rendering** and **frames**.

# What is Rendering?

In 3D computer graphics, you take a bunch of points, join them together and create an image on the screen. This image is known as a render.

Rendering an image from points involves calculating light and shade for each pixel on the screen. Light bounces around a scene, so you have to decide how complicated your lighting is and how long each image takes to render. A single image in a Pixar movie might take days to render, but games require real-time rendering, where you see the image immediately.

There are many ways to render a 3D image, but most start with a model built in a modeling app such as Blender or Maya. Take, for example, this train model that was built in Blender:



A train model in Blender

This model, like all other models, is made up of **vertices**. A vertex refers to a point in three dimensional space where two or more lines, curves or edges of a geometrical shape meet, such as the corners of a cube. The number of vertices in a model may vary from a handful, as in a cube, to thousands or even millions in more complex models. A 3D renderer will read in these vertices using *model loader code*, which parses the list of vertices. The renderer then passes the vertices to the GPU, where shader functions process the vertices to create the final image or texture to be sent back to the CPU and displayed on the screen.

The following render uses the 3D train model and some different shading techniques to make it appear as if the train were made of shiny copper:



*Shading techniques cause reflection*

The entire process, from importing a model's vertices to generating the final image on your screen, is commonly known as the *rendering pipeline*. The rendering pipeline is a list of commands sent to the GPU, along with resources (vertices, materials and lights) that make up the final image.

The pipeline includes programmable and non-programmable functions. The programmable parts of the pipeline, known as **vertex functions** and **fragment functions**, are where you can manually influence the final look of your rendered models. You'll learn more about each later in the book.

## What is a Frame?

A game wouldn't be much fun if all it did was render a single still image. Moving a character around the screen in a fluid manner requires the GPU to render a still image roughly sixty times a second. Each still image is known as a **frame**, and the speed at which the images appear is known as the **frame rate**.

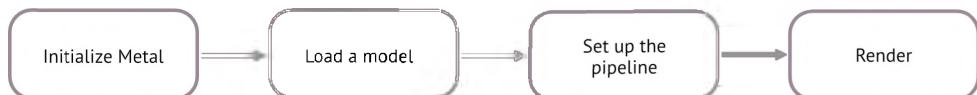
When your favorite game appears to stutter, it's usually because of a decrease in the frame rate, especially if there's an excessive amount of background processing eating away at the GPU. When designing a game, it's important to balance the result you want with what the hardware can deliver.

While it might be cool to add real-time shadows, water reflections and millions of blades of animated grass — all of which you'll learn how to do in this book — finding the right balance between what is possible and what the GPU can process in 1/60th of a second can be tough.

## Your First Metal App

In your first Metal app, the shape you'll render will look more like a flat circle than a 3D sphere. That's because your first model will not include any perspective or shading. However, its vertex mesh contains the full three-dimensional information.

The process of Metal rendering is much the same no matter the size and complexity of your app, and you'll become very familiar with the following sequence of drawing your models on the screen:

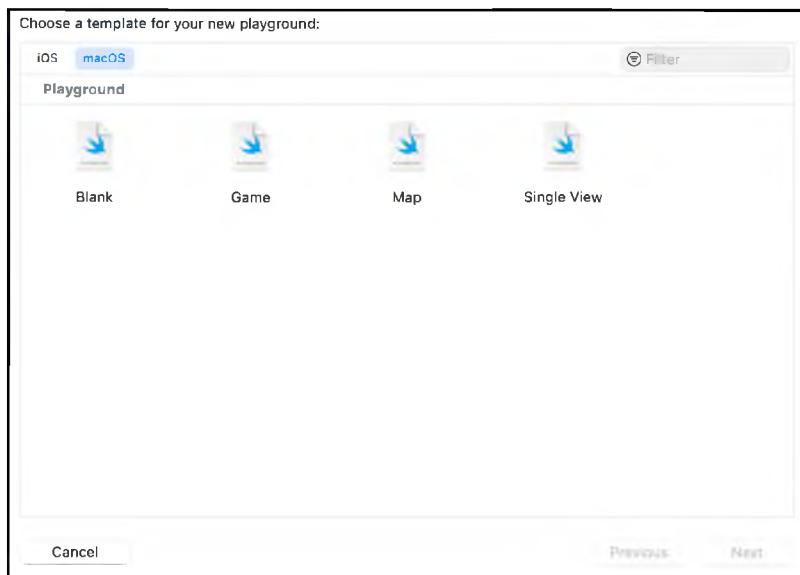


You may initially feel a little overwhelmed by the number of steps Metal requires, but don't worry. You'll always perform these steps in the same sequence, and they'll gradually become second nature.

This chapter won't go into detail on every step, but as you progress through the book, you'll get more information as you need it. For now, concentrate on getting your first Metal app running.

## Getting Started

- Start Xcode, and create a new playground by selecting **File ▶ New ▶ Playground...** from the main menu. When prompted for a template, choose **macOS Blank**.



*The playground template*

- Name the playground **Chapter1**, and click **Create**.
- Next, delete everything in the playground.

## The Metal View

Now that you have a playground, you'll create a view to render into.

- Import the two main frameworks that you'll be using by adding this:

```
import PlaygroundSupport
import MetalKit
```

**PlaygroundSupport** lets you see live views in the assistant editor, and **MetalKit** is a framework that makes using Metal easier. **MetalKit** has a customized view named **MTKView** and many convenience methods for loading textures, working with Metal buffers and interfacing with another useful framework: **Model I/O**, which you'll learn about later.

► Now, add this:

```
guard let device = MTLCreateSystemDefaultDevice() else {
    fatalError("GPU is not supported")
}
```

This code checks for a suitable GPU by creating a `device`:

**Note:** Are you getting an error? If you accidentally created an iOS playground instead of a macOS playground, you'll get a fatal error because the iOS simulator is not supported.

► To set up the view, add this:

```
let frame = CGRect(x: 0, y: 0, width: 600, height: 600)
let view = MTKView(frame: frame, device: device)
view.clearColor = MTLClearColor(red: 1,
    green: 1, blue: 0.8, alpha: 1)
```

This code configures an `MTKView` for the Metal renderer. `MTKView` is a subclass of `NSView` on macOS and of `UIView` on iOS. `MTLClearColor` represents an RGBA value — in this case, cream. The color value is stored in `clearColor` and is used to set the color of the view.

## The Model

Model I/O is a framework that integrates with Metal and SceneKit. Its main purpose is to load 3D models that were created in apps like Blender or Maya, and to set up data buffers for easier rendering. Instead of loading a 3D model, you're going to load a Model I/O basic 3D shape, also called a **primitive**. A primitive is typically considered a cube, a sphere, a cylinder or a torus.

► Add this code to the end of the playground:

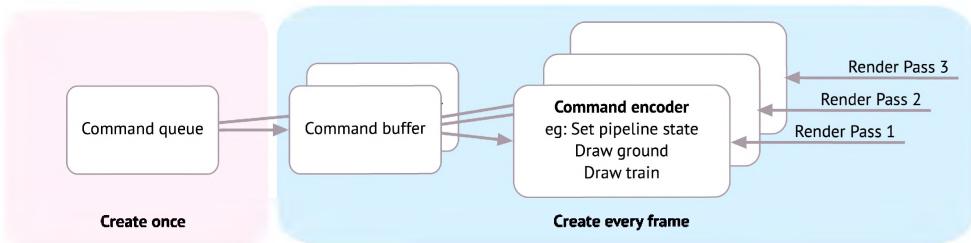
```
// 1
let allocator = MTKMeshBufferAllocator(device: device)
// 2
let mdlMesh = MDLMesh(sphereWithExtent: [0.75, 0.75, 0.75],
    segments: [100, 100],
    inwardNormals: false,
    geometryType: .triangles,
    allocator: allocator)
// 3
let mesh = try MTKMesh(mesh: mdlMesh, device: device)
```

Going through the code:

1. The allocator manages the memory for the mesh data.
2. Model I/O creates a sphere with the specified size and returns an `MDLMesh` with all the vertex information in data buffers.
3. For Metal to be able to use the mesh, you convert it from a Model I/O mesh to a MetalKit mesh.

## Queues, Buffers and Encoders

Each frame consists of commands that you send to the GPU. You wrap up these commands in a **render command encoder**. Command buffers organize these command encoders and a command queue organizes the command buffers.



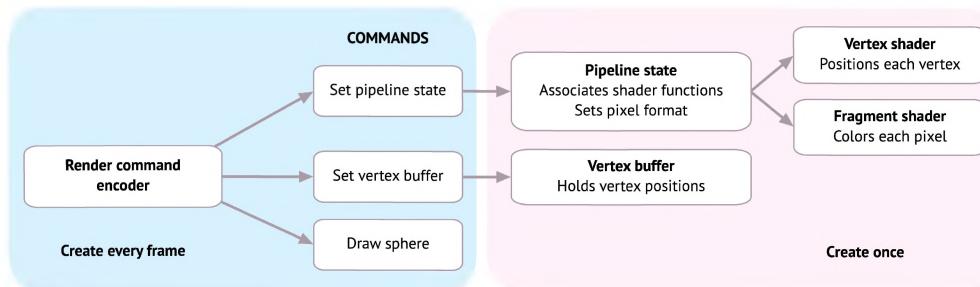
► Add this code to create a command queue:

```
guard let commandQueue = device.makeCommandQueue() else {  
    fatalError("Could not create a command queue")  
}
```

You should set up the device and the command queue at the start of your app, and generally, you should use the same device and command queue throughout.



On each frame, you'll create a command buffer and at least one render command encoder. These are lightweight objects that point to other objects, such as shader functions and pipeline states, that you set up only once at the start of the app.



## Shader Functions

Shader functions are small programs that run on the GPU. You write these programs in the Metal Shading Language, which is a subset of C++. Normally, you'd create a separate file with a `.metal` extension specifically for shader functions but for now, create a multi-line string containing the shader function code, and add it to your playground:

```

let shader = """
#include <metal_stdlib>
using namespace metal;

struct VertexIn {
    float4 position [[attribute(0)]];
};

vertex float4 vertex_main(const VertexIn vertex_in [[stage_in]])
{
    return vertex_in.position;
}

fragment float4 fragment_main() {
    return float4(1, 0, 0, 1);
}
"""
  
```

There are two shader functions in here: a vertex function named `vertex_main` and a fragment function named `fragment_main`. The vertex function is where you usually manipulate vertex positions and the fragment function is where you specify the pixel color.

To set up a Metal library containing these two functions, add the following:

```
let library = try device.makeLibrary(source: shader, options: nil)
let vertexFunction = library.makeFunction(name: "vertex_main")
let fragmentFunction = library.makeFunction(name: "fragment_main")
```

The compiler will check that these functions exist and make them available to a pipeline descriptor.

## The Pipeline State

In Metal, you set up a **pipeline state** for the GPU. By setting up this state, you're telling the GPU that nothing will change until the state changes. With the GPU in a fixed state, it can run more efficiently. The pipeline state contains all sorts of information that the GPU needs, such as which pixel format it should use and whether it should render with depth. The pipeline state also holds the vertex and fragment functions that you just created.

However, you don't create a pipeline state directly, rather you create it through a descriptor. This descriptor holds everything the pipeline needs to know, and you only change the necessary properties for your particular rendering situation.

► Add this code:

```
let pipelineDescriptor = MTLRenderPipelineDescriptor()
pipelineDescriptor.colorAttachments[0].pixelFormat = .bgra8Unorm
pipelineDescriptor.vertexFunction = vertexFunction
pipelineDescriptor.fragmentFunction = fragmentFunction
```

Here, you've specified the pixel format to be 32 bits with color pixel order of blue/green/red/alpha. You also set the two shader functions.



You'll describe to the GPU how the vertices are laid out in memory using a **vertex descriptor**. Model I/O automatically creates a vertex descriptor when it loads the sphere mesh, so you can just use that one.

► Add this code:

```
pipelineDescriptor.vertexDescriptor =  
    MTKMetalVertexDescriptorFromModelIO(mesh.vertexDescriptor)
```

You've now set up the pipeline descriptor with the necessary information. `MTLRenderPipelineDescriptor` has many other properties, but for now, you'll use the defaults.

► Now, add this code:

```
let pipelineState =  
    try device.makeRenderPipelineState(descriptor:  
        pipelineDescriptor)
```

This code creates the pipeline state from the descriptor. Creating a pipeline state takes valuable processing time, so all of the above should be a one-time setup. In a real app, you might create several pipeline states to call different shading functions or use different vertex layouts.

## Rendering

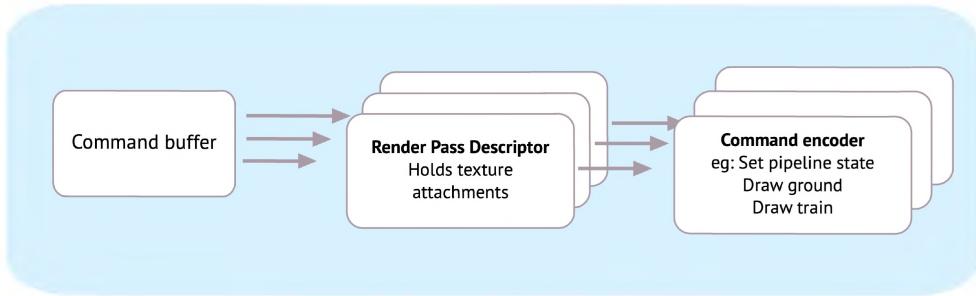
From now on, the code should be performed every frame. `MTKView` has a delegate method that runs every frame, but as you're doing a simple render which will simply fill out a static view, you don't need to keep refreshing the screen every frame.

When performing graphics rendering, the GPU's ultimate job is to output a single texture from a 3d scene. This texture is similar to the digital image created by a physical camera. The texture will be displayed on the device's screen each frame.

## Render Passes

If you're trying to achieve a realistic render, you'll want to take into account shadows, lighting and reflections. Each of these takes a lot of calculation and is generally done in separate **render passes**. For example, a shadow render pass will render the entire scene of 3D models, but only retain grayscale shadow information.

A second render pass would render the models in full color. You can then combine the shadow and color textures to produce the final output texture that will go to the screen.



For the first part of this book, you'll use a single render pass. Later, you'll learn about multipass rendering.

Conveniently, MTKView provides a render pass descriptor that will hold a texture called the **drawable**.

► Add this code to the end of the playground:

```
// 1
guard let commandBuffer = commandQueue.makeCommandBuffer(),
// 2
    let renderPassDescriptor = view.currentRenderPassDescriptor,
// 3
    let renderEncoder = commandBuffer.makeRenderCommandEncoder(
        descriptor: renderPassDescriptor)
else { fatalError() }
```

Here's what's happening:

1. You create a command buffer. This stores all the commands that you'll ask the GPU to run.
2. You obtain a reference to the view's render pass descriptor. The descriptor holds data for the render destinations, known as **attachments**. Each attachment needs information, such as a texture to store to, and whether to keep the texture throughout the render pass. The render pass descriptor is used to create the render command encoder.
3. From the command buffer, you get a render command encoder using the render pass descriptor. The render command encoder holds all the information necessary to send to the GPU so that it can draw the vertices.

If the system fails to create a Metal object, such as the command buffer or render encoder, that's a fatal error. The view's `currentRenderPassDescriptor` may not be available in a particular frame, and usually you'll just return from the rendering delegate method. Because you're asking for it only once in this playground, you get a fatal error.

- Add the following code:

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

This code gives the render encoder the pipeline state that you set up earlier.

The sphere mesh that you loaded earlier holds a buffer containing a simple list of vertices.

- Give this buffer to the render encoder by adding the following code:

```
renderEncoder.setVertexBuffer(  
    mesh.vertexBuffers[0].buffer, offset: 0, index: 0)
```

The `offset` is the position in the buffer where the vertex information starts. The `index` is how the GPU vertex shader function locates this buffer.

## Submeshes

The mesh is made up of submeshes. When artists create 3D models, they design them with different material groups. These translate to submeshes. For example, if you were rendering a car object, you might have a shiny car body and rubber tires. One material is `shiny` paint and another is `rubber`. On import, Model I/O creates two different submeshes that index to the correct vertices for that group. One vertex can be rendered multiple times by different submeshes. This sphere only has one submesh, so you'll use only one.

- Add this code:

```
guard let submesh = mesh.submeshes.first else {  
    fatalError()  
}
```

Now for the exciting part: drawing! You draw in Metal with a `draw call`.



► Add this code:

```
renderEncoder.drawIndexedPrimitives(  
    type: .triangle,  
    indexCount: submesh.indexCount,  
    indexType: submesh.indexType,  
    indexBuffer: submesh.indexBuffer.buffer,  
    indexBufferOffset: 0)
```

Here, you're instructing the GPU to render a vertex buffer consisting of triangles with the vertices placed in the correct order by the submesh index information. This code does not do the actual render — that doesn't happen until the GPU has received all the command buffer's commands.

► To complete sending commands to the render command encoder and finalize the frame, add this code:

```
// 1  
renderEncoder.endEncoding()  
// 2  
guard let drawable = view.currentDrawable else {  
    fatalError()  
}  
// 3  
commandBuffer.present(drawable)  
commandBuffer.commit()
```

Going through the code:

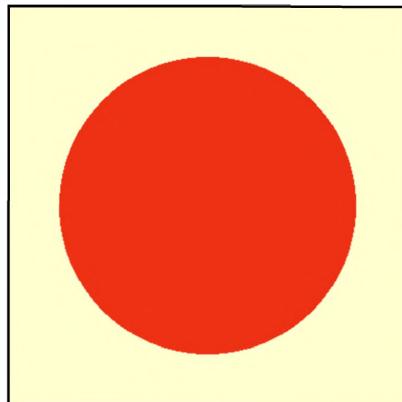
1. You tell the render encoder that there are no more draw calls and end the render pass.
2. You get the `drawable` from the `MTKView`. The `MTKView` is backed by a Core Animation `CAMetalLayer` and the layer owns a drawable texture which Metal can read and write to.
3. Ask the command buffer to present the `MTKView`'s drawable and commit to the GPU.

► Finally, add this code to the end of the playground:

```
PlaygroundPage.current.liveView = view
```

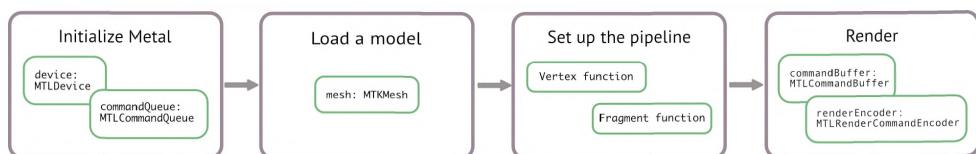
With that line of code, you'll be able to see the Metal view in the Assistant editor.

- Run the playground, and in the playground's live view, you'll see a red sphere on a cream background.



**Note:** Sometimes playgrounds don't compile or run when they should. If you're sure you've written the code correctly, then restart Xcode and reload the playground. Wait for a second or two before running.

Congratulations! You've written your first Metal app, and you've also used many of the Metal API commands that you'll use in every Metal app you write.



## Challenge

Where you created the initial sphere mesh, experiment with setting the sphere to different sizes. For example, change the size from:

```
[0.75, 0.75, 0.75]
```

To:

```
[0.2, 0.75, 0.2]
```

Change the color of the sphere. In the shader function string, you'll see:

```
return float4(1, 0, 0, 1);
```

This code returns red=1, green=0, blue=0, alpha=1, which results in the red color. Try changing the numbers (from zero to 1) for a different color. Try this green, for example:

```
return float4(0, 0.4, 0.21, 1);
```

In the next chapter, you'll examine 3D models up close in Blender. Then continuing in your Swift Playground, you'll import and render a train model.

## Key Points

- Rendering means to create an image from three-dimensional points.
- A frame is an image that the GPU renders sixty times a second (optimally).
- `device` is a software abstraction for the hardware GPU.
- A 3D model consists of a vertex mesh with shading materials grouped in submeshes.
- Create a command queue at the start of your app. This action organizes the command buffer and command encoders that you'll create every frame.
- Shader functions are programs that run on the GPU. You position vertices and color the pixels in these programs.
- The render pipeline state fixes the GPU into a particular state. It can set which shader functions the GPU should run and how vertex layouts are formatted.

Learning computer graphics is difficult. The Metal API is modern, and it takes a lot of pain out of the learning, but you need to know a lot of information up-front. Even if you feel overwhelmed at the moment, continue with the next chapters. Repetition will help with your understanding.



# Chapter 2: 3D Models

Do you know what makes a good game even better? Gorgeous graphics!

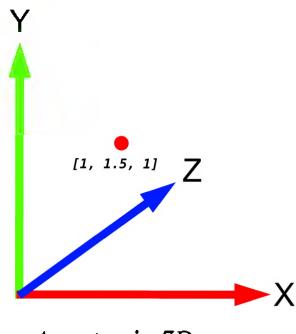
Creating amazing graphics – like those in *Divinity: Original Sin 2*, *Diablo 3* and *The Witcher 3* – generally requires a team of programmers and 3D artists who are fairly skilled at what they do. The graphics you see onscreen are created using 3D models that are rendered with custom renderers, similar to the one you wrote in the previous chapter, only more advanced. Nevertheless, the principle of rendering 3D models is still the same.

In this chapter, you'll learn all about 3D models, including how to render them onscreen, and how to work with them in Blender.



# What Are 3D Models?

3D models are made up of vertices. Each vertex refers to a point in 3D space using **x**, **y** and **z** values.

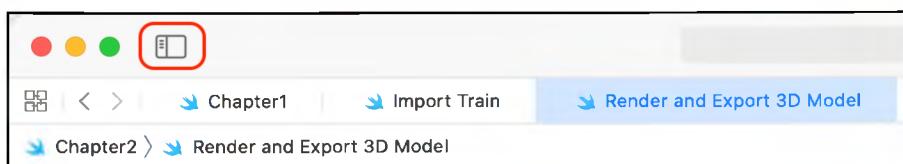


A vertex in 3D space.

As you saw in the previous chapter, you send these vertex points to the GPU for rendering. You need three vertices to create a triangle, and GPUs are able to render triangles efficiently. To show smaller details, a 3D model may also use textures. You'll learn more about textures in Chapter 8, "Textures".

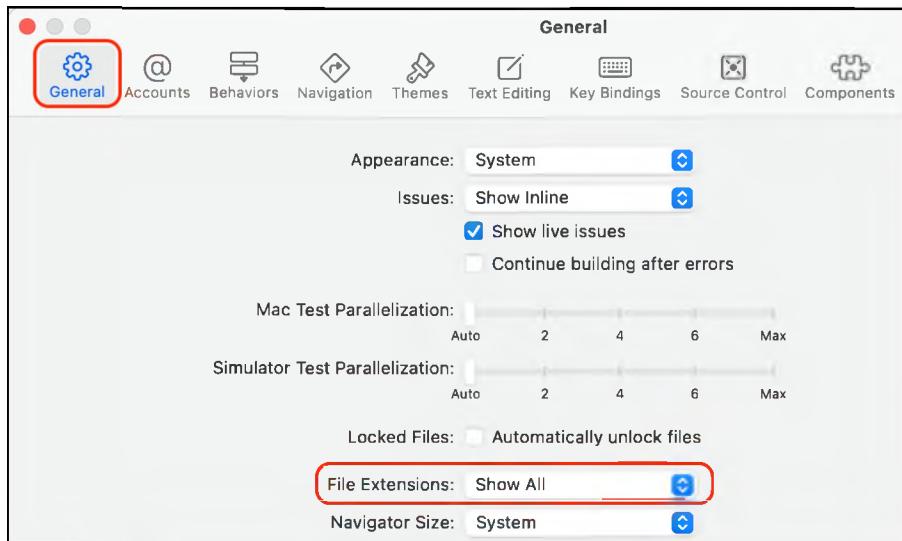
- Open the starter playground for this chapter.

This playground contains the train model in two formats (**.obj** and **.usd**), as well as two pages (**Render and Export 3D Model** and **Import Train**). If you don't see these items, you may need to hide/show the Project navigator using the icon at the top-left.



The Project Navigator

To show the file extensions, open **Xcode Preferences**, and on the **General** tab, choose **File Extensions: Show All**.



### Show File Extensions

- From the Project navigator, select **Render and Export 3D Model**.

This page contains the code from Chapter 1, “Hello, Metal!”. Examine the rendered sphere in the playground’s live view. Notice how the sphere renders as a solid red shape and appears flat.

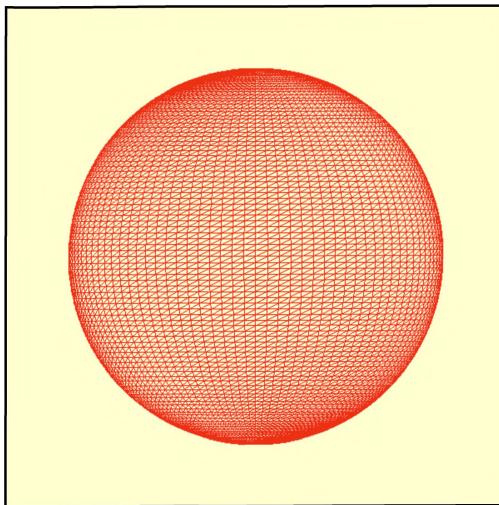
To see the edges of each individual triangle, you can render the model in **wireframe**.

- To render in wireframe, add the following line of code just before the draw call:

```
renderEncoder.setTriangleFillMode(.lines)
```

This code tells the GPU to render lines instead of solid triangles.

► Run the playground:

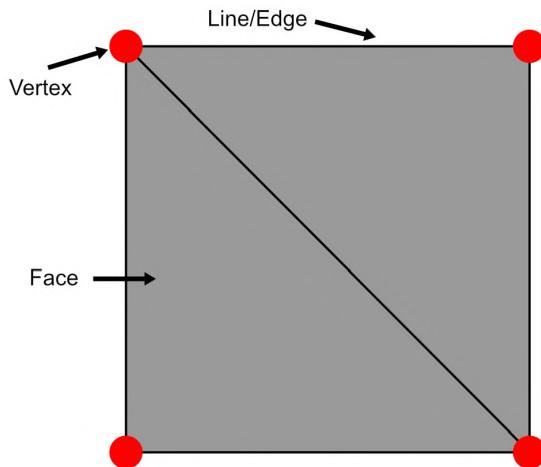


*A sphere rendered in wireframe.*

There's a bit of an optical illusion happening here. It may not look like it, but the GPU is rendering straight lines. The reason the sphere edges look curved is because of the number of triangles the GPU is rendering. If you render fewer triangles, curved models tend to look "blocky".

You can really see the 3D nature of the sphere now. The model's triangles are evenly spaced horizontally, but because you're viewing on a two dimensional screen, they appear smaller at the edges of the sphere than the triangles in the middle.

In 3D apps such as Blender or Maya, you generally manipulate **points**, **lines** and **faces**. Points are the vertices; lines, also called edges, are the lines between the vertices; and faces are the triangular flat areas.



*Vertex, line and face.*

The vertices are generally ordered into triangles because GPU hardware is specialized to process them. The GPU's core instructions are expecting to see a triangle. Of all possible shapes, why a triangle?

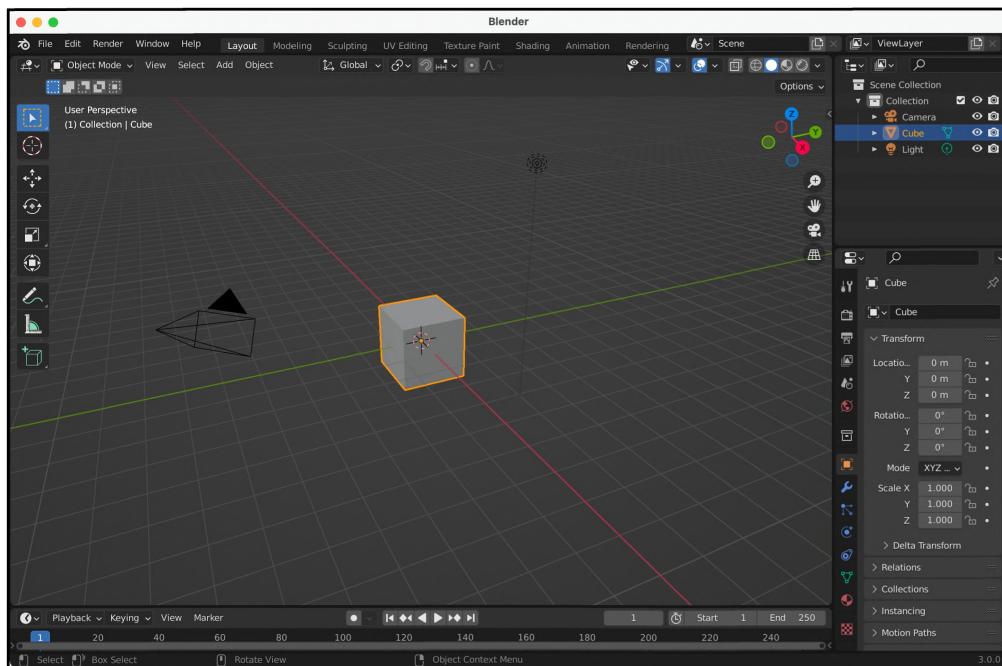
- A triangle has the least number of points of any polygon that can be drawn in two dimensions.
- No matter which way you move the points of a triangle, the three points will always be on the same plane.
- When you divide a triangle starting from any vertex, it always becomes two triangles.

When you're modeling in a 3D app, you generally work with quads (four point polygons). Quads work well with subdivision or smoothing algorithms.

# Creating Models With Blender

To create 3D models, you need a 3D modeling app. These apps range from free to hugely expensive. The best of the free apps — and the one used throughout this book — is Blender (v. 3.0). A lot of professionals use Blender, but if you’re more familiar with another 3D app — such as Cheetah3D, Maya or Houdini — then you’re welcome to use it since the concepts are the same.

- Download and install Blender from <https://www.blender.org>.
- Launch Blender. Click outside of the splash screen to close it, and you’ll see an interface similar to this one:



*The Blender Interface.*

Your interface may look different. However, if you want your Blender interface to look like the image shown here, choose **Edit > Preferences...**. Click the hamburger menu at the bottom left, choose **Load Factory Preferences**, and then click the pop-up **Load Factory Preferences**, which will appear under the cursor. Click **Save Preferences** to retain these preferences for future sessions.

**Note:** If you want to create your own models, the best place to start is with our Blender tutorial (<https://bit.ly/3gwKiel>). This tutorial teaches you how to make a mushroom. You can then render your mushroom in your playground at the end of this chapter.



*A mushroom modeled in Blender.*

## 3D File Formats

There are several standard 3D file formats. Here's an overview of what each one offers:

- **.obj:** This format, developed by Wavefront Technologies, has been around for awhile, and almost every 3D app supports importing and exporting .obj files. You can specify materials (textures and surface properties) using an accompanying .mtl file, however, this format does not support animation or vertex colors.
- **.glTF:** Developed by Khronos — who oversee Vulkan and OpenGL — this format is relatively new and is still under active development. It has strong community support because of its flexibility. It supports animated models.
- **.blend:** This is the native Blender file format.

- **.fbx:** A proprietary format owned by Autodesk. This is a commonly used format that supports animation but is losing favor because it's proprietary and doesn't have a single standard.
- **.usd:** A scalable open source format introduced by Pixar. USD can reference many models and files, which is not ideal for sharing assets. **.usdz** is a USD archive file that contains everything needed for the model or scene. Apple uses the **USDZ** format for their AR models.

An **.obj** file contains only a single model, whereas **.glTF** and **.usd** files are containers for entire scenes, complete with models, animation, cameras and lights.

In this book, you'll use **Wavefront OBJ (.obj)**, **USD**, **USDZ** and **Blender format (.blend)**.

**Note:** You can use Apple's Reality Converter to convert 3D files to **USDZ**. Apple also provides tools for validating and inspecting **USDZ** files (<https://apple.co/3gykNcl>), as well as a gallery of sample **USDZ** files (<https://apple.co/3iJzMBW>).

## Exporting to Blender

Now that you have Blender all set up, it's time to export a model from your playground into Blender.

► Still in **Render and Export 3D Model**, near the top of the playground where you create the mesh, change:

```
let mdlMesh = MDLMesh(  
    sphereWithExtent: [0.75, 0.75, 0.75],  
    segments: [100, 100],  
    inwardNormals: false,  
    geometryType: .triangles,  
    allocator: allocator)
```

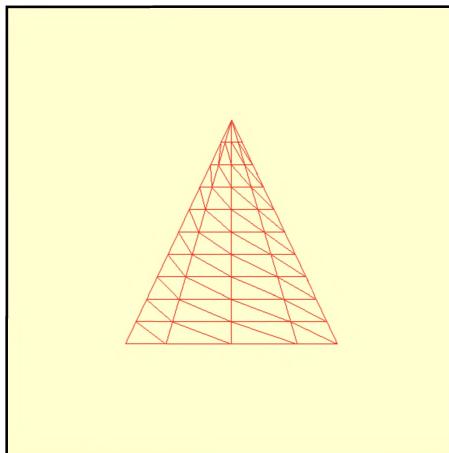
To:

```
let mdlMesh = MDLMesh(  
    coneWithExtent: [1,1,1],  
    segments: [10, 10],  
    inwardNormals: false,  
    cap: true,
```



```
geometryType: .triangles,  
allocator: allocator)
```

This code will generate a primitive cone mesh in place of the sphere. Run the playground, and you'll see the wireframe cone.



*A cone model.*

This is the model you'll export using Model I/O.

- Open Finder, and in the **Documents** folder, create a new directory named **Shared Playground Data**. All of your saved files from the Playground will end up here, so make sure you name it correctly.

**Note:** The global constant `playgroundSharedDataDirectory` holds this folder name.

- To export the cone, add this code just after creating the mesh:

```
// begin export code  
// 1  
let asset = MDLAsset()  
asset.add(mdlMesh)  
// 2  
let fileExtension = "obj"  
guard MDLAsset.canExportFileExtension(fileExtension) else {  
    fatalError("Can't export a .\$(fileExtension) format")  
}  
// 3  
do {
```

```
let url = playgroundSharedDataDirectory
    .AppendingPathComponent("primitive.\(fileExtension)")
try asset.export(to: url)
} catch {
    fatalError("Error \(error.localizedDescription)")
}
// end export code
```

Let's have a closer look at the code:

1. The top level of a scene in Model I/O is an **MDLAsset**. You can build a complete scene hierarchy by adding child objects such as meshes, cameras and lights to the asset.
  2. Check that Model I/O can export an **.obj** file type.
  3. Export the cone to the directory stored in **Shared Playground Data**.
- Run the playground to export the cone object.

**Note:** If your playground crashes, it's probably because you haven't created the **Shared Playground Data** directory in **Documents**.

## The **.obj** File Format

- In Finder, navigate to **Documents > Shared Playground Data**.

Here, you'll find the two exported files, **primitive.obj** and **primitive.mtl**.

- Using a plain text editor, open **primitive.obj**.

The following is an example **.obj** file. It describes a plane primitive with four corner vertices. The cone **.obj** file looks similar, except it has more data.

```
# Apple ModelIO OBJ File: plane
mtllib plane.mtl
g submesh
v 0 0.5 -0.5
v 0 -0.5 -0.5
v 0 -0.5 0.5
v 0 0.5 0.5
vn -1 0 0
vt 1 0
vt 0 0
vt 0 1
```



```
vt 1 1
usemtl material_1
f 1/1/1 2/2/1 3/3/1
f 1/1/1 3/3/1 4/4/1
s off
```

Here's the breakdown:

- **mtllib**: This is the name of the accompanying .mtl file. This file holds the material details and texture file names for the model.
- **g**: Starts a group of vertices.
- **v**: Vertex. For the cone, you'll have 102 of these.
- **vn**: Surface normal. This is a vector that points orthogonally — that's directly outwards. You'll read more about normals later.
- **vt**: uv coordinate that determines the vertex's position on a 2D texture. Textures use uv coordinates rather than xy coordinates.
- **usemtl**: The name of a material providing the surface information — such as color — for the following faces. This material is defined in the accompanying .mtl file.
- **f**: Defines faces. You can see here that the plane has two faces, and each face has three elements consisting of a vertex/texture/normal index. In this example, the last face listed: 4/4/1 would be the fourth vertex element / the fourth texture element / the first normal element: 0 0.5 0.5 / 1 1 / -1 0 0.
- **s**: Smoothing, currently off, means there are no groups that will form a smooth surface.

## The .mtl File Format

The second file you exported contains the model's materials. Materials describe how the 3D renderer should color the vertex. For example, should the vertex be smooth and shiny? Pink? Reflective? The .mtl file contains values for these properties.

► Using a plain text editor, open **primitive.mtl**:

```
# Apple Model/I/O MTL File: primitive.mtl

newmtl material_1
    Kd 1 1 1
    Ka 0 0 0
```



```
Ks 0
ao 0
subsurface 0
metallic 0
specularTint 0
roughness 0.9
anisotropicRotation 0
sheen 0.05
sheenTint 0
clearCoat 0
clearCoatGloss 0
```

Here's the breakdown:

- **newmtl material\_1**: This is the group that contains all of the cone's vertices.
- **Kd**: The diffuse color of the surface. In this case, 1 1 1 will color the object white.
- **Ka**: The ambient color. This models the ambient lighting in the room.
- **Ks**: The specular color. The specular color is the color reflected from a highlight.

You'll read more about these and the other material properties later.

## Importing the Cone

It's time to import the cone into Blender.

► To start with a clean and empty Blender file:

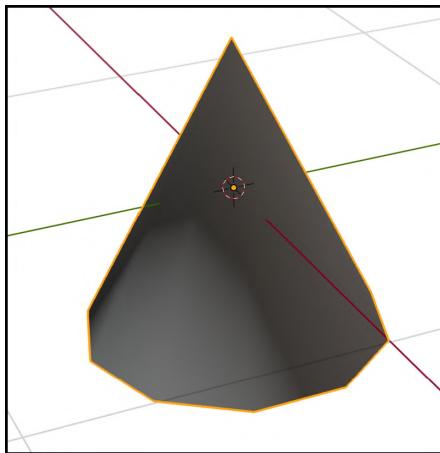
1. Open Blender.
2. Choose **File** ▶ **New** ▶ **General**.
3. Left-click the cube that appears in the start-up file to select it.
4. Press **X** to delete the cube.
5. Left-click **Delete** in the menu under the cursor to confirm the deletion.

You now have a clear and ready-for-importing Blender file, so let's get to it.

► Choose **File** ▶ **Import** ▶ **Wavefront (.obj)**, and select **primitive.obj** from the **Documents** ▶ **Shared Playground Data** Playground directory.

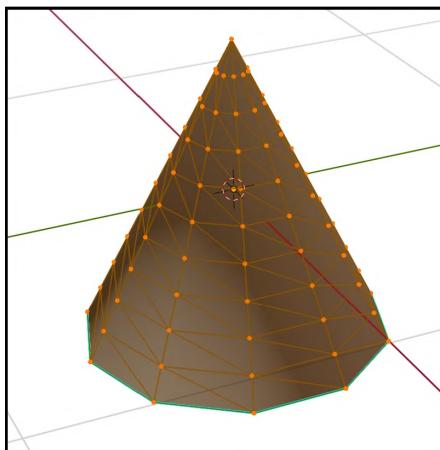


The cone imports into Blender.



*The cone in Blender.*

► **Left-click the cone** to select it, and press **Tab** to put Blender into Edit Mode. Edit Mode allows you to see the vertices and triangles that make up the cone.



*Edit mode*

While you're in Edit Mode, you can move the vertices around and add new vertices to create any 3D model you can imagine.

**Note:** In the resources directory for this chapter, there's a file with links to some excellent Blender tutorials.

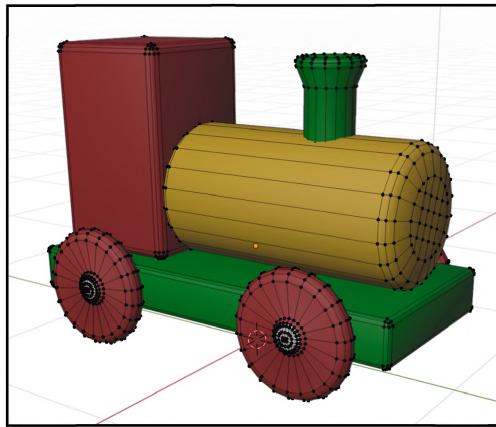
Using only a playground, you now have the ability to create, render and export a primitive. In the next part of this chapter, you'll review and render a more complex model with separate material groups.

## Material Groups

- In Blender, open **train.blend**, which you'll find in the resources directory for this chapter.

This file is the original Blender file of the .obj train in your playground.

- **Left-click the model** to select it, and press **Tab** to go into Edit Mode.

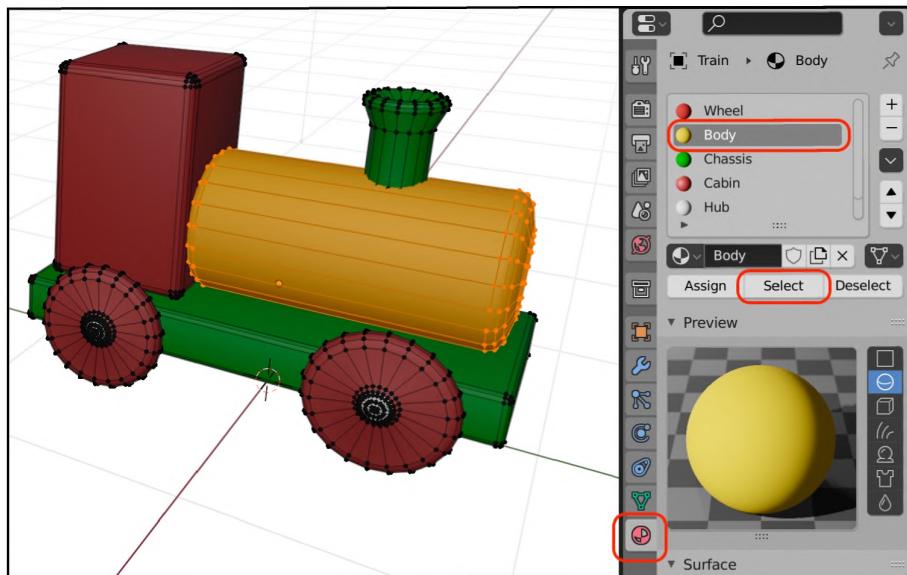


*The train in edit mode.*

Unlike the cone, the train model has several material groups — one for each color. On the right-hand side of the Blender screen, you'll see the Properties panel, with the **Material context** already selected (that's the icon at the bottom of the vertical list of icons), and the list of materials within this model at the top.

- Select **Body**, and then click **Select** underneath the material list.

The vertices assigned to this material are now colored orange.



*Material groups*

Notice how the vertices are separated into different groups or materials. This separation makes it easier to select the various parts within Blender and also gives you the ability to assign different colors.

**Note:** When you first import this model into your playground, the renderer will render each of the material groups, but it may not pick up the correct colors. One way to verify a model's appearance is to view it in Blender.

► Go back to Xcode, and from the Project navigator, open the **Import Train** playground page. This playground renders — but does not export — the wireframe cone.

In the playground's **Resources** folder, you'll see three files: **train.mtl**, **train.obj** and **train.usd**.

**Note:** Files in the Playground Resources folder are available to all playground pages. Files in each page's Resources folder are only available to that page.

- In **Import Train**, remove the line where you create the MDLMesh cone:

```
let mdlMesh = MDLMesh(
    coneWithExtent: [1, 1, 1],
    segments: [10, 10],
    inwardNormals: false,
    cap: true,
    geometryType: .triangles,
    allocator: allocator)
```

Don't worry about that compile error. You've still got some work to do.

- Replacing the code you just removed, add this code in its place:

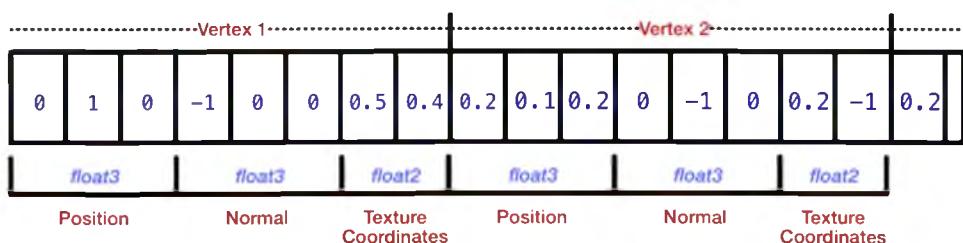
```
guard let assetURL = Bundle.main.url(
    forResource: "train",
    withExtension: "obj") else {
    fatalError()
}
```

This code sets up the file URL for the .obj format of the model. Later, you can try the .usd format, and you should get the same result.

## Vertex Descriptors

Metal uses descriptors as a common pattern to create objects. You saw this pattern in the previous chapter when you set up a pipeline descriptor to describe a pipeline state. Before loading the model, you'll tell Metal how to lay out the vertices and other data by creating a vertex descriptor.

The following diagram describes an incoming buffer of model vertex data. It has two vertices with position, normal and texture coordinate **attributes**. The vertex descriptor informs Metal how you want to view this data.



*The vertex descriptor*

- Add this code below the code you just added:

```
// 1
let vertexDescriptor = MTLVertexDescriptor()
// 2
vertexDescriptor.attributes[0].format = .float3
// 3
vertexDescriptor.attributes[0].offset = 0
// 4
vertexDescriptor.attributes[0].bufferIndex = 0
```

Looking closer:

1. You create a vertex descriptor that you'll use to configure all of the properties that an object will need to know about.

**Note:** You can reuse this vertex descriptor with either the same values or reconfigured values to instantiate a different object.

2. The .obj file holds normal and texture coordinate data as well as vertex position data. For the moment, you don't need the surface normals or texture coordinates; you only need the position. You tell the descriptor that the xyz position data should load as a float3, which is a SIMD data type consisting of three Float values. An MTLVertexDescriptor has an array of 31 attributes where you can configure the data format — and in future chapters, you'll load the normal and texture coordinate attributes.
3. The offset specifies where in the buffer this particular data will start.
4. When you send your vertex data to the GPU via the render encoder, you send it in an MTLBuffer and identify the buffer by an index. There are 31 buffers available and Metal keeps track of them in a **buffer argument table**. You use buffer 0 here so that the vertex shader function will be able to match the incoming vertex data in buffer 0 with this vertex layout.

- Now add this code below the previous lines:

```
// 1
vertexDescriptor.layouts[0].stride =
    MemoryLayout<SIMD3<Float>>.stride
// 2
let meshDescriptor =
    MTKModelIOVertexDescriptorFromMetal(vertexDescriptor)
// 3
(meshDescriptor.attributes[0] as! MDLVertexAttribute).name =
```



### MDLVertexAttributePosition

Going through everything:

1. Here, you specify the stride for buffer 0. The stride is the number of bytes between each set of vertex information. Referring back to the previous diagram which described position, normal and texture coordinate information, the stride between each vertex would be `float3 + float3 + float2`. However, here you're only loading position data, so to get to the next position, you jump by a stride of `float3`.

Using the buffer layout index and stride format, you can set up complex vertex descriptors referencing multiple `MTLBuffers` with different layouts. You have the option of interleaving position, normal and texture coordinates; or you can lay out a buffer containing all of the position data first, followed by other data.

**Note:** The `SIMD3<Float>` type is Swift's equivalent to `float3`. Later, you'll use a typealias for `float3`.

2. Model I/O needs a slightly different format vertex descriptor, so you create a new Model I/O descriptor from the Metal vertex descriptor. If you have a Model I/O descriptor and need a Metal one, `MTKMetalVertexDescriptorFromModelIO()` provides a solution.
3. Assign a string name “position” to the attribute. This tells Model I/O that this is positional data. The normal and texture coordinate data is also available, but with this vertex descriptor, you told Model I/O that you're not interested in those attributes.

► Continue by adding this code:

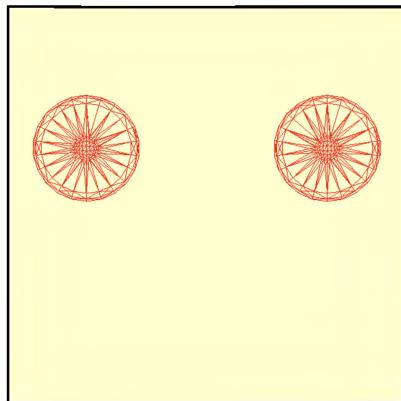
```
let asset = MDLAsset(  
    url: assetURL,  
    vertexDescriptor: meshDescriptor,  
    bufferAllocator: allocator)  
let mdlMesh =  
    asset.childObjects(of: MDLMesh.self).first as! MDLMesh
```

This code reads the asset using the URL, vertex descriptor and memory allocator. You then read in the first Model I/O mesh buffer in the asset. Some more complex objects will have multiple meshes, but you'll deal with that later.



Now that you've loaded the model vertex information, the rest of the code will be the same, and your playground will load mesh from the new `mdlMesh` variable.

- Run the playground to see your train in wireframe.

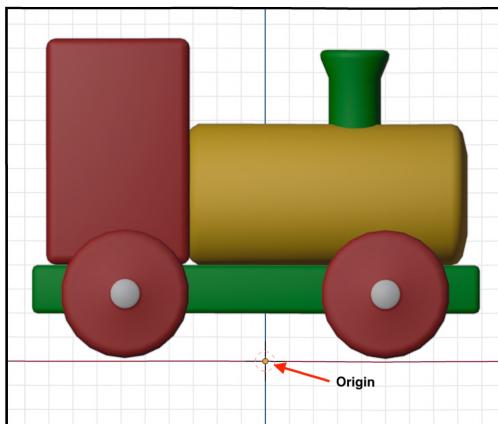


*Train wireframe wheels*

Well, that's not good. The train is missing some wheels, and the ones that are there are way too high off the ground. Plus, the rest of the train is missing! Time to fix these problems, starting with the train's wheels.

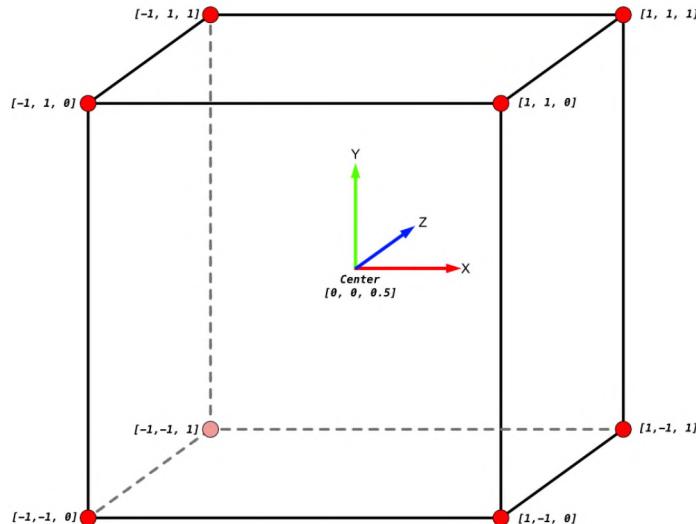
## Metal Coordinate System

All models have an **origin**. The origin is the location of the mesh. The train's origin is at `[0, 0, 0]`. In Blender, this places the train right at the center of the scene.



*The origin*

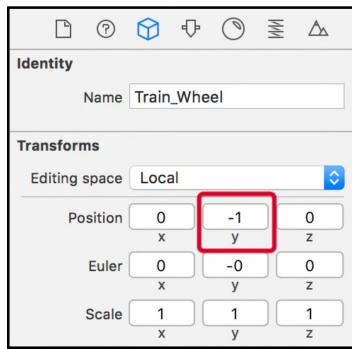
The Metal NDC (Normalized Device Coordinate) system is a 2-unit wide by 2-unit high by 1-unit deep box where **X** is right / left, **Y** is up / down and **Z** is in / out of the screen.



*NDC (Normalized Device Coordinate) system*

To normalize means *to adjust to a standard scale*. On a screen, you might address a location in screen coordinates of width 0 to 375, whereas the Metal normalized coordinate system doesn't care what the physical width of a screen is — its coordinates along the X axis are **-1.0** to **1.0**. In Chapter 6, “Coordinate Spaces”, you'll learn about various coordinate systems and spaces. Because the origin of the train is at **[0, 0, 0]**, the train appears halfway up the screen, which is where **[0, 0, 0]** is in the Metal coordinate system.

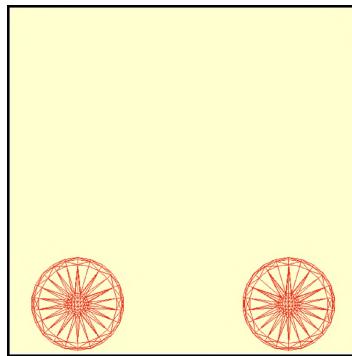
- Select **train.obj** in the Project navigator. The SceneKit editor opens and shows you the train model. Currently, the editor doesn't apply the materials, so the train appears white on a white background. You can still select the train by clicking somewhere around the center of the window. Do that now, and you'll see three arrows appear when you select the train. Now, open the **Node inspector** on the right, and change **y Position** to **-1**.



The Node inspector

**Note:** Typically, you'd change the position of the model in code. However, the purpose of this example is to illustrate how you can affect the model.

- Go back to **Import Train**, and run the playground. The wheels now appear at the bottom of the screen.

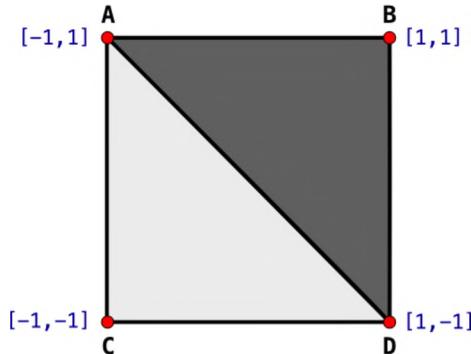


Wheels on the ground

Now that the wheels are fixed, you're ready to solve the case of the missing train!

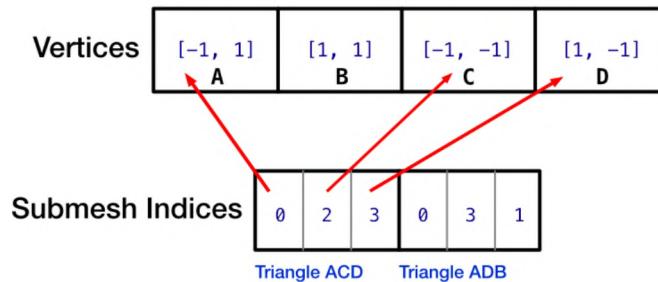
## Submeshes

So far, your primitive models included only one material group, and thus one submesh. Here's a plane with four vertices and two material groups.



*Vertices on a plane*

When Model I/O loads this plane, it places the four vertices in an `MTLBuffer`. The following image shows the vertex position data and also how two submesh buffers index into the vertex data.



*Submesh buffers*

The first submesh buffer holds the vertex indices of the light-colored triangle ACD. These indices point to vertices 0, 2 and 3. The second submesh buffer holds the indices of the dark triangle ADB. The submesh also has an **offset** where the submesh buffer starts. The index can be held in either a `uint16` or a `uint32`. The offset of this second submesh buffer would be three times the size of the `uint` type.

## Winding Order

The vertex order, also known as the **winding order**, is important here. The vertex order of this plane is counter-clockwise, as is the default .obj winding order. With a counter-clockwise winding order, triangles that are defined in *counter-clockwise* order are facing toward you. Whereas triangles that are in *clockwise* order are facing away from you. In the next chapter, you'll go down the graphics pipeline and you'll see that the GPU can cull triangles that are not facing toward you, saving valuable processing time.

## Render Submeshes

Currently, you're only rendering the first submesh, but because the train has several material groups, you'll need to loop through the submeshes to render them all.

- Toward the end of the playground, change:

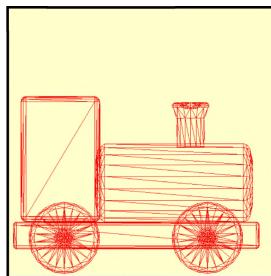
```
guard let submesh = mesh.submeshes.first else {
    fatalError()
}
renderEncoder.drawIndexedPrimitives(
    type: .triangle,
    indexCount: submesh.indexCount,
    indexType: submesh.indexType,
    indexBuffer: submesh.indexBuffer.buffer,
    indexBufferOffset: 0)
```

To:

```
for submesh in mesh.submeshes {
    renderEncoder.drawIndexedPrimitives(
        type: .triangle,
        indexCount: submesh.indexCount,
        indexType: submesh.indexType,
        indexBuffer: submesh.indexBuffer.buffer,
        indexBufferOffset: submesh.indexBuffer.offset
    )
}
```

This code loops through the submeshes and issues a draw call for each one. The mesh and submeshes are in `MTLBuffers`, and the submesh holds the index listing of the vertices in the mesh.

- Run the playground, and your train renders completely — minus the material colors, which you’ll take care of in Chapter 11, “Maps & Materials”.



*The final train*

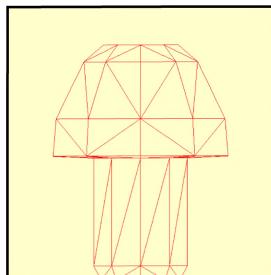
Congratulations! You’re now rendering 3D models. For now, don’t worry that you’re only rendering them in two dimensions or that the colors aren’t correct. After the next chapter, you’ll know more about the internals of rendering. Following on from that, you’ll learn how to move those vertices into the third dimension.

## Challenge

If you’re in for a fun challenge, complete the Blender tutorial to make a mushroom (<https://bit.ly/3gwKiel>), and then export what you make in Blender to an .obj file. If you want to skip the modeling, you’ll find the **mushroom.obj** file in the resources directory for this chapter.

- Import **mushroom.obj** into the playground and render it.

If you use the mushroom from the resources directory, you’ll first have to scale and reposition the mushroom in the SceneKit editor to view it correctly.



*Wireframe mushroom*

If you have difficulty, the completed playground is in the Projects ➤ Challenge directory for this chapter.

## Key Points

- 3D models consist of vertices. Each vertex has a position in 3D space.
- In 3D modeling apps, you create models using quads, or polygons with four vertices. On import, Model I/O converts these quads to triangles.
- Triangles are the GPU's native format.
- Blender is a fully-featured professional free 3D modeling, animation and rendering app available from <https://www.blender.org>.
- There are many 3D file formats. Apple has standardized its AR models on Pixar's USD format in a compressed USDZ format.
- Vertex descriptors describe the buffer format for the model's vertices. You set the GPU pipeline state with the vertex descriptor, so that the GPU knows what the vertex buffer format is.
- A model is made up of at least one submesh. This submesh corresponds to a material group where you can define the color and other surface attributes of the group.
- Metal Normalized Device Coordinates are  $-1$  to  $1$  on the X and Y axes, and  $0$  to  $1$  on the Z axis. X is left / right, Y is down / up and Z is front / back.
- The GPU will render only vertices positioned in Metal NDC.



# Chapter 3: The Rendering Pipeline

Now that you know a bit more about 3D models and rendering, it's time to take a drive through the rendering pipeline. In this chapter, you'll create a Metal app that renders a red cube. As you work your way through this chapter, you'll get a closer look at the hardware that's responsible for turning your 3D objects into the gorgeous pixels you see onscreen. First up, the GPU and CPU.



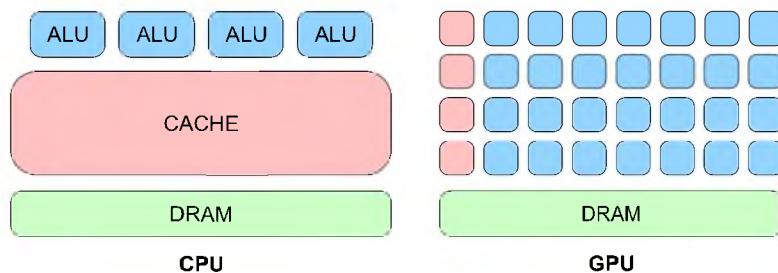
# The GPU and CPU

Every computer comes equipped with a **Graphics Processing Unit (GPU)** and **Central Processing Unit (CPU)**.

The GPU is a specialized hardware component that can process images, videos and massive amounts of data *really* fast. This operation is known as **throughput** and is measured by the amount of data processed in a specific unit of time. The CPU, on the other hand, manages resources and is responsible for the computer's operations. Although the CPU can't process huge amounts of data like the GPU, it *can* process many sequential tasks (one after another) really fast. The time necessary to process a task is known as **latency**.

The ideal setup includes low latency and high throughput. Low latency allows for the serial execution of queued tasks, so the CPU can execute the commands without the system becoming slow or unresponsive — and high throughput lets the GPU render videos and games asynchronously without stalling the CPU. Because the GPU has a highly parallelized architecture specialized in doing the same task repeatedly and with little or no data transfers, it can process larger amounts of data.

The following diagram shows the major differences between the CPU and GPU.

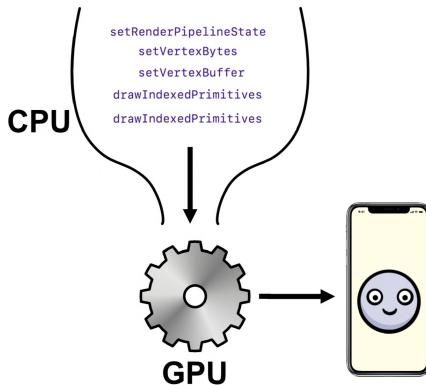


*Differences between CPU and GPU*

The CPU has a large cache memory and a handful of **Arithmetic Logic Unit (ALU)** cores. In contrast, the GPU has a small cache memory and many ALU cores. The low latency cache memory on the CPU is used for fast access to temporary resources. The ALU cores on the GPU handle calculations without saving partial results to memory.

The CPU typically has only a few cores, while the GPU has hundreds — even thousands of cores. With more cores, the GPU can split the problem into many smaller parts, each running on a separate core in parallel, which helps to hide latency. At the end of processing, the partial results are combined, and the final result is returned to the CPU. But cores aren't the only thing that matters.

Besides being slimmed down, GPU cores also have special circuitry for processing geometry and are often called **shader cores**. These shader cores are responsible for the beautiful colors you see onscreen. The GPU writes an entire frame at a time to fit the full rendering window; it then proceeds to rendering the next frame as quickly as possible, so it can maintain a respectable frame rate.



*CPU sending commands to GPU*

The CPU continues to issue commands to the GPU, ensuring that the GPU always has work to do. However, at some point, either the CPU will finish sending commands or the GPU will finish processing them. To avoid stalling, Metal on the CPU queues up multiple commands in command buffers and will issue new commands, sequentially, for the next frame without waiting for the GPU to finish the previous frame. This means that no matter who finishes the work first, there will always be more work to do.

The GPU part of the graphics pipeline starts after the GPU receives all of the commands and resources. To get started with the rendering pipeline, you'll set up these commands and resources in a new project.

## The Metal Project

So far, you've been using Playgrounds to learn about Metal. Playgrounds are great for testing and learning new concepts, but it's also important to understand how to set up a full Metal project using SwiftUI.

- In Xcode, create a new project using the **Multiplatform App** template.
- Name your project **Pipeline**, and fill out your team and organization identifier. Leave all of the checkbox options unchecked.

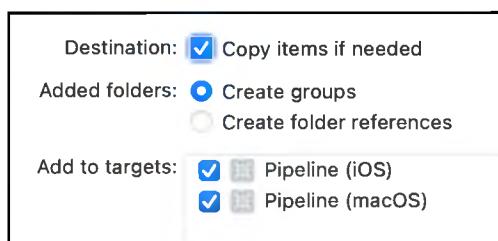
- Choose the location for your new project.

Excellent, you now have a fancy, new SwiftUI app. **ContentView.swift** is the main view for the app; this is where you'll call your Metal view.

The MetalKit framework contains an **MTKView**, which is a special Metal rendering view. This is a **UIView** on iOS and an **NSView** on macOS. To interface with **UIKit** or **Cocoa** UI elements, you'll use a **Representable** protocol that sits between **SwiftUI** and your **MTKView**. If you want to understand how this protocol works, you can find the information in our book, **SwiftUI Apprentice**.

This configuration is all rather complicated, so in the resources folder for this chapter, you'll find a pre-made **MetalView.swift**.

- Drag this file into your project, making sure that you check all of the checkboxes so that you copy the file and add it to both targets.



*Adding files to targets*

- Open **MetalView.swift**. **MetalView** is a SwiftUI View structure that contains the **MTKView** property and hosts the Metal view.

- Open **ContentView.swift**, and change:

```
Text("Hello, world!")
    .padding
```

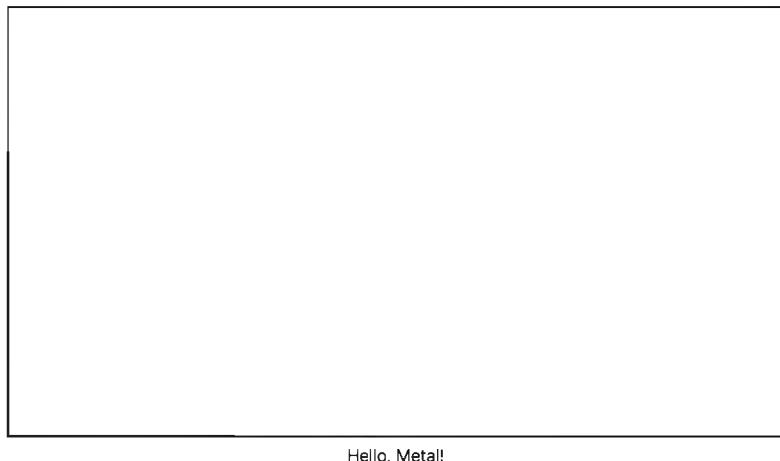
To:

```
VStack {
    MetalView()
        .border(Color.black, width: 2)
    Text("Hello, Metal!")
}
.padding()
```

Here, you add `MetalView` to the view hierarchy and give it a border.

- Build and run your application using either the macOS target or the iOS target.

You'll see your hosted `MTKView`. The advantage of using SwiftUI is that it's relatively easy to layer UI elements — such as the "Hello Metal" text here — underneath your Metal view.



*Initial SwiftUI View*

You now have a choice. You can subclass `MTKView` and replace the `MTKView` in `MetalView` with the subclassed one. In this case, the subclass's `draw(_:)` would get called every frame, and you'd put your drawing code in that method. However, in this book, you'll set up a `Renderer` class that conforms to `MTKViewDelegate` and sets `Renderer` as a delegate of `MTKView`. `MTKView` calls a delegate method every frame, and this is where you'll place the necessary drawing code.

**Note:** If you're coming from a different API world, you might be looking for a game loop construct. You do have the option of using `CADisplayLink` for timing, but Apple introduced `MetalKit` with its protocols to manage the game loop more easily.

## The Renderer Class

- Create a new Swift file named **Renderer.swift**, and replace its contents with the following code:

```
import MetalKit

class Renderer: NSObject {
    init(metalView: MTKView) {
        super.init()
    }
}

extension Renderer: MTKViewDelegate {
    func mtkView(
        view: MTKView,
        drawableSizeWillChange size: CGSize
    ) {
    }

    func draw(in view: MTKView) {
        print("draw")
    }
}
```

Here, you create an initializer and make `Renderer` conform to `MTKViewDelegate` with the two `MTKView` delegate methods:

- `mtkView(_:_:drawableSizeWillChange:)`: Called every time the size of the window changes. This allows you to update render texture sizes and camera projection.
- `draw(in:)`: Called every frame. This is where you write your render code.

- Open `MetalView.swift`, and in `MetalView`, add a property to hold the renderer:

```
@State private var renderer: Renderer?
```

- Change body to:

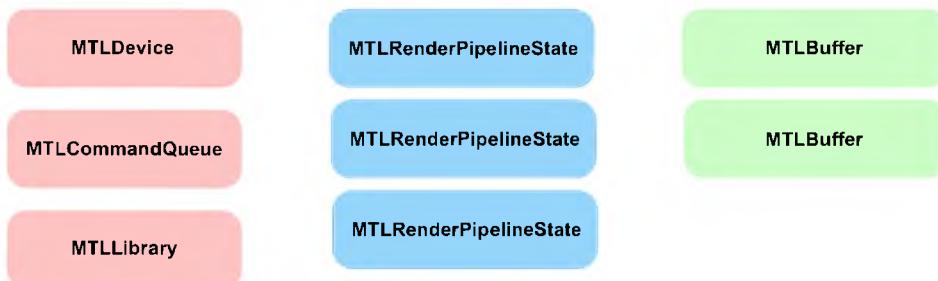
```
var body: some View {
    MetalViewRepresentable(metalView: $metalView)
        .onAppear {
            renderer = Renderer(metalView: metalView)
        }
}
```

Here, you initialize the renderer when the metal view first appears.

## Initialization

Just as you did in the first chapter, you need to set up the Metal environment.

Metal has a major advantage over OpenGL in that you're able to instantiate some objects up-front rather than create them during each frame. The following diagram indicates some of the Metal objects you can create at the start of the app.



*Create these outside the render loop*

- **MTLDevice:** The software reference to the GPU hardware device.
- **MTLCommandQueue:** Responsible for creating and organizing **MTLCommandBuffers** every frame.
- **MTLLibrary:** Contains the source code from your vertex and fragment shader functions.
- **MTLRenderPipelineState:** Sets the information for the draw — such as which shader functions to use, what depth and color settings to use and how to read the vertex data.
- **MTLBuffer:** Holds data — such as vertex information — in a form that you can send to the GPU.

Typically, you'll have one **MTLDevice**, one **MTLCommandQueue** and one **MTLLibrary** object in your app. You'll also have several **MTLRenderPipelineState** objects that will define the various pipeline states, as well as several **MTLBuffers** to hold the data. Before you can use these objects, however, you need to initialize them.

► Open **Renderer.swift**, and add these properties to **Renderer**:

```
static var device: MTLDevice!
static var commandQueue: MTLCommandQueue!
static var library: MTLLibrary!
var mesh: MTKMesh!
```

```
var vertexBuffer: MTLBuffer!
var pipelineState: MTLRenderPipelineState!
```

All of these properties are currently implicitly unwrapped optionals for convenience, but you can add error-checking later if you wish.

You're using class properties for the device, the command queue and the library to ensure that only one of each exists. In rare cases, you may require more than one, but in most apps, one is enough.

- Still in **Renderer.swift**, add the following code to `init(metalView:)` before `super.init()`:

```
guard
    let device = MTLCreateSystemDefaultDevice(),
    let commandQueue = device.makeCommandQueue() else {
    fatalError("GPU not available")
}
Renderer.device = device
Renderer.commandQueue = commandQueue
metalView.device = device
```

This code initializes the GPU and creates the command queue.

- Finally, after `super.init()`, add this:

```
metalView.clearColor = MTLClearColor(
    red: 1.0,
    green: 1.0,
    blue: 0.8,
    alpha: 1.0)
metalView.delegate = self
```

This code sets `metalView.clearColor` to a cream color. It also sets `Renderer` as the delegate for `metalView` so that the view will call the `MTKViewDelegate` drawing methods.

- Build and run the app to make sure everything's set up and working. If everything is good, you'll see the SwiftUI view as before, and in the debug console, you'll see the word "draw" repeatedly. Use this console statement to verify that your app is calling `draw(in:)` for every frame.

**Note:** You won't see `metalView`'s cream color because you're not asking the GPU to do any drawing yet.

## Create the Mesh

You've already created a sphere and a cone using Model I/O; now it's time to create a cube.

- In `init(metalView:)`, before calling `super.init()`, add this:

```
// create the mesh
let allocator = MTKMeshBufferAllocator(device: device)
let size: Float = 0.8
let mdlMesh = MDLMesh(
    boxWithExtent: [size, size, size],
    segments: [1, 1, 1],
    inwardNormals: false,
    geometryType: .triangles,
    allocator: allocator)
do {
    mesh = try MTKMesh(mesh: mdlMesh, device: device)
} catch let error {
    print(error.localizedDescription)
}
```

This code creates the cube mesh, as you did in the previous chapter.

- Then, set up the `MTLBuffer` that contains the vertex data you'll send to the GPU.

```
vertexBuffer = mesh.vertexBuffers[0].buffer
```

This code puts the mesh data in an `MTLBuffer`. Next, you need to set up the pipeline state so that the GPU will know how to render the data.

## Set Up the Metal Library

First, set up the `MTLLibrary` and ensure that the vertex and fragment shader functions are present.

- Continue adding code before `super.init()`:

```
// create the shader function library
let library = device.makeDefaultLibrary()
Renderer.library = library
let vertexFunction = library?.makeFunction(name: "vertex_main")
let fragmentFunction =
    library?.makeFunction(name: "fragment_main")
```

Here, you set up the default library with some shader function pointers. You'll create these shader functions later in this chapter. Unlike OpenGL shaders, these functions are compiled when you compile your project, which is more efficient than compiling your functions on the fly. The result is stored in the library.

## Create the Pipeline State

To configure the GPU's state, you create a **pipeline state object (PSO)**. This pipeline state can be a render pipeline state for rendering vertices, or a compute pipeline state for running a compute kernel.

► Continue adding code before `super.init()`:

```
// create the pipeline state object
let pipelineDescriptor = MTLRenderPipelineDescriptor()
pipelineDescriptor.vertexFunction = vertexFunction
pipelineDescriptor.fragmentFunction = fragmentFunction
pipelineDescriptor.colorAttachments[0].pixelFormat =
    metalView.colorPixelFormat
pipelineDescriptor.vertexDescriptor =
    MTKMetalVertexDescriptorFromModelIO(mdlMesh.vertexDescriptor)
do {
    pipelineState =
        try device.makeRenderPipelineState(
            descriptor: pipelineDescriptor)
} catch let error {
    fatalError(error.localizedDescription)
}
```

The PSO holds a potential state for the GPU. The GPU needs to know its complete state before it can start managing vertices. Here, you set the two shader functions the GPU will call and the pixel format for the texture to which the GPU will write. You also set the pipeline's vertex descriptor; this is how the GPU will know how to interpret the vertex data that you'll present in the mesh data `MTLBuffer`.

**Note:** If you need to use a different data buffer layout or call different vertex or fragment functions, you'll need additional pipeline states. Creating pipeline states is relatively time-consuming — which is why you do it up-front — but switching pipeline states during frames is fast and efficient.

The initialization is complete, and your project compiles. Next up, you'll start on drawing your model.

## Render Frames

MTKView calls `draw(in:)` for every frame; this is where you'll set up your GPU render commands.

- In `draw(in:)`, replace the `print` statement with this:

```
guard  
    let commandBuffer = Renderer.commandQueue.makeCommandBuffer(),  
    let descriptor = view.currentRenderPassDescriptor,  
    let renderEncoder =  
        commandBuffer.makeRenderCommandEncoder(  
            descriptor: descriptor) else {  
    return  
}
```

You'll send a series of commands to the GPU contained in **command encoders**. In one frame, you might have multiple command encoders, and the **command buffer** manages these.

You create a render command encoder using a **render pass descriptor**. This contains the render target textures that the GPU will draw into. In a complex app, you may well have multiple render passes in one frame, with multiple target textures. You'll learn how to chain render passes together later.

- Continue adding this code:

```
// drawing code goes here  
  
// 1  
renderEncoder.endEncoding()  
// 2  
guard let drawable = view.currentDrawable else {  
    return  
}  
commandBuffer.present(drawable)  
// 3  
commandBuffer.commit()
```

Here's a closer look at the code:

1. After adding the GPU commands to a command encoder, you end its encoding.
2. You present the view's drawable texture to the GPU.
3. When you commit the command buffer, you send the encoded commands to the GPU for execution.

## Drawing

It's time to set up the list of commands that the GPU will need to draw your frame. In other words, you'll:

- Set the pipeline state to configure the GPU hardware.
  - Give the GPU the vertex data.
  - Issue a draw call using the mesh's submesh groups.
- Still in `draw(in:)`, replace the comment:

```
// drawing code goes here
```

With:

```
renderEncoder.setRenderPipelineState(pipelineState)
renderEncoder.setVertexBuffer(vertexBuffer, offset: 0, index: 0)
for submesh in mesh.submeshes {
    renderEncoder.drawIndexedPrimitives(
        type: .triangle,
        indexCount: submesh.indexCount,
        indexType: submesh.indexType,
        indexBuffer: submesh.indexBuffer.buffer,
        indexBufferOffset: submesh.indexBuffer.offset)
}
```

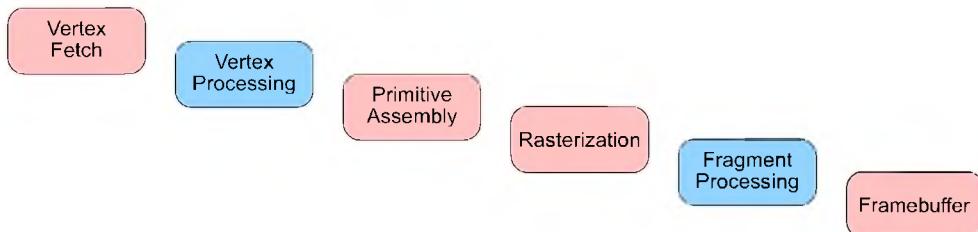
Great, you set up the GPU commands to set the pipeline state and the vertex buffer, and to perform the draw calls on the mesh's submeshes. When you commit the command buffer at the end of `draw(in:)`, you're telling the GPU that the data and pipeline are ready, and it's time for the GPU to take over.



# The Render Pipeline

Are you ready to investigate the GPU pipeline? Great, let's get to it!

In the following diagram, you can see the stages of the pipeline.



*The render pipeline*

The graphics pipeline takes the vertices through multiple stages, during which the vertices have their coordinates transformed between various spaces.

**Note:** This chapter describes immediate-mode rendering (IMR) architecture. Apple's chips for iOS since A11, and Silicon for macOS, use tile-based rendering (TBR). New Metal features are able to take advantage of TBR. However, for simplicity, you'll start off with a basic understanding of general GPU architecture. If you want a preview of some differences, watch Apple's WWDC 2020 video Bring your Metal app to Apple silicon Macs (<https://developer.apple.com/videos/play/wwdc2020/10631/>).

As a Metal programmer, you're only concerned about the Vertex and Fragment Processing stages since they're the only two programmable stages. Later in the chapter, you'll write both a vertex shader and a fragment shader. For all the non-programmable pipeline stages, such as Vertex Fetch, Primitive Assembly and Rasterization, the GPU has specially designed hardware units to serve those stages.

## 1 - Vertex Fetch

The name of this stage varies among different graphics **Application Programming Interfaces (APIs)**. For example, **DirectX** calls it **Input Assembler**.

To start rendering 3D content, you first need a scene. A scene consists of models that have meshes of vertices. One of the simplest models is the cube which has six faces (12 triangles). As you saw in the previous chapter, you use a vertex descriptor to define the way vertices are read in along with their attributes — such as position, texture coordinates, normal and color. You do have the option **not** to use a vertex descriptor and just send an array of vertices in an **MTLBuffer**, however, if you decide not to use one, you'll need to know how the vertex buffer is organized ahead of time.

When the GPU fetches the vertex buffer, the **MTLRenderCommandEncoder** draw call tells the GPU whether the buffer is indexed. If the buffer is not indexed, the GPU assumes the buffer is an array, and it reads in one element at a time, in order.

In the previous chapter, you saw how Model I/O imports .obj files and sets up their buffers indexed by submesh. This indexing is important because vertices are cached for reuse. For example, a cube has 12 triangles and eight vertices (at the corners). If you don't index, you'll have to specify the vertices for each triangle and send 36 vertices to the GPU. This may not sound like a lot, but in a model that has several thousand vertices, vertex caching is important.

There is also a second cache for shaded vertices so that vertices that are accessed multiple times are only shaded once. A shaded vertex is one to which color was already applied. But that happens in the next stage.

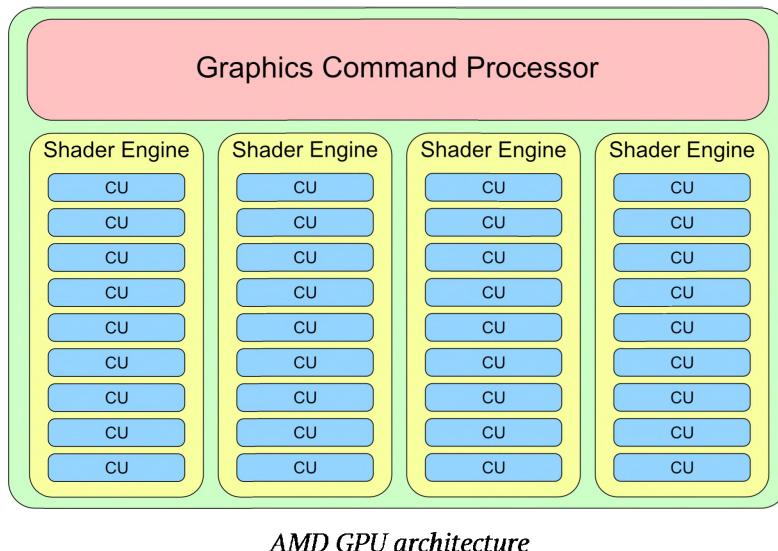
A special hardware unit known as the **Scheduler** sends the vertices and their attributes on to the **Vertex Processing** stage.

## 2 - Vertex Processing

In the Vertex Processing stage, vertices are processed individually. You write code to calculate per-vertex lighting and color. More importantly, you send vertex coordinates through various coordinate spaces to reach their position in the final framebuffer.

You briefly learned about shader functions and about the **Metal Shading Language (MSL)** in Chapter 1, “Hello, Metal!”. Now it’s time to see what happens under the hood at the hardware level.

Look at this diagram of the architecture of an AMD GPU:



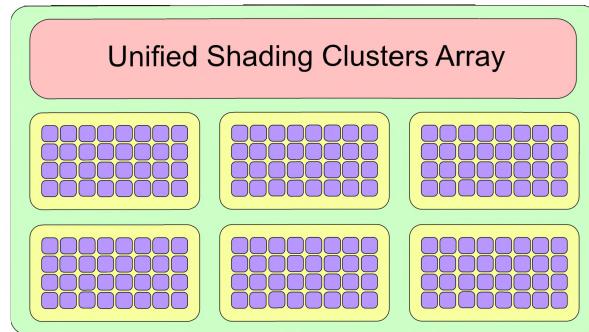
Going top-down, the GPU has:

- **1 Graphics Command Processor:** This coordinates the work processes.
- **4 Shader Engines (SE):** An **SE** is an organizational unit on the GPU that can serve an entire pipeline. Each **SE** has a geometry processor, a rasterizer and Compute Units.
- **9 Compute Units (CU):** A **CU** is nothing more than a group of shader cores.
- **64 shader cores:** A **shader core** is the basic building block of the GPU where all of the shading work is done.

In total, the **36 CUs** have **2,304 shader cores**. Compare that to the number of cores in your 8-core CPU.

For mobile devices, the story is a little different. For comparison, look at the following image showing a GPU similar to those in recent iOS devices. Instead of having **SEs** and **CUs**, the PowerVR GPU has **Unified Shading Clusters (USC)**.

This particular GPU model has **6 USC**s and **32 cores** per **USC** for a total of only **192 cores**.



*The PowerVR GPU*

**Note:** The iPhone X had the first mobile GPU entirely designed in-house by Apple. As it turns out, Apple has not made the GPU hardware specifications public.

So what can you do with that many cores? Since these cores are specialized in both vertex and fragment shading, one obvious thing to do is give all the cores work to do in parallel so that the processing of vertices or fragments is done faster. There are a few rules, though.

Inside a CU, you can only process either vertices or fragments, and only at one time. (Good thing there's thirty-six of those!) Another rule is that you can only process one shader function per SE. Having four SE's lets you combine work in interesting and useful ways. For example, you can run one fragment shader on one SE and a second fragment shader on a second SE at one time. Or you can separate your vertex shader from your fragment shader and have them run in parallel but on different SEs.



## Creating a Vertex Shader

It's time to see vertex processing in action. The **vertex shader** you're about to write is minimal, but it encapsulates most of the necessary vertex shader syntax you'll need in this and subsequent chapters.

- Create a new file using the **Metal File** template, and name it **Shaders.metal**. Then, add this code at the end of the file:

```
// 1
struct VertexIn {
    float4 position [[attribute(0)]];
};

// 2
vertex float4 vertex_main(const VertexIn vertexIn [[stage_in]])
{
    return vertexIn.position;
}
```

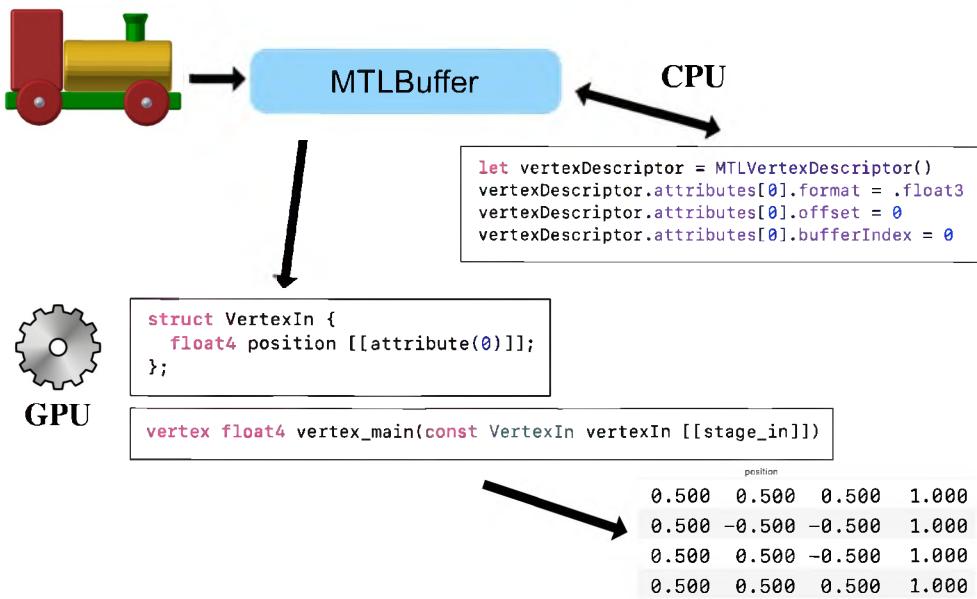
Going through the code:

1. Create a struct **VertexIn** to describe the vertex attributes that match the vertex descriptor you set up earlier. In this case, just **position**.
2. Implement a vertex shader, **vertex\_main**, that takes in **VertexIn** structs and returns vertex positions as **float4** types.

Remember that vertices are indexed in the vertex buffer. The vertex shader gets the current vertex index via the **[[stage\_in]]** attribute and unpacks the **VertexIn** structure cached for the vertex at the current index.

Compute Units can process (at one time) batches of vertices up to their maximum number of shader cores. This batch can fit entirely in the CU cache and vertices can thus be reused as needed. The batch will keep the CU busy until the processing is done but other CUs should become available to process the next batch.

As soon as the vertex processing is done, the cache is cleared for the next batches of vertices. At this point, vertices are now ordered and grouped, ready to be sent to the primitive assembly stage.



### Vertex processing

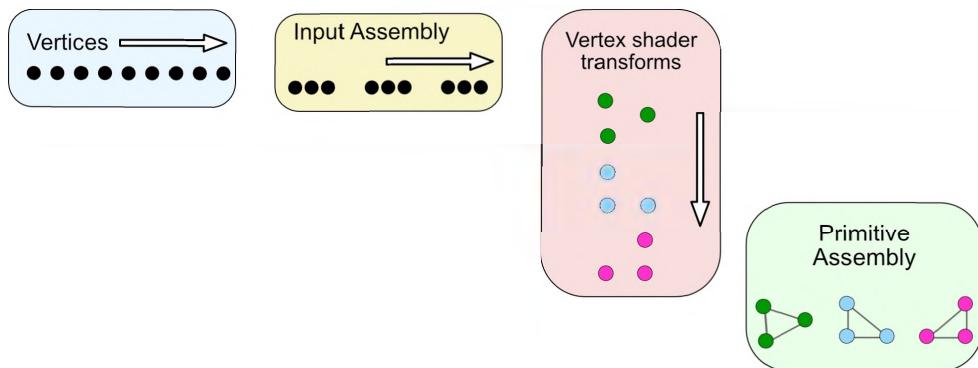
To recap, the CPU sent the GPU a vertex buffer that you created from the model's mesh. You configured the vertex buffer using a vertex descriptor that tells the GPU how the vertex data is structured. On the GPU, you created a structure to encapsulate the vertex attributes. The vertex shader takes in this structure as a function argument, and through the `[[stage_in]]` qualifier, acknowledges that `position` comes from the CPU via the `[[attribute(0)]]` position in the vertex buffer. The vertex shader then processes all of the vertices and returns their positions as a `float4`.

**Note:** When you use a vertex descriptor with attributes, you don't have to match types. The `MTLBuffer` `position` is a `float3`, whereas `VertexIn` defines the `position` as a `float4`.

A special hardware unit known as the **Distributer** sends the grouped blocks of vertices on to the **Primitive Assembly** stage.

## 3 - Primitive Assembly

The previous stage sent processed vertices grouped into blocks of data to this stage. The important thing to keep in mind is that vertices belonging to the same geometrical shape (primitive) are always in the same block. That means that the one vertex of a point, or the two vertices of a line, or the three vertices of a triangle, will always be in the same block, hence a second block fetch isn't necessary.



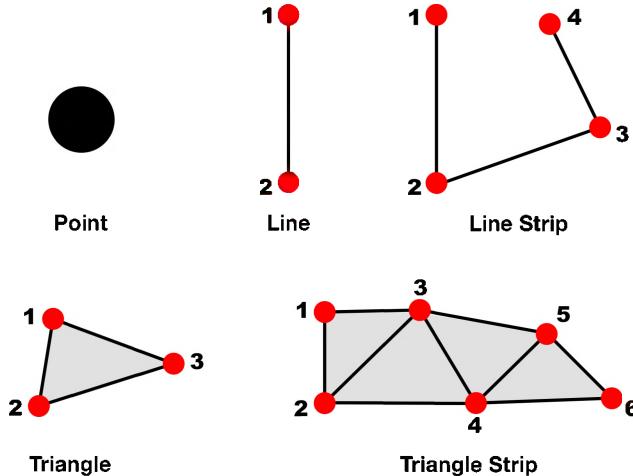
*Primitive assembly*

Along with vertices, the CPU also sends vertex connectivity information when it issues the **draw call** command, like this:

```
renderEncoder.drawIndexedPrimitives(
    type: .triangle,
    indexCount: submesh.indexCount,
    indexType: submesh.indexType,
    indexBuffer: submesh.indexBuffer.buffer,
    indexBufferOffset: 0)
```

The first argument of the draw function contains the most important information about vertex connectivity. In this case, it tells the GPU that it should draw triangles from the vertex buffer it sent.

The Metal API provides five primitive types:

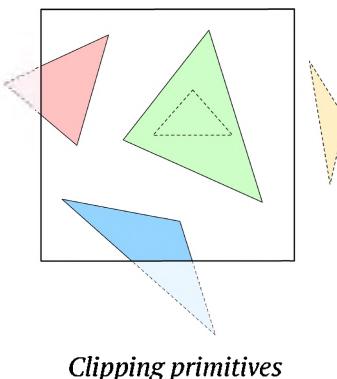


*The primitive types*

- **point:** For each vertex, rasterize a point. You can specify the size of a point that has the attribute `[[point_size]]` in the vertex shader.
- **line:** For each pair of vertices, rasterize a line between them. If a vertex was already included in a line, it cannot be included again in other lines. The last vertex is ignored if there are an odd number of vertices.
- **lineStrip:** Same as a simple line, except that the line strip connects all adjacent vertices and forms a poly-line. Each vertex (except the first) is connected to the previous vertex.
- **triangle:** For every sequence of three vertices, rasterize a triangle. The last vertices are ignored if they cannot form another triangle.
- **triangleStrip:** Same as a simple triangle, except adjacent vertices can be connected to other triangles as well.

There is one more primitive type known as a **patch**, but this needs special treatment. You'll read more about patches in Chapter 19, "Tessellation & Terrains".

As you read in the previous chapter, the pipeline specifies the winding order of the vertices. If the winding order is counter-clockwise, and the triangle vertex order is counter-clockwise, the vertices are front-faced; otherwise, the vertices are back-faced and can be culled since you can't see their color and lighting. Primitives are culled when they're entirely occluded by other primitives. However, if they're only partially off-screen, they'll be clipped.



*Clipping primitives*

For efficiency, you should set winding order and enable back-face culling in the pipeline state.

At this point, primitives are fully assembled from connected vertices and are ready to move on to the rasterizer.

## 4 - Rasterization

There are two modern rendering techniques currently evolving on separate paths but sometimes used together: **ray tracing** and **rasterization**. They are quite different, and both have pros and cons. Ray tracing — which you'll read more about in Chapter 27, "Rendering With Rays" — is preferred when rendering content that is static and far away, while rasterization is preferred when the content is closer to the camera and more dynamic.

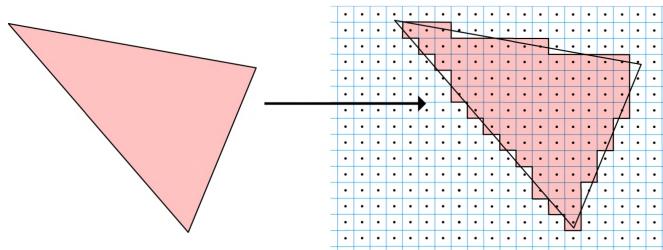
With ray tracing, for each pixel on the screen, it sends a ray into the scene to see if there's an intersection with an object. If yes, change the pixel color to that object's color, but only if the object is closer to the screen than the previously saved object for the current pixel.

Rasterization works the other way around. For each object in the scene, send rays back into the screen and check which pixels are covered by the object. Depth information is kept the same way as for ray tracing, so it will update the pixel color if the current object is closer than the previously saved one.

At this point, all connected vertices sent from the previous stage need to be represented on a two-dimensional grid using their X and Y coordinates. This step is known as the **triangle setup**. Here is where the rasterizer needs to calculate the slope or steepness of the line segments between any two vertices. When the three slopes for the three vertices are known, the triangle can be formed from these three edges.

Next, a process known as **scan conversion** runs on each line of the screen to look for intersections and to determine what's visible and what's not. To draw on the screen at this point, you need only the vertices and the slopes they determine.

The scan algorithm determines if all the points on a line segment or all the points inside of a triangle are visible, in which case the triangle is filled with color entirely.



*Rasterizing triangles*

For mobile devices, the rasterization takes advantage of the tiled architecture of PowerVR GPUs by rasterizing the primitives on a 32x32 tile grid in parallel. In this case, 32 is the number of screen pixels assigned to a tile, but this size perfectly fits the number of cores in a **USC**.

What if one object is behind another object? How can the rasterizer determine which object to render? This hidden surface removal problem can be solved by using stored depth information (early-Z testing) to determine whether each point is in front of other points in the scene.

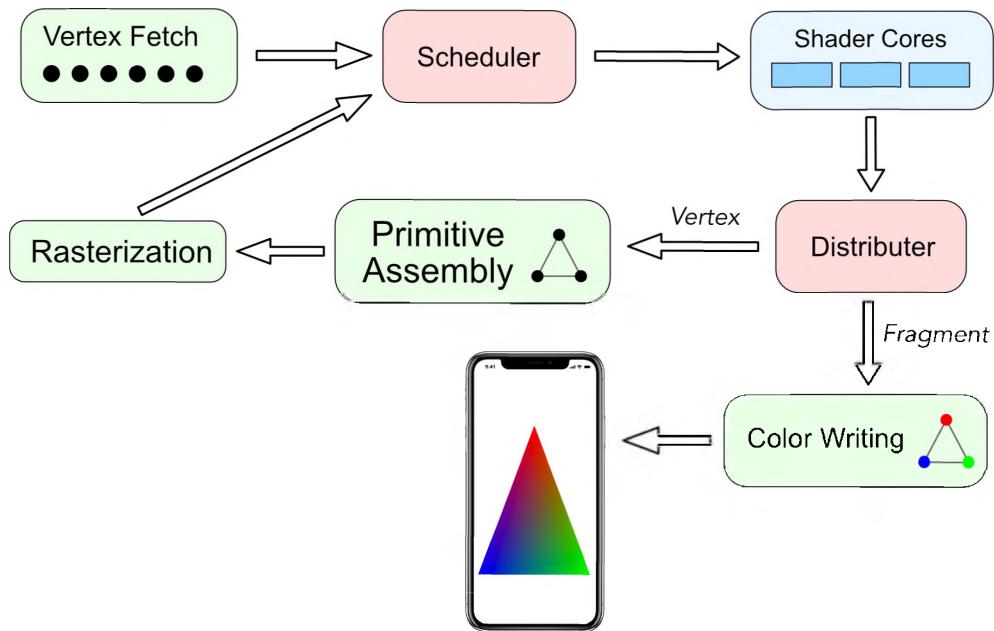
After rasterization is finished, three more specialized hardware units take the stage:

- A buffer known as **Hierarchical-Z** is responsible for removing fragments that were marked for culling by the rasterizer.
- The **Z and Stencil Test** unit then removes non-visible fragments by comparing them against the depth and stencil buffer.
- Finally, the **Interpolator** unit takes the remaining visible fragments and generates fragment attributes from the assembled triangle attributes.

At this point, the **Scheduler** unit, again, dispatches work to the shader cores, but this time it's the rasterized fragments sent for **Fragment Processing**.

## 5 - Fragment Processing

Time for a quick review of the pipeline.



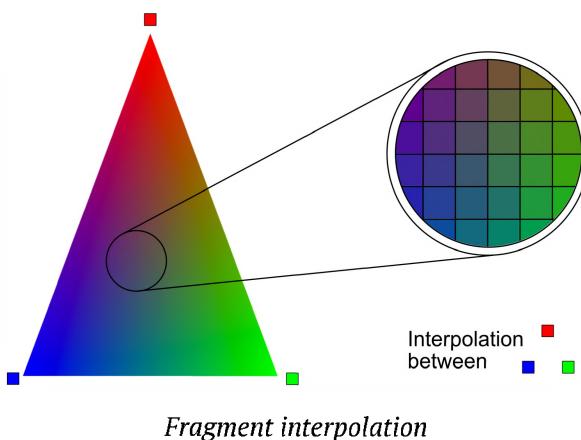
*Fragment Processing*

- The **Vertex Fetch** unit grabs vertices from the memory and passes them to the **Scheduler** unit.
- The **Scheduler** unit knows which shader cores are available, so it dispatches work on them.
- After the work is done, the **Distributer** unit knows if this work was **Vertex or Fragment Processing**. If the work was **Vertex Processing**, it sends the result to the **Primitive Assembly** unit. This path continues to the **Rasterization** unit, and then back to the **Scheduler** unit. If the work was **Fragment Processing**, it sends the result to the **Color Writing** unit.
- Finally, the colored pixels are sent back to the memory.

The primitive processing in the previous stages is sequential because there's only one **Primitive Assembly** unit and one **Rasterization** unit. However, as soon as fragments reach the **Scheduler** unit, work can be *forked* (divided) into many tiny parts, and each part is given to an available shader core.

Hundreds or even thousands of cores are now doing parallel processing. When the work is complete, the results will be *joined* (merged) and sent to the memory, again sequentially.

The fragment processing stage is another programmable stage. You create a fragment shader function that will receive the lighting, texture coordinate, depth and color information that the vertex function outputs. The fragment shader output is a single color for that fragment. Each of these fragments will contribute to the color of the final pixel in the framebuffer. All of the attributes are interpolated for each fragment.



*Fragment interpolation*

For example, to render this triangle, the vertex function would process three vertices with the colors red, green and blue. As the diagram shows, each fragment that makes up this triangle is interpolated from these three colors. Linear interpolation simply averages the color at each point on the line between two endpoints. If one endpoint has red color, and the other has green color, the midpoint on the line between them will be yellow. And so on.

The interpolation equation is parametric and has this form, where parameter **p** is the percentage (or a range from 0 to 1) of a color's presence:

```
newColor = p * oldColor1 + (1 - p) * oldColor2
```

Color is easy to visualize, but the other vertex function outputs are also similarly interpolated for each fragment.

**Note:** If you don't want a vertex output to be interpolated, add the attribute `[[flat]]` to its definition.

## Creating a Fragment Shader

- In `Shaders.Metal`, add the fragment function to the end of the file:

```
fragment float4 fragment_main() {  
    return float4(1, 0, 0, 1);  
}
```

This is the simplest fragment function possible. You return the interpolated color red in the form of a `float4`. All the fragments that make up the cube will be red. The GPU takes the fragments and does a series of post-processing tests:

- **alpha-testing** determines which opaque objects are drawn (and which are not) based on depth testing.
- In the case of translucent objects, **alpha-blending** will combine the color of the new object with that already saved in the color buffer previously.
- **scissor testing** checks whether a fragment is inside of a specified rectangle; this test is useful for masked rendering.
- **stencil testing** checks how the stencil value in the framebuffer where the fragment is stored, compares to a specified value we choose.
- In the previous stage **early-Z testing** ran; now a **late-Z testing** is done to solve more visibility issues; stencil and depth tests are also useful for ambient occlusion and shadows.
- Finally, **antialiasing** is also calculated here so that final images that get to the screen do not look jagged.

You'll learn more about post-processing tests in Chapter 20, "Fragment Post-Processing".

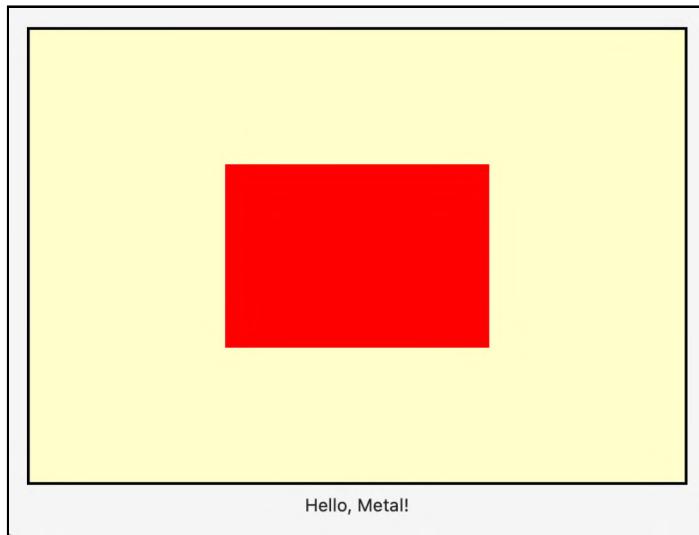
## 6 - Framebuffer

As soon as fragments have been processed into pixels, the **Distributer** unit sends them to the **Color Writing** unit. This unit is responsible for writing the final color in a special memory location known as the **framebuffer**. From here, the view gets its colored pixels refreshed every frame. But does that mean the color is written to the framebuffer while being displayed on the screen?

A technique known as **double-buffering** is used to solve this situation. While the first buffer is being displayed on the screen, the second one is updated in the background. Then, the two buffers are swapped, and the second one is displayed on the screen while the first one is updated, and the cycle continues.

Whew! That was a lot of hardware information to take in. However, the code you've written is what every Metal renderer uses, and despite just starting out, you should begin to recognize the rendering process when you look at Apple's sample code.

- Build and run the app, and you'll see a beautifully rendered red cube:



*A rendered cube*

Notice how the cube is not square. Remember that Metal uses **Normalized Device Coordinates (NDC)** that is  $-1$  to  $1$  on the X axis. Resize your window, and the cube will maintain a size relative to the size of the window. In Chapter 6, “Coordinate Spaces”, you’ll learn how to position objects precisely on the screen.

What an incredible journey you’ve had through the rendering pipeline. In the next chapter, you’ll explore vertex and fragment shaders in greater detail.

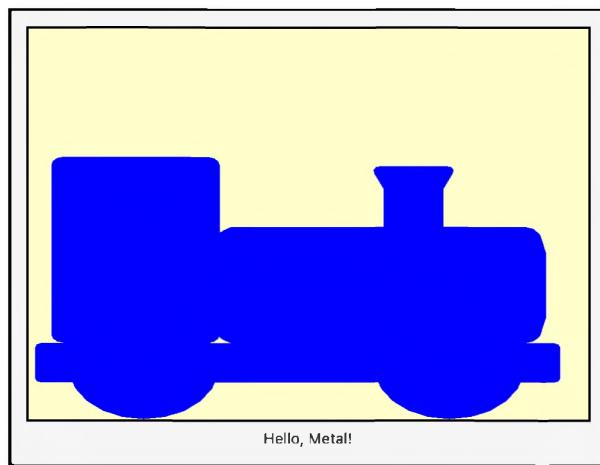
## Challenge

Using the `train.usd` model in the resources folder for this project, replace the cube with this train. When importing the model, be sure to select **Create Groups** and remember to add the model to both targets.

Instead of changing the model's vertical position in the SceneKit editor, change it in the vertex function using this code:

```
float4 position = vertexIn.position;  
position.y -= 1.0;
```

Finally, color your train blue.



*Challenge result*

Refer to the previous chapter for asset loading and the vertex descriptor code if you need help. The finished code for this challenge is in the project challenge directory for this chapter.

## Key Points

- CPUs are best for processing sequential tasks fast, whereas GPUs excel at processing small tasks synchronously.
- SwiftUI is a great host for MTKViews, as you can layer UI elements easily.
- Separate Metal tasks where you can to the initialize phase. Initialize the device, command queues, pipeline states and model data buffers once at the start of your app.
- Each frame, create a command buffer and one or more command encoders.
- GPU architecture allows for a strict pipeline. Configure this using PSOs (pipeline state objects).
- There are two programmable stages in a simple rendering GPU pipeline. You calculate vertex positions using the vertex shader, and calculate the color that appears on the screen using the fragment shader.

# Chapter 4: The Vertex Function

So far, you've worked your way through 3D models and the graphics pipeline. Now, it's time to look at the first of two programmable stages in Metal, the vertex stage — and more specifically, the vertex function.



# Shader Functions

There are three types of shader functions:

- **Vertex function:** Calculates the position of a vertex.
- **Fragment function:** Calculates the color of a fragment.
- **Kernel function:** Used for general-purpose parallel computations, such as image processing.

In this chapter, you'll focus only on the vertex function. In Chapter 7, “The Fragment Function”, you'll explore how to control the color of each fragment. And in Chapter 16, “GPU Compute Programming”, you'll discover how to use parallel programming with multiple threads to write to buffers and textures.

By now, you should be familiar with vertex descriptors, and how to use them to describe how to lay out the vertex attributes from your loaded 3D model. To recap:

- **MDLVertexDescriptor:** You use a Model I/O vertex descriptor to read in the .obj file. Model I/O creates buffers with the desired layout of attributes, such as position, normals and texture coordinates.
- **MTLVertexDescriptor:** You use a Metal vertex descriptor when creating the pipeline state. The GPU vertex function uses the `[[stage_in]]` attribute to match the incoming data with the vertex descriptor in the pipeline state.

As you work through this chapter, you'll construct your own vertex mesh and send vertices to the GPU *without* using a vertex descriptor. You'll learn how to manipulate these vertices in the vertex function, and then you'll upgrade to using a vertex descriptor. In the process, you'll see how using Model I/O to import your meshes does a lot of the heavy lifting for you.

## The Starter Project

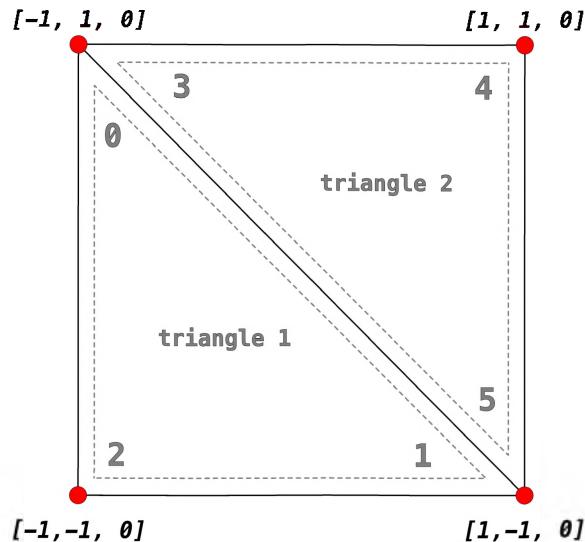
- Open the starter project.

This SwiftUI project contains a reduced `Renderer` so that you can add your own mesh, and the shader functions are bare-bones so that you can build them up. You're not doing any drawing yet, so there's nothing to see when you run the app.



## Rendering a Quad

You create a quad using two triangles — and each triangle has three vertices, for a total of six vertices.



*A quad mesh*

► Create a new Swift file named **Quad.swift**.

► Replace the existing code with:

```
import MetalKit

struct Quad {
    var vertices: [Float] = [
        -1, 1, 0,      // triangle 1
        1, -1, 0,
        -1, -1, 0,
        -1, 1, 0,      // triangle 2
        1, 1, 0,
        1, -1, 0
    ]
}
```

As you know, a vertex is made of an x, y and z value. Each group of three `Floats` in `vertices` describes one vertex. Here, the **winding order** of the points is clockwise, which is important.

- Add a new vertex buffer property to Quad and initialize it:

```
let vertexBuffer: MTLBuffer

init(device: MTLDevice, scale: Float = 1) {
    vertices = vertices.map {
        $0 * scale
    }
    guard let vertexBuffer = device.makeBuffer(
        bytes: &vertices,
        length: MemoryLayout<Float>.stride * vertices.count,
        options: []) else {
        fatalError("Unable to create quad vertex buffer")
    }
    self.vertexBuffer = vertexBuffer
}
```

With this code, you initialize the Metal buffer with the array of vertices. You multiply each vertex by `scale`, which lets you set the size of the quad during initialization.

- Open `Renderer.swift`, and add a new property for the quad mesh:

```
lazy var quad: Quad = {
    Quad(device: Renderer.device, scale: 0.8)
}()
```

Here, you initialize `quad` with `Renderer`'s device — and because you initialize `device` in `init(metalView:)`, you must initialize `quad` lazily. You also resize the quad so that you can see it properly. (If you were to leave the `scale` at the default of 1.0, the quad would cover the entire screen. Covering the screen is useful for full-screen drawing since you can only draw fragments where you're rendering geometry.)

- In `draw(in:)`, after `// do drawing here`, add:

```
renderEncoder.setVertexBuffer(
    quad.vertexBuffer,
    offset: 0,
    index: 0)
```

You create a command on the render command encoder to set the vertex buffer in the buffer argument table at index 0.

- Add the draw call:

```
renderEncoder.drawPrimitives(
    type: .triangle,
    vertexStart: 0,
    vertexCount: quad.vertices.count)
```



Here, you draw the quad's six vertices.

► Open **Shaders.metal**.

► Replace the vertex function with:

```
vertex float4 vertex_main(  
    constant float3 *vertices [[buffer(0)]],  
    uint vertexID [[vertex_id]])  
{  
    float4 position = float4(vertices[vertexID], 1);  
    return position;  
}
```

There's an error with this code, which you'll observe and fix shortly.

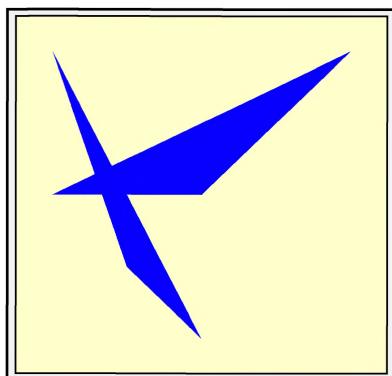
The GPU performs the vertex function for each vertex. In the draw call, you specified that there are six vertices. So, the vertex function will perform six times.

When you pass a pointer into the vertex function, you must specify an address space, either `constant` or `device`. `constant` is optimized for accessing the same variable over several vertex functions in parallel. `device` is best for accessing different parts of a buffer over the parallel functions — such as when using a buffer with points and color data interleaved.

`[[vertex_id]]` is an attribute qualifier that gives you the current vertex. You can use this as an entry into the array held in `vertices`.

You might notice that you're sending the GPU a buffer that you filled with an array of `FLOATs`. In the vertex function, you read the same buffer as an array of `float3s`, leading to an error in the display.

► Build and run.



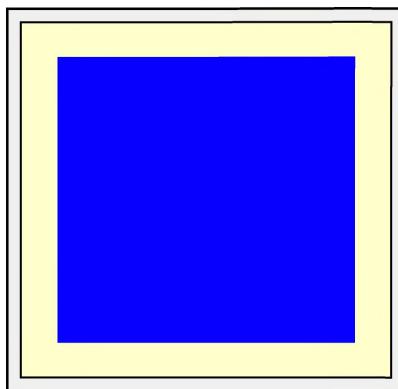
A rendering error

Although you might get a different render, the vertices are in the wrong position because a `float3` type takes up more memory than three `Float` types. The SIMD `float3` type is padded and takes up the same memory as the `float4` type, which is 16 bytes. Changing this parameter to a `packed_float3` will fix the error since a `packed_float3` takes up 12 bytes.

**Note:** You can check the sizes of types in the Metal Shading Language Specification at <https://apple.co/2UT993x>.

In the vertex function, change `float3` in the first parameter to `packed_float3`.

- Build and run.



*The rendering error corrected*

The quad now displays correctly.

Alternatively, you could have defined the `Float` array `vertices` as an array of `simd_float3`. In that case, you'd use `float3` in the vertex function, as both types take 16 bytes. However, sending 16 bytes per vertex is slightly less efficient than sending 12 bytes per vertex.

## Calculating Positions

Metal is all about gorgeous graphics and fast, smooth animation. As a next step, you'll make your quad move up and down the screen. To do this, you'll have a timer that updates every frame, and the position of each vertex will depend on this timer.

The vertex function is where you update vertex positions, so you'll send the timer data to the GPU.

- Open `Renderer.swift`, and add a new property to `Renderer`:

```
var timer: Float = 0
```

► In `draw(in:)`, right before:

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

► Add the following code:

```
// 1  
timer += 0.005  
var currentTime = sin(timer)  
// 2  
renderEncoder.setVertexBytes(  
    &currentTime,  
    length: MemoryLayout<Float>.stride,  
    index: 11)
```

Let's have a closer look:

1. For every frame, you update the timer. You want your cube to move up and down the screen, so you'll use a value between -1 and 1. Using `sin()` is a great way to achieve this balance as sine values are always -1 to 1. You can change the speed of your animation by changing the value that you add to this timer for each frame.
  2. If you're only sending a small amount of data — say less than 4KB — to the GPU, `setVertexBytes(_ : length : index :)` is an alternative to setting up an `MTLBuffer`. Here, you set `currentTime` to index 11 in the buffer argument table. Keeping buffers 1 through 10 for vertex attributes — such as vertex positions — helps you to remember which buffers hold what data.

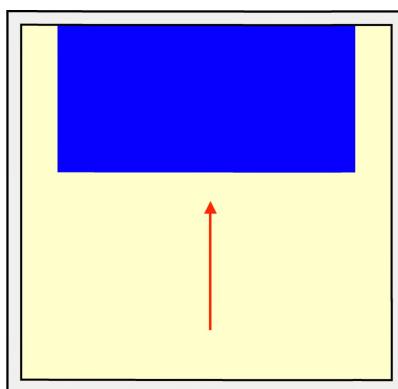
- Open **Shaders.metal**, and replace the vertex function:

```
vertex float4 vertex_main(
    constant packed_float3 *vertices [[buffer(0)]],
    constant float &timer [[buffer(11)]],
    uint vertexID [[vertex_id]])
{
    float4 position = float4(vertices[vertexID], 1);
    position.y += timer;
    return position;
}
```

You receive the single value `timer` as a `float` in buffer 11. You add the timer value to the `y` position and return the new position from the function.

In the next chapter, you'll start learning how to project vertices into 3D space using matrix multiplication. But, you don't always need matrix multiplication to move vertices; here, you can achieve the translation of the position in `y` using simple addition.

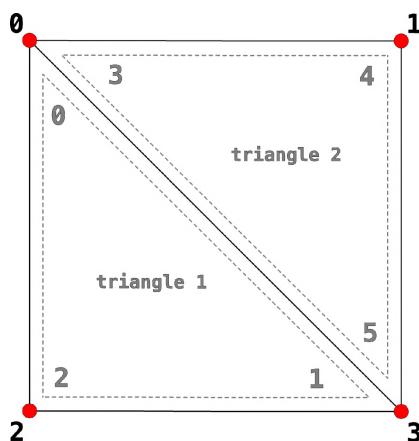
► Build and run the app, and you'll see a lovely animated quad.



An animated quad

## More Efficient Rendering

Currently, you're using six vertices to render two triangles.



The mesh of two triangles

Of those vertices, 0 and 3 are in the same position, as are 1 and 5. If you render a mesh with thousands — or even millions of vertices, it's important to reduce duplication as much as possible. You can do this with indexed rendering.

Create a structure of only unique positions, and then use indices to get the right position for a vertex.

- Open **Quad.swift**, and rename `vertices` to `oldVertices`.
- Add the following structures to `Quad`:

```
var vertices: [Float] = [
    -1, 1, 0,
    1, 1, 0,
    -1, -1, 0,
    1, -1, 0
]

var indices: [UInt16] = [
    0, 3, 2,
    0, 1, 3
]
```

`vertices` now holds the unique four points of the quad in any order. `indices` holds the index of each vertex in the correct vertex order. Refer to `oldVertices` to make sure your indices are correct.

- Add a new Metal buffer to hold `indices`:

```
let indexBuffer: MTLBuffer
```

- At the end of `init(device:scale:)`, add:

```
guard let indexBuffer = device.makeBuffer(
    bytes: &indices,
    length: MemoryLayout<UInt16>.stride * indices.count,
    options: []) else {
    fatalError("Unable to create quad index buffer")
}
self.indexBuffer = indexBuffer
```

You create the index buffer the same way you did the vertex buffer.

- Open **Renderer.swift**, and in `draw(in:)` before the draw call, add:

```
renderEncoder.setVertexBuffer(
    quad.indexBuffer,
    offset: 0,
```



```
    index: 1)
```

Here, you send the index buffer to the GPU.

- Change the draw call to:

```
renderEncoder.drawPrimitives(  
    type: .triangle,  
    vertexStart: 0,  
    vertexCount: quad.indices.count)
```

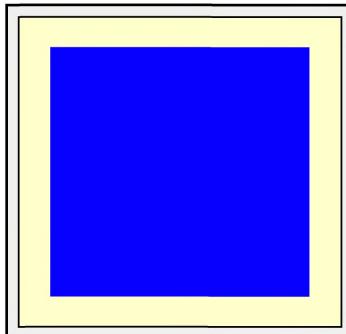
Use the index count for the number of vertices to render; not the vertex count.

- Open **Shaders.metal**, and change the vertex function to:

```
vertex float4 vertex_main(  
    constant packed_float3 *vertices [[buffer(0)]],  
    constant ushort *indices [[buffer(1)]],  
    constant float &timer [[buffer(11)]],  
    uint vertexID [[vertex_id]])  
{  
    ushort index = indices[vertexID];  
    float4 position = float4(vertices[index], 1);  
    return position;  
}
```

Here, `vertexID` is the index into the buffer holding the quad indices. You use the value in the `indices` buffer to index the correct vertex in the `vertices` buffer.

- Build and run. Sure, your quad is positioned the same way as before, but now you're sending less data to the GPU.



*Indexed mesh*

From the number of entries in arrays, it might appear as if you're actually sending more data — but you're not! The memory footprint of `oldVertices` is 72 bytes, whereas the footprint of `vertices + indices` is 60 bytes.



# Vertex Descriptors

A more efficient draw call is available when you use indices for rendering vertices. However, you first need to set up a vertex descriptor in the pipeline.

It's always a good idea to use vertex descriptors, as most often, you won't only send positions to the GPU. You'll also send attributes such as normals, texture coordinates and colors. When you can lay out your own vertex data, you have more control over how your engine handles model meshes.

► Create a new Swift file named **VertexDescriptor.swift**.

► Replace the code with:

```
import MetalKit

extension MTLVertexDescriptor {
    static var defaultLayout: MTLVertexDescriptor {
        let vertexDescriptor = MTLVertexDescriptor()
        vertexDescriptor.attributes[0].format = .float3
        vertexDescriptor.attributes[0].offset = 0
        vertexDescriptor.attributes[0].bufferIndex = 0

        let stride = MemoryLayout<Float>.stride * 3
        vertexDescriptor.layouts[0].stride = stride
        return vertexDescriptor
    }
}
```

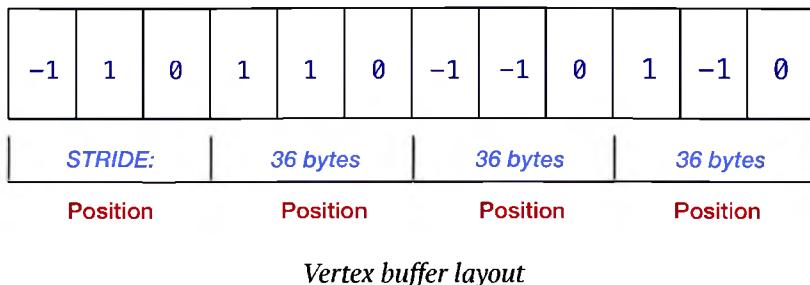
Here, you set up a vertex layout that has only one attribute. This attribute describes the position of each vertex.

A vertex descriptor holds arrays of attributes and buffer layouts.

- **attributes:** For each attribute, you specify the type format and offset in bytes of the first item from the beginning of the buffer. You also specify the index of the buffer that holds the attribute.
- **buffer layout:** You specify the length of the stride of all attributes combined in each buffer. It may be confusing here as you're using index 0 to index into both `layouts` and `attributes`, but the `layouts` index 0 corresponds to the `bufferIndex` 0 used by `attributes`.

**Note:** `stride` describes how many bytes are between each instance. Due to internal padding and byte alignment, this value can be different from `size`. For an excellent explanation of `size`, `stride` and alignment, check out Greg Heo's article at <https://bit.ly/2V3gBJl>.

To the GPU, the `vertexBuffer` now looks like this:



- Open `Renderer.swift`, and locate where you create the pipeline state in `init(metalView:)`.
- Before creating the pipeline state in `do {}`, add the following code to the pipeline state descriptor:

```
pipelineDescriptor.vertexDescriptor =  
    MTLVertexDescriptor.defaultLayout
```

The GPU will now expect vertices in the format described by this vertex descriptor.

- In `draw(in:)`, remove:

```
renderEncoder.setVertexBuffer(  
    quad.indexBuffer,  
    offset: 0,  
    index: 1)
```

You'll include the index buffer in the draw call.

- Change the draw call to:

```
renderEncoder.drawIndexedPrimitives(  
    type: .triangle,  
    indexCount: quad.indices.count,  
    indexType: .uint16,  
    indexBuffer: quad.indexBuffer,  
    indexBufferOffset: 0)
```

This draw call expects the index buffer to use `UInt16`, which is how you described your indices array in `Quad`. You don't explicitly send `quad.indexBuffer` to the GPU because this draw call will do it for you.

- Open `Shaders.metal`.

- Replace the vertex function with:

```
vertex float4 vertex_main(  
    float4 position [[attribute(0)]] [[stage_in]],  
    constant float &timer [[buffer(11)]])  
{  
    return position;  
}
```

You did all the heavy lifting for the layout on the Swift side, so that takes the size of the vertex function way down. :]

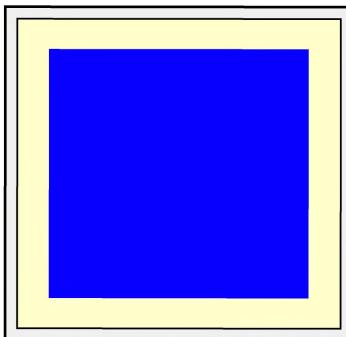
You describe each per-vertex input with the `[[stage_in]]` attribute. The GPU now looks at the pipeline state's vertex descriptor.

`[[attribute(0)]]` is the attribute in the vertex descriptor that describes the position. Even though you defined your original vertex data as three `FLOATs`, you can define the position as `float4` here. The GPU can make the conversion.

It's worth noting that when the GPU adds the `w` information to the `xyz` position, it adds `1.0`. As you'll see in the following chapters, this `w` value is quite important during rasterization.

The GPU now has all of the information it needs to calculate the position for each vertex.

- Build and run the app to ensure that everything still works. The resulting render will be the same as before.



*Rendering using a vertex descriptor*

## Adding Another Vertex Attribute

You probably won't ever have just one attribute, so let's add a color attribute for each vertex.

You have a choice whether to use two buffers or interleave the color between each vertex position. If you choose to interleave, you'll set up a structure to hold position and color. In this example, however, it's easier to add a new colors buffer to match each vertex.

- Open `Quad.swift`, and add the new array:

```
var colors: [simd_float3] = [
    [1, 0, 0], // red
    [0, 1, 0], // green
    [0, 0, 1], // blue
    [1, 1, 0] // yellow
]
```

You now have four RGB colors to match the four vertices.

- Create a new buffer property:

```
let colorBuffer: MTLBuffer
```

- At the end of `init(device:scale:)`, add:

```
guard let colorBuffer = device.makeBuffer(
```

```

bytes: &colors,
length: MemoryLayout<simd_float3>.stride * indices.count,
options: []) else {
    fatalError("Unable to create quad color buffer")
}
self.colorBuffer = colorBuffer

```

You initialize `colorBuffer` the same way as the previous two buffers.

- Open **Renderer.swift**, and in `draw(in:)` right before the draw call, add:

```

renderEncoder.setVertexBuffer(
    quad.colorBuffer,
    offset: 0,
    index: 1)

```

You send the color buffer to the GPU using buffer index 1, which must match the index in the vertex descriptor layout.

- Open **VertexDescriptor.swift**, and add the following code to `defaultLayout` before return:

```

vertexDescriptor.attributes[1].format = .float3
vertexDescriptor.attributes[1].offset = 0
vertexDescriptor.attributes[1].bufferIndex = 1
vertexDescriptor.layouts[1].stride =
    MemoryLayout<simd_float3>.stride

```

Here, you describe the layout of the color buffer in buffer index 1.

- Open **Shaders.metal**.
- You can only use `[[stage_in]]` on one parameter, so create a new structure:

```

struct VertexIn {
    float4 position [[attribute(0)]];
    float4 color [[attribute(1)]];
};

```

- Change the vertex function to:

```

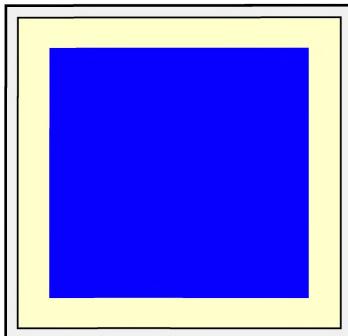
vertex float4 vertex_main(
    VertexIn in [[stage_in]],
    constant float &timer [[buffer(11)]])
{
    return in.position;
}

```



This code is still short and concise. The GPU knows how to retrieve position and color from the buffers because of the `[[attribute(n)]]` qualifier in the structure, which looks at the pipeline state's vertex descriptor.

- Build and run to ensure your blue quad still renders.



*Quad with two attributes*

The fragment function determines the color of each rendered fragment. You need to pass the vertex's color to the fragment function. You'll learn more about the fragment function in Chapter 7, "The Fragment Function".

- Still in `Shaders.metal`, add this structure:

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

Instead of returning just the position from the vertex function, you'll now return both position and color. You specify a `position` attribute to let the GPU know which property in this structure is the position.

- Replace the vertex function with:

```
vertex VertexOut vertex_main(  
    VertexIn in [[stage_in]],  
    constant float &timer [[buffer(11)]]) {  
    VertexOut out {  
        .position = in.position,  
        .color = in.color  
    };  
    return out;  
}
```

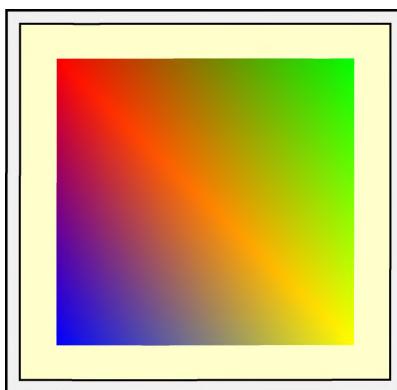
You now return a `VertexOut` instead of a `float4`.

- Change the fragment function to:

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

The `[[stage_in]]` attribute indicates that the GPU should take the `VertexOut` output from the vertex function and match it with the rasterized fragments. Here, you return the vertex color. Remember from Chapter 3, “The Rendering Pipeline”, that each fragment’s input gets interpolated.

- Build and run the app, and you’ll see the quad rendered with beautiful colors.



*Interpolated vertex colors*

## Rendering Points

Instead of rendering triangles, you can render points and lines.

- Open `Renderer.swift`, and in `draw(in:)`, change:

```
renderEncoder.drawIndexedPrimitives(  
    type: .triangle,
```

- To:

```
renderEncoder.drawIndexedPrimitives(  
    type: .point,
```

If you build and run now, the GPU will render the points, but it doesn’t know what point size to use, so it flickers over various point sizes. To fix this problem, you’ll also return a point size when returning data from the vertex function.

- Open **Shaders.metal**, and add this property to **VertexOut**:

```
float pointSize [[point_size]];
```

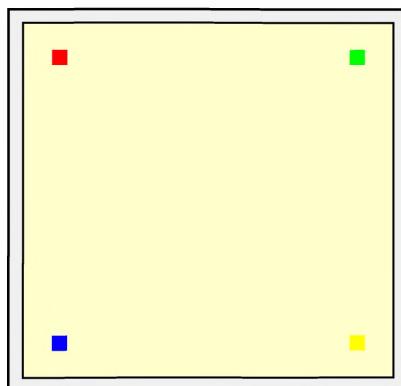
The `[[point_size]]` attribute will tell the GPU what point size to use.

- Replace the initialization of **out** with:

```
VertexOut out {  
    .position = in.position,  
    .color = in.color,  
    .pointSize = 30  
};
```

Here, you assign the point size of 30.

- Build and run to see your points rendered with their vertex color:



*Rendering points*

## Challenge

So far, you've sent vertex positions to the GPU in an array buffer. But this isn't entirely necessary. All the GPU needs to know is how many vertices to draw. Your challenge is to remove the vertex and index buffers, and draw 50 points in a circle. Here's an overview of the steps you'll need to take, along with some code to get you started:

1. In `Renderer`, remove the vertex descriptor from the pipeline.
2. Replace the draw call in `Renderer` so that it doesn't use indices but does draw 50 vertices.
3. In `draw(in:)`, remove all of the `setVertexBuffer` commands.
4. The GPU will need to know the total number of points, so send this value the same way you did `timer` in buffer 0.
5. Replace the vertex function with:

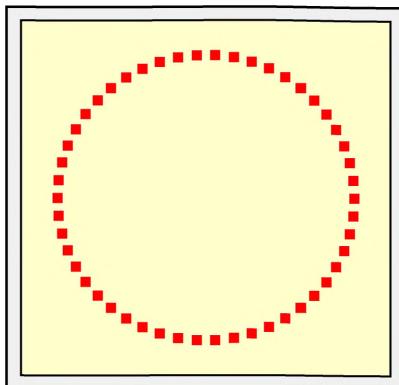
```
vertex VertexOut vertex_main(
    constant uint &count [[buffer(0)]],
    constant float &timer [[buffer(11)]],
    uint vertexID [[vertex_id]])
{
    float radius = 0.8;
    float pi = 3.14159;
    float current = float(vertexID) / float(count);
    float2 position;
    position.x = radius * cos(2 * pi * current);
    position.y = radius * sin(2 * pi * current);
    VertexOut out {
        .position = float4(position, 0, 1),
        .color = float4(1, 0, 0, 1),
        .pointSize = 20
    };
    return out;
}
```

Remember, this is an exercise to help you understand how to position points on the GPU *without* holding any equivalent data on the Swift side. So, don't worry too much about the math. You can use the sine and cosine of the current vertex ID to plot the point around a circle.



Notice that there's no built-in value for pi on the GPU.

You'll see your 50 points plotted into a circle.



*Points in a circle*

Try animating the points by adding `timer` to `current`.

If you have any difficulties, you can find the solution in the project challenge directory for this chapter.

## Key Points

- The vertex function's fundamental task is positioning vertices. When you render a model, you send the GPU the model's vertices in its original position. The vertex shader will then reposition those vertices to the correct spot in your 3D world.
- Shader code uses attributes such as `[[buffer(0)]]` and `[position]` extensively. To find out more about these attributes, refer to the Metal Shading Language specification document (<https://apple.co/3hPTbjQ>).
- You can pass any data in an `MTLBuffer` to the GPU using `setVertexBuffer(_:_offset:_index:_)`. If the data is less than 4KB, you don't have to set up a buffer; you can, instead, pass a structure using `setVertexBytes(_:_length:_index:_)`.
- When possible, use indexed rendering. With indexed rendering, you pass less data to the GPU — and memory bandwidth is a major bottleneck.
- When possible, use vertex descriptors. With vertex descriptors, the GPU knows the format of the data being passed, and you'll get fewer errors in your code when you change a type on the Swift side and forget to change the shader function.

# Chapter 5: 3D Transformations

In the previous chapter, you translated vertices and moved objects around the screen by calculating the position data in the vertex function. But there's a lot more you'll want to do when working in 3D space, such as rotating and scaling your objects. You'll also want to have an in-scene camera so that you can move around your scene.

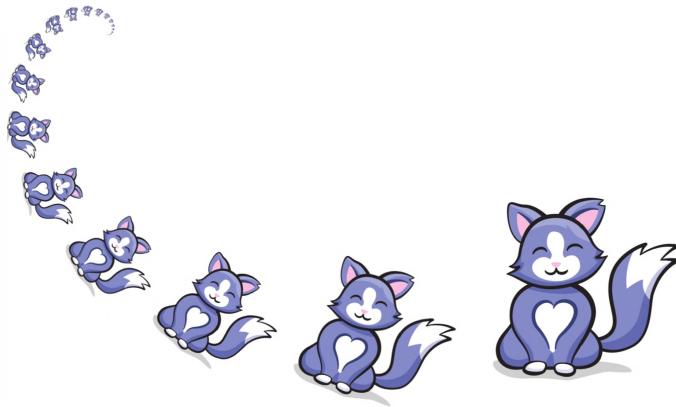
To move, scale and rotate a triangle, you'll use matrices — and once you've mastered one triangle, it's a cinch to rotate a model with thousands of triangles at once!

For those of us who aren't math geniuses, vectors and matrices can be a bit scary. Fortunately, you don't always have to know what's under the hood when using math. To help, this chapter focuses not on the math, but the matrices. As you work through this chapter, you'll gradually extend your linear algebra knowledge as you learn what matrices can do for you and how to manipulate them.



# Transformations

Look at the following picture.



*Affine Transformations*

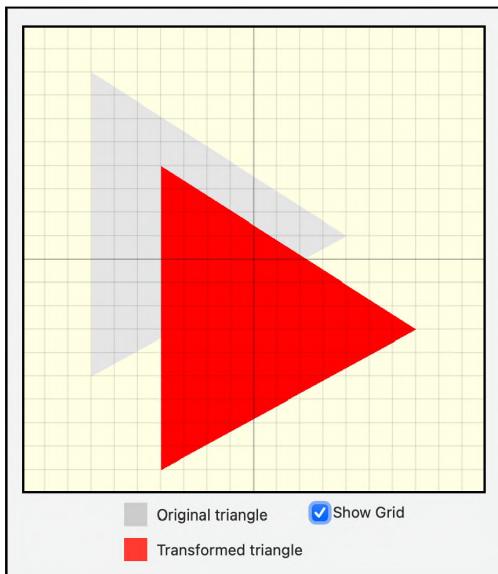
Using the vector image editor, *Affinity Designer*, you can scale and rotate a cat through a series of affine transformations. Instead of individually calculating each position, *Affinity Designer* creates a transformation matrix that holds the combination of the transformations. It then applies the transformation to each element.

**Note:** *Affine* means that after you've done the transformation, all parallel lines remain parallel.

Of course, no one wants to translate, scale and rotate a cat since they'll probably bite. So instead, you'll translate, scale and rotate a triangle.

## The Starter Project & Setup

- Open and run the starter project located in the starter folder for this chapter.



*The starter project*

This project renders a Triangle twice rather than a Quad.

In Renderer, you'll see two draw calls (one for each triangle). Renderer passes position to the vertex function and color to the fragment function; it does this for each triangle. The gray triangle is at its original position, and the red triangle has transformations.

- Before moving on to the next step, make sure you understand the code in Renderer's `draw(in:)` and the vertex function in **Shaders.metal**.

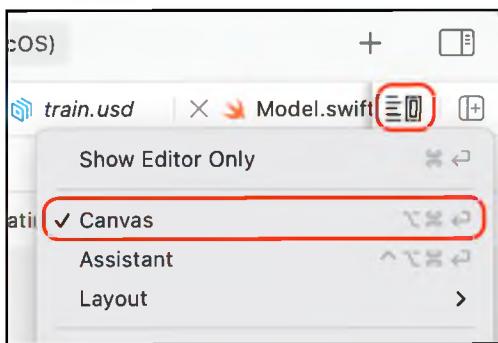
**ContentView.swift** is now located in the group **SwiftUI Views**, and it displays a grid over the metal view so that you can visualize your vertex positions more easily.

## Setting Up the Preview Using SwiftUI

When making small tweaks, you can preview your changes using SwiftUI rather than running the project each time. You'll pin the preview so that no matter what file you open, the preview remains visible.

**Note:** If you have a small screen, you may prefer to run the project each time to test it.

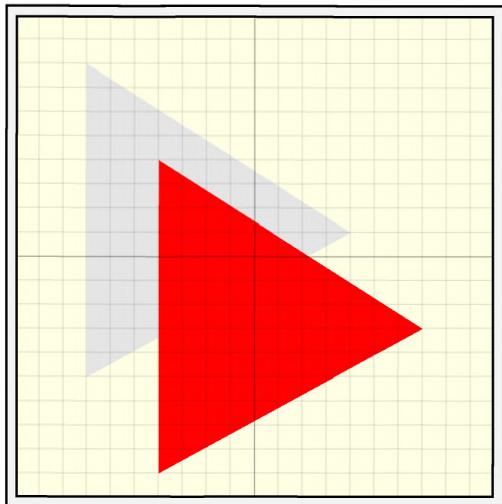
- Open **ContentView.swift**.
- At the top-right of the Xcode window, click **Adjust Editor Options** and choose **Canvas** from the menu. You can change the layout from this menu so that the preview canvas is on the right.



*Showing the preview canvas*

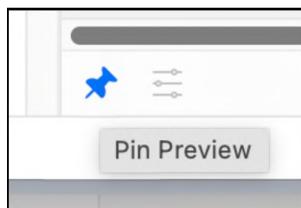
- Build the project using the shortcut **Command-B**, and press **Option-Command-P** to resume the preview. (If the resume preview shortcut doesn't work, try **Editor Menu** ▶ **Canvas** ▶ **Refresh Canvas**. The shortcut should now work. You may also have to sign your team in the project's target.)

**Note:** Whenever you make changes to your code, you'll need to build your app using **Command-B**. You then need to press **Option-Command-P** to resume/refresh the preview as the preview will not automatically build the Metal shaders.



*The preview*

- At the bottom-left of the preview window, you'll see an option to pin the preview.



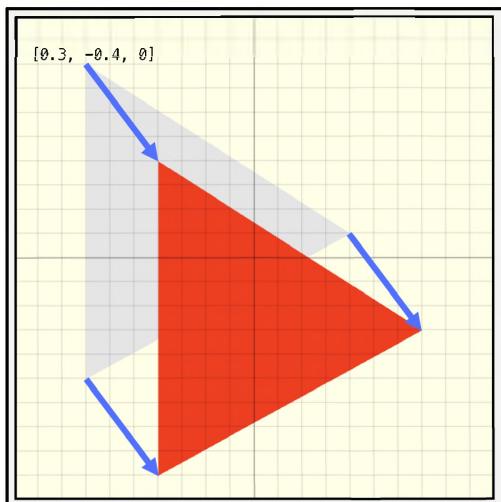
*Pinning the preview*

With the preview pinned, you can now move to other files without the preview disappearing.

# Translation

The starter project renders two triangles:

- A gray triangle without any transformations.
- A red triangle translated with `position = simd_float3(0.3, -0.4, 0)`.



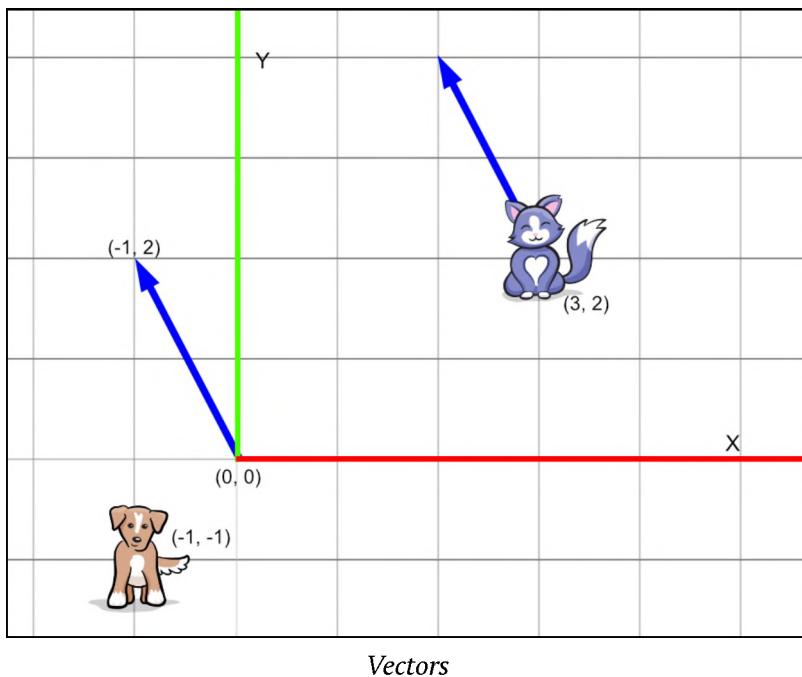
*Displacement Vectors*

In the first challenge in the previous chapter, you calculated the position of each vertex in the shader function. A more common computer graphics paradigm is to set the position of each vertex of the model in the vertex buffer, and then send a matrix to the vertex shader that contains the model's current position, rotation and scale.

# Vectors & Matrices

You can better describe position as a **displacement vector** of  $[0.3, -0.4, 0]$ . You move each vertex 0.3 units in the x-direction, and -0.4 in the y-direction from its starting position.

In the following image, the blue arrows are vectors.



Vectors

The left blue arrow is a vector with a value of  $[-1, 2]$ . The right blue arrow — the one near the cat — is also a vector with a value of  $[-1, 2]$ . Positions (points) are locations in space, whereas vectors are displacements in space. In other words, a vector contains the amount and direction to move. If you were to displace the cat by the blue vector, it would end up at point  $(2, 4)$ . That's the cat's position  $(3, 2)$  plus the vector  $[-1, 2]$ .

This 2D vector is a **1x2** matrix. It has one column and two rows.

**Note:** You can order Matrices by rows or columns. Metal matrices are constructed in **column-major** order, which means that columns are contiguous in memory.

A matrix is a two-dimensional array. Even the single number **1** is a  $1 \times 1$  matrix. In fact, the number 1 is unique in that when you multiply a number by 1, the answer is always that number. All square matrices — where the array width is the same as the array height — have a matrix with this same property. It's called the **identity matrix**. Any vector or matrix multiplied by an identity matrix returns the same value.

A  $4 \times 4$  identity matrix looks like this (all zeros, except for the diagonal 1s):

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

An identity matrix

A 3D **transformation matrix** has four rows and four columns. A transformation matrix holds scaling and rotation information in the upper left  $3 \times 3$  matrix, with the translation information in the last column. When you multiply vectors and matrices, the number of columns of the left side matrix or vector must equal the number of rows of the right side. For example, you can't multiply a `float3` by a `float4x4`.

## The Magic of Matrices

When you multiply matrices, you combine them into one matrix. You can then multiply a vector by this matrix to transform the vector. For example, you can set up a rotation matrix and a translation matrix. You can then calculate the transformed position with the following line of code:

```
translationMatrix * rotationMatrix * positionVector
```

Matrix multiplication goes from right to left. Here, the rotation is applied before the translation.

This is a fundamental of linear algebra — and if you want to continue with computer graphics, you'll need to understand linear algebra more fully. For now, understanding the concepts of setting up a transformation matrix can take you a long way.

## Creating a Matrix

- Open **Renderer.swift**, and locate where you render the first gray triangle in `draw(in:)`.
- Change the position code from:

```
var position = SIMD3<Float>(0, 0, 0)
renderEncoder.setVertexBytes(
    &position,
    length: MemoryLayout<SIMD3<Float>>.stride,
    index: 11)
```

- To:

```
var translation = matrix_float4x4()
translation.columns.0 = [1, 0, 0, 0]
translation.columns.1 = [0, 1, 0, 0]
translation.columns.2 = [0, 0, 1, 0]
translation.columns.3 = [0, 0, 0, 1]
var matrix = translation
renderEncoder.setVertexBytes(
    &matrix,
    length: MemoryLayout<matrix_float4x4>.stride,
    index: 11)
```

Here, you create an identity matrix and a render command to send to the GPU.

- Locate the position code for the second red triangle and change:

```
position = SIMD3<Float>(0.3, -0.4, 0)
renderEncoder.setVertexBytes(
    &position,
    length: MemoryLayout<SIMD3<Float>>.stride,
    index: 11)
```

- To:

```
let position = SIMD3<Float>(0.3, -0.4, 0)
translation.columns.3.x = position.x
translation.columns.3.y = position.y
translation.columns.3.z = position.z
matrix = translation
renderEncoder.setVertexBytes(
    &matrix,
    length: MemoryLayout<matrix_float4x4>.stride,
    index: 11)
```

You'll use this matrix to translate the position in the vertex shader.

- Open `Shaders.metal`, and change:

```
constant float3 &position [[buffer(11)]])
```

- To:

```
constant float4x4 &matrix [[buffer(11)]])
```

You receive the matrix into the shader.

- In the vertex function, change:

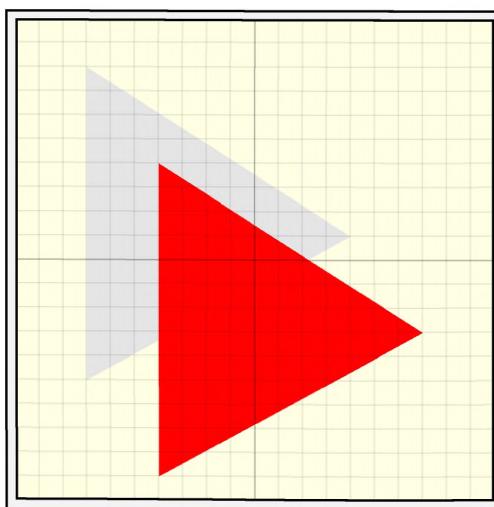
```
float3 translation = in.position.xyz + position;
```

- To:

```
float3 translation = in.position.xyz + matrix.columns[3].xyz;
```

You use the fourth column of the matrix as the displacement vector.

- Build and resume the preview. So far, the output is the same.



*Translation by adding a matrix column to the position*

Remember that this matrix is also going to hold rotation and scaling information, so to calculate the position, instead of adding the translation displacement vector, you'll do matrix multiplication.

- Change the contents of the vertex function to:

```
float4 translation = matrix * in.position;
VertexOut out {
    .position = translation
};
return out;
```

- Build and resume the preview, and you'll see there's still no change.

You can now add scaling and rotation to the matrix in `Renderer` without having to change the shader function each time.

## Scaling

- Open `Renderer.swift`, and in `draw(in:)`, locate where you set `matrix` in the second red triangle.

- Before `matrix = translation`, add this:

```
let scaleX: Float = 1.2
let scaleY: Float = 0.5
let scaleMatrix = float4x4(
    [scaleX, 0, 0, 0],
    [0, scaleY, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1])
```

Without going into mathematics too much, you can use this code to set up a scale matrix.

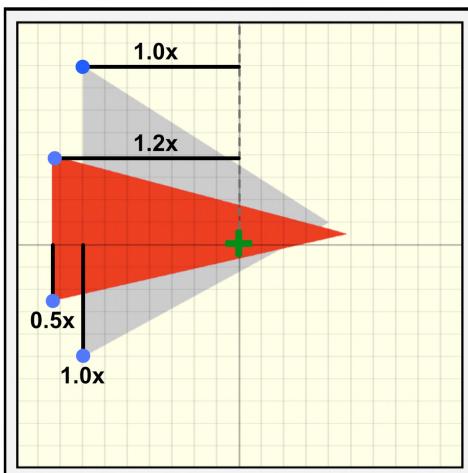
- Change `matrix = translation` to:

```
matrix = scaleMatrix
```

You multiply the translation matrix by the scale matrix instead of the translation matrix.



- Build and preview the app.



*Scaling with a matrix*

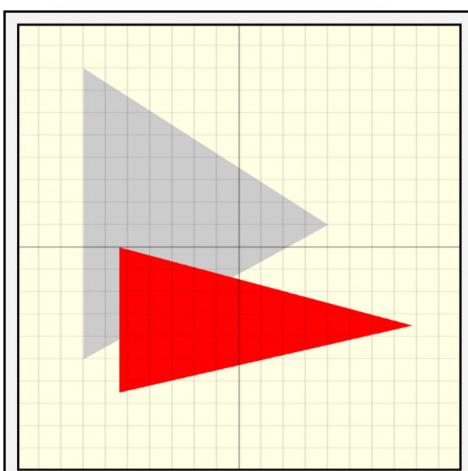
In the vertex function, the matrix multiplies each vertex of the triangle by the x and y scales.

- Change `matrix = scaleMatrix` to:

```
matrix = translation * scaleMatrix
```

This code translates the scaled triangle.

- Build and preview the app.



*A translated and scaled triangle*

# Rotation

You perform rotation in a similar way to scaling.

- Change `matrix = translation * scaleMatrix`, to this:

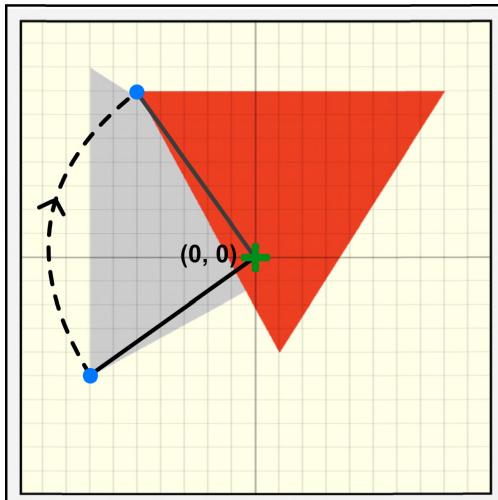
```
let angle = Float.pi / 2.0
let rotationMatrix = float4x4(
    [cos(angle), -sin(angle), 0, 0],
    [sin(angle), cos(angle), 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1])

matrix = rotationMatrix
```

Here, you set a rotation around the z-axis of the angle in radians.

**Note:** `Float.pi / 2.0` is the same as  $90^\circ$ , which is 1.5708 radians. A radian is the standard unit in computer graphics. This is the formula to convert degrees to radians:  $\text{degrees} * \pi / 180 = \text{radians}$ .

- Build and preview, and you'll see how each of the vertices of the red triangle are rotated by  $90^\circ$  around the origin  $[0, 0, 0]$ .



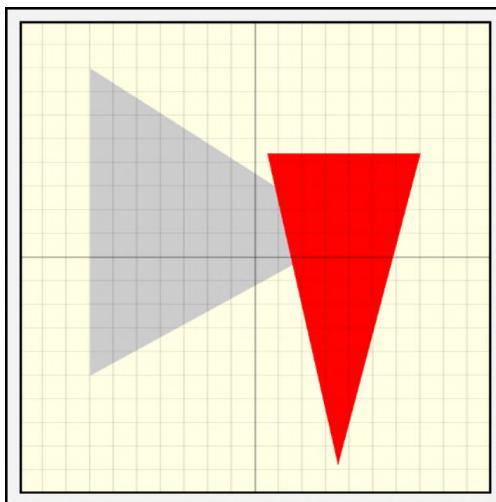
*Rotating about the origin*

- Replace `matrix = rotationMatrix` with:

```
matrix = translation * rotationMatrix * scaleMatrix
```

This code first scales each vertex, then rotates, then translates.

- Build and preview.



*Scale, rotate and translate*

**The order of matrix operations is important.** Experiment with changing the order to see what happens.

Scaling and rotation take place at the origin point (coordinates  $[0, 0, 0]$ ). There may be times, however, that you want the rotation to take place around a different point. For example, let's rotate the triangle around the right-most point of the triangle when it's in its identity position (i.e., the same position and rotation as the gray triangle).

To rotate the triangle, you'll set up a translation matrix with the vector between the origin and the right-most point, taking the following steps:

1. Translate all the vertices using the translation matrix.
2. Rotate.
3. Translate back again.

- Before setting `matrix` in the red triangle, add this code:

```
translation.columns.3.x = triangle.vertices[6]
translation.columns.3.y = triangle.vertices[7]
translation.columns.3.z = triangle.vertices[8]
```

You set the translation matrix to move to the third vertex of the triangle, which is the right-most point.

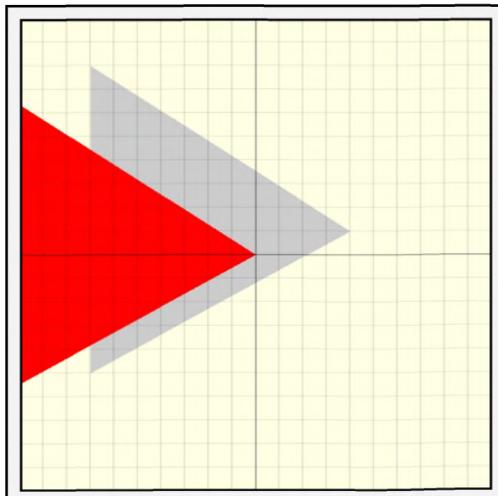
Remember the steps. Step 1 is to translate all of the vertices by the distance from the origin. You can achieve this by setting a matrix to the vertex's vector value and using the translate matrix's **inverse**.

Don't forget to build and preview after each of the following steps so that you can see what the matrix multiplication is doing.

- Change `matrix = translation * rotationMatrix * scaleMatrix` to:

```
matrix = translation.inverse
```

This code places the right-most vertex at the origin, translating all other vertices by the same amount.

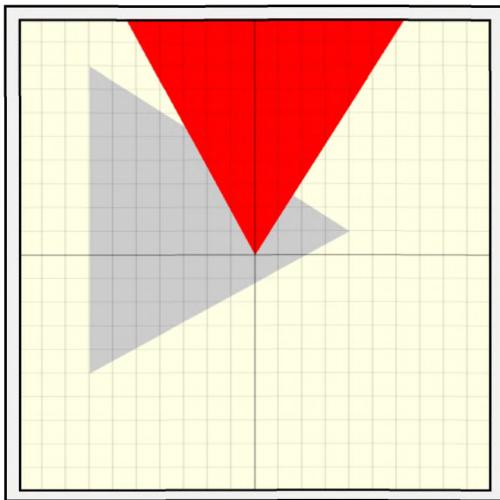


*Rotate about a point (1)*

- Change the code you just entered to:

```
matrix = rotationMatrix * translation.inverse
```

The triangle rotates by 90° around the origin.

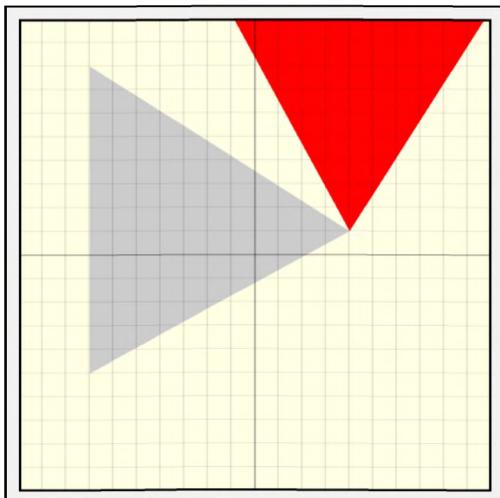


*Rotate about a point (2)*

► Change the code you just entered to:

```
matrix = translation * rotationMatrix * translation.inverse
```

Fantastic! You're doing all of the steps of translating each vertex by the distance of the right-most vertex from the origin. After that, you're rotating each vertex and translating it back again, causing the triangle to rotate around its right-most point.



*Rotate about a point (3)*

## Key Points

- A vector is a matrix with only one row or column.
- By combining three matrices for translation, rotation and scale, you can position a model anywhere in the scene.
- In the resources folder for this chapter, **references.markdown** suggests further reading to help better understand transformations with linear algebra.

# 6 Chapter 6: Coordinate Spaces

To easily find a point on a grid, you need a coordinate system. For example, if the grid happens to be your iPhone 13 screen, the center point might be `x: 195, y: 422`. However, that point may be different depending on what space it's in.

In the previous chapter, you learned about matrices. By multiplying a vertex's position by a particular matrix, you can convert the vertex position to a different **coordinate space**. There are typically six spaces a vertex travels as its making its way through the pipeline:

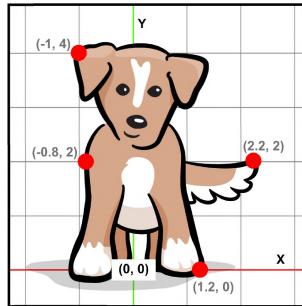
- Object
- World
- Camera
- Clip
- NDC (Normalized Device Coordinate)
- Screen

Since this is starting to read like a description of Voyager leaving our solar system, let's have a quick conceptual look at each coordinate space before attempting the conversions.



## Object Space

If you're familiar with the Cartesian coordinate system, you know that it uses two points to map an object's location. The following image shows a 2D grid with the possible vertices of the dog mapped using Cartesian coordinates.

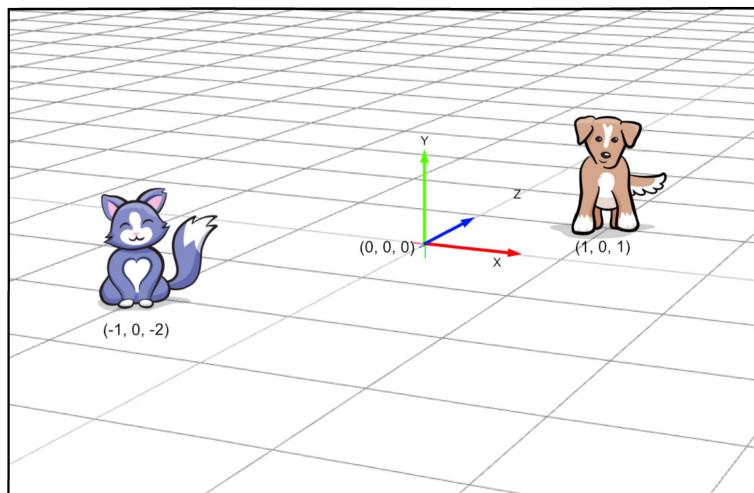


*Vertices in object space*

The positions of the vertices are in relation to the dog's **origin**, which is located at  $(0, 0)$ . The vertices in this image are located in **object space** (or **local** or **model** space). In the previous chapter, `Triangle` held an array of vertices in object space, describing the vertex of each point of the triangle.

## World Space

In the following image, the direction arrows mark the world's origin at  $(0, 0, 0)$ . So, in world space, the dog is at  $(1, 0, 1)$  and the cat is at  $(-1, 0, -2)$ .



*Vertices in world space*

Of course, we all know that cats *always* see themselves at the center of the universe, so, naturally, the cat is located at  $(0, 0, 0)$  in *cat space*. This *cat space* location makes the dog's position,  $(2, 0, 3)$ , relative to the cat. When the cat moves around in his *cat space*, he remains at  $(0, 0, 0)$ , while the position of everything else changes relative to the cat.

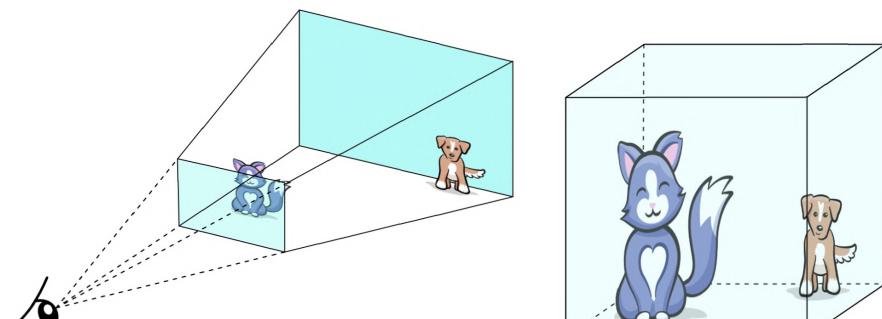
**Note:** Cat space is not recognized as a traditional 3D coordinate space, but mathematically, you can create your own space and use any position in the universe as the origin. Every other point in the universe is now *relative* to that origin. In a later chapter, you'll discover other spaces besides the ones described here.

## Camera Space

Enough about the cat. Let's move on to the dog. For him, the center of the universe is the person holding the camera. So, in **camera space** (or **view space**), the camera is at  $(0, 0, 0)$  and the dog is approximately at  $(-3, -2, 7)$ . When the camera moves, it stays at  $(0, 0, 0)$ , but the positions of the dog and cat move relative to the camera.

## Clip Space

The main reason for doing all this math is to project with perspective. In other words, you want to take a three-dimensional scene into a two-dimensional space. Clip space is a distorted cube that's ready for flattening.



Clip space

In this scene, the dog and the cat are the same size, but the dog appears smaller because of its location in 3D space. Here, the dog is farther away than the cat, so he looks smaller.

**Note:** You could use orthographic or isometric projection instead of perspective projection, which comes in handy if you're rendering engineering drawings.

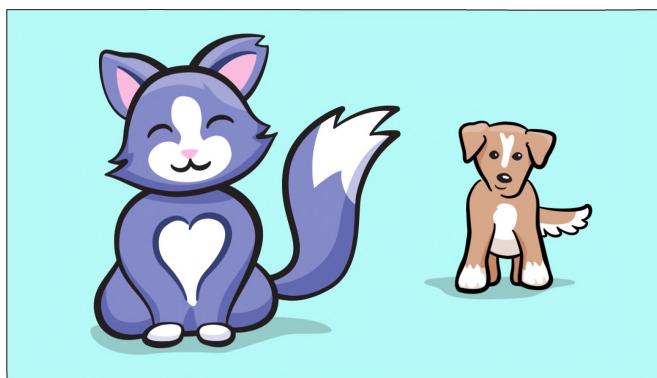
## NDC (Normalized Device Coordinate) Space

Projection into clip space creates a half cube of  $w$  size. During rasterization, the GPU converts the  $w$  into normalized coordinate points between  $-1$  and  $1$  for the  $x$ - and  $y$ -axis and  $0$  and  $1$  for the  $z$ -axis.

## Screen Space

Now that the GPU has a normalized cube, it will flatten clip space into two dimensions and convert everything into screen coordinates, ready to display on the device's screen.

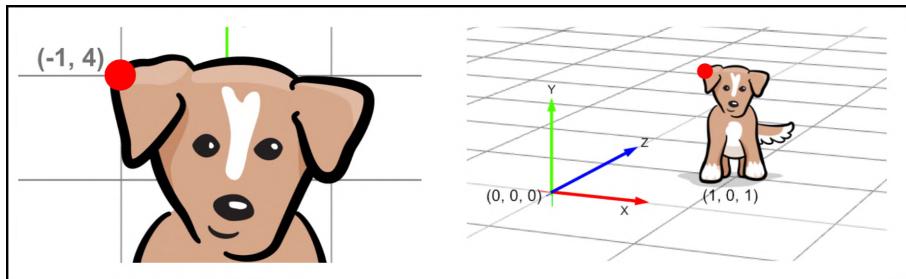
In this final image, the flattened scene has perspective — with the dog being smaller than the cat, indicating the distance between the two.



*Final render*

## Converting Between Spaces

To convert from one space to another, you can use transformation matrices. In the following image, the vertex on the dog's ear is  $(-1, 4, 0)$  in object space. But in world space, the origin is different, so the vertex – judging from the image – is at about  $(0.75, 1.5, 1)$ .



*Converting object to world*

To change the dog vertex positions from object space to world space, you can translate (move) them using a transformation matrix. Since you control four spaces, you have access to three corresponding matrices:



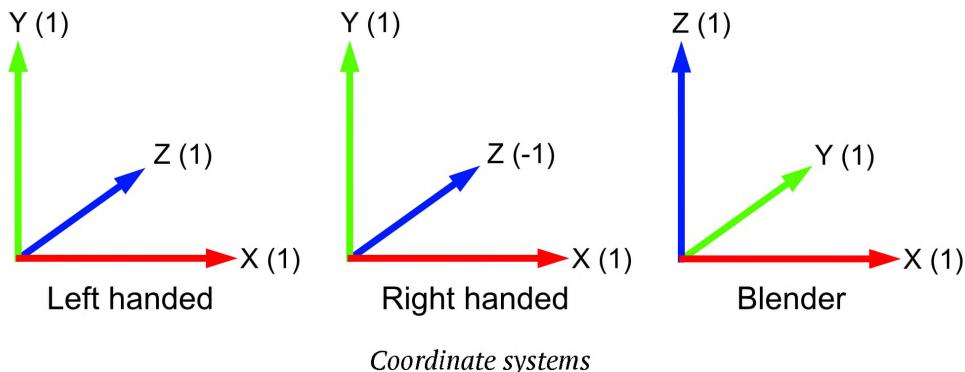
*The three transformation matrices*

- Model matrix: between object and world space
- View matrix: between world and camera space
- Projection matrix: between camera and clip space

## Coordinate Systems

Different graphics APIs use different coordinate systems. You already found out that Metal's NDC (Normalized Device Coordinates) uses 0 to 1 on the z-axis. You also may already be familiar with OpenGL, which uses 1 to -1 on the z-axis.

In addition to being different sizes, OpenGL's z-axis points in the opposite direction from Metal's z-axis. That's because OpenGL's system is a **right-handed** coordinate system, and Metal's system is a **left-handed** coordinate system. Both systems use x to the right and y as up. Blender uses a different coordinate system, where z is up, and y is into the screen.



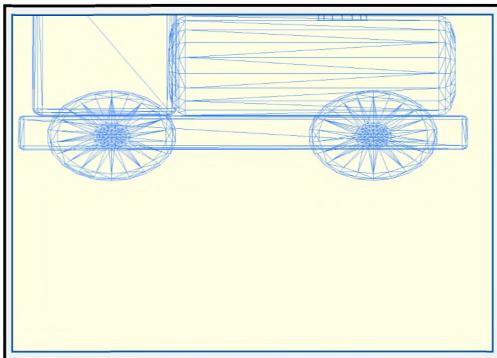
If you're consistent with your coordinate system and create matrices accordingly, it doesn't matter what coordinate system you use. In this book, we're using Metal's left-handed coordinate system, but we could have used a right-handed coordinate system with different matrix creation methods instead.

## The Starter Project

With a better understanding of coordinate systems and spaces, you're ready to start creating matrices.

- In Xcode, open the starter project for this chapter and either set up the SwiftUI preview (as described in the previous chapter) or build and run the app. Note that the width constraint is removed, so you may have to zoom out the preview.

The project is similar to the playground you set up in Chapter 2, “3D Models”, where you rendered **train.usd**.



*Starter project*

**MathLibrary.swift** — located in the **Utility** group — contains methods that are extensions on `float4x4` for creating the translation, scale and rotation matrices. This file also contains `typealiases` for `float2/3/4`, so you don’t have to type `simd_float2/3/4`.

**Model.swift** contains the model initialization and loading code. This file also contains `render(encoder:)` — called from `Renderer`’s `draw(in:)` — which renders the model.

**VertexDescriptor.swift** creates a default `MDLVertexDescriptor`. The default `MTKVertexDescriptor` is derived from this descriptor. When using Model I/O to load models with vertex descriptors, the code can get a bit lengthy. Rather than creating a MetalKit `MTKVertexDescriptor`, it’s easier to create a Model I/O `MDLVertexDescriptor` and then convert to the `MTKVertexDescriptor` that the pipeline state object needs using `MTKMetalVertexDescriptorFromModelIO(_:)`. If you examine the vertex descriptor code from the previous chapter, the same process is used for both vertex descriptors. You describe attributes and layouts.

At the moment, your train:

- Takes up the entire width of the screen.
- Has no depth perspective.
- Stretches to fit the size of the application window.

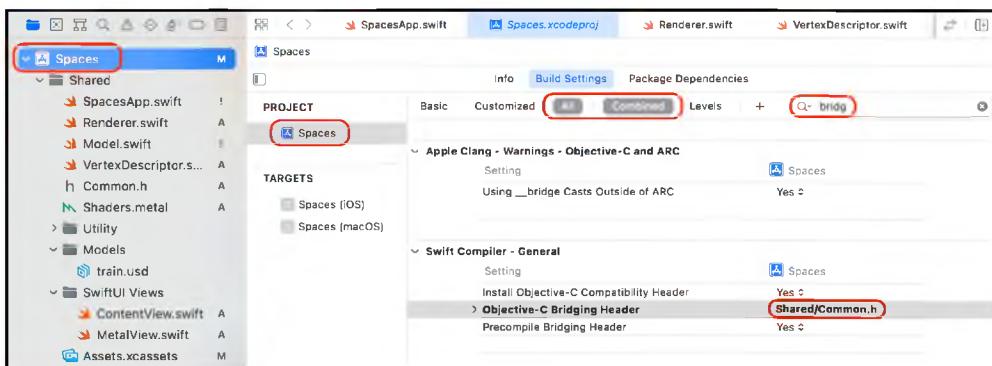
You can decouple the train's vertex positions from the window size by taking the train into other coordinate spaces. The vertex function is responsible for converting the model vertices through these various coordinate spaces, and that's where you'll perform the matrix multiplications that do the conversions between different spaces.

## Uniforms

Constant values that are the same across all vertices or fragments are generally referred to as **uniforms**. The first step is to create a uniform structure to hold the conversion matrices. After that, you'll apply the uniforms to every vertex.

Both the shaders and the code on the Swift side will access these uniform values. If you were to create a `struct` in `Renderer` and a matching `struct` in `Shaders.metal`, there's a better chance you'll forget to keep them synchronized. Therefore, the best approach is to create a bridging header that both C++ and Swift can access. You'll do that now:

- Using the macOS **Header File** template, create a new file in the **Shared** group and name it **Common.h**.
- In the Project navigator, click the main **Spaces** project folder.
- Select the project **Spaces**, and then select **Build Settings** along the top. Make sure **All** and **Combined** are highlighted.
- In the search bar, type **bridg** to filter the settings. Double-click the **Objective-C Bridging Header** value and enter **Shared/Common.h**.



*Setting up the bridging header*

This configuration tells Xcode to use this file for both the C++ derived Metal Shading Language and Swift.

- In **Common.h** before the final `#endif`, add the following code:

```
#import <simd/simd.h>
```

This code imports the **simd** framework, which provides types and functions for working with vectors and matrices.

- Next, add the uniforms structure:

```
typedef struct {
    matrix_float4x4 modelMatrix;
    matrix_float4x4 viewMatrix;
    matrix_float4x4 projectionMatrix;
} Uniforms;
```

These three matrices — each with four rows and four columns — will hold the necessary conversion between the spaces.

## The Model Matrix

Your train vertices are currently in object space. To convert these vertices to world space, you'll use `modelMatrix`. By changing `modelMatrix`, you'll be able to translate, scale and rotate your train.

- In **Renderer.swift**, add the new structure to **Renderer**:

```
var uniforms = Uniforms()
```

You defined `Uniforms` in **Common.h** (the bridging header file), so Swift is able to recognize the `Uniforms` type.

- At the bottom of `init(metalView:)`, add:

```
let translation = float4x4(translation: [0.5, -0.4, 0])
let rotation =
    float4x4(rotation: [0, 0, Float(45).degreesToRadians])
uniforms.modelMatrix = translation * rotation
```

Here, you set `modelMatrix` to have a translation of 0.5 units to the right, 0.4 units down and a counterclockwise rotation of 45 degrees.

- In `draw(in:)` before `model.render(encoder: renderEncoder)`, add this:

```
renderEncoder.setVertexBytes(
    &uniforms,
    length: MemoryLayout<Uniforms>.stride,
```



```
index: 11)
```

This code sets up the uniform matrix values on the Swift side.

- Open **Shaders.metal**, and import the bridging header file after setting the namespace:

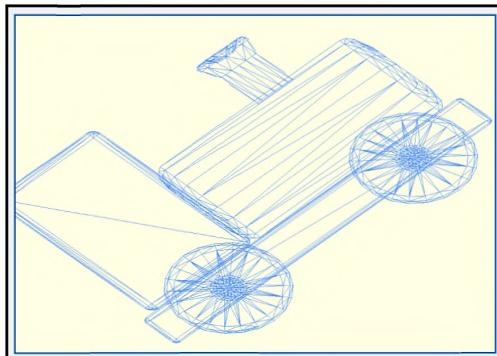
```
#import "Common.h"
```

- Change the vertex function to:

```
vertex VertexOut vertex_main(
    VertexIn in [[stage_in]],
    constant Uniforms &uniforms [[buffer(11)]])
{
    float4 position = uniforms.modelMatrix * in.position;
    VertexOut out {
        .position = position
    };
    return out;
}
```

Here, you receive the **Uniforms** structure as a parameter, and then you multiply all of the vertices by the model matrix.

- Build and preview the app.



*Train in world space*

In the vertex function, you multiply the vertex position by the model matrix. All of the vertices are rotated then translated. The train vertex positions still relate to the width of the screen, so the train looks stretched. You'll fix that momentarily.

## View Matrix

To convert between world space and camera space, you set a view matrix. Depending on how you want to move the camera in your world, you can construct the view matrix appropriately. The view matrix you'll create here is a simple one, best for FPS (First Person Shooter) style games.

- In `Renderer.swift` at the end of `init(metalView:)`, add this code:

```
uniforms.viewMatrix = float4x4(translation: [0.8, 0, 0]).inverse
```

Remember that all of the objects in the scene should move in the opposite direction to the camera. `inverse` does an opposite transformation. So, as the camera moves to the right, everything in the world appears to move 0.8 units to the left. With this code, you set the camera in world space, and then you add `.inverse` so that the objects will react in inverse relation to the camera.

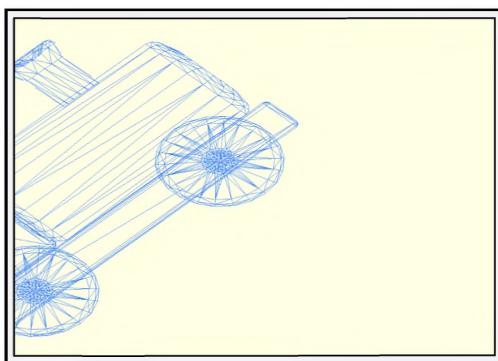
- In `Shaders.metal`, change:

```
float4 position = uniforms.modelMatrix * in.position;
```

- To:

```
float4 position = uniforms.viewMatrix * uniforms.modelMatrix  
* in.position;
```

- Build and preview the app.



*Train in camera space*

The train moves `0.8` units to the left. Later, you'll be able to navigate through a scene using the keyboard, and just changing the view matrix will update all of the objects in the scene around the camera.

The last matrix you'll set will prepare the vertices to move from camera space to clip space. This matrix will also allow you to use unit values instead of the `-1` to `1` NDC (Normalized Device Coordinates) that you've been using. To demonstrate why this is necessary, you'll add some animation to the train and rotate it on the y-axis.

► Open **Renderer.swift**, and in `draw(in:)`, just above the following code:

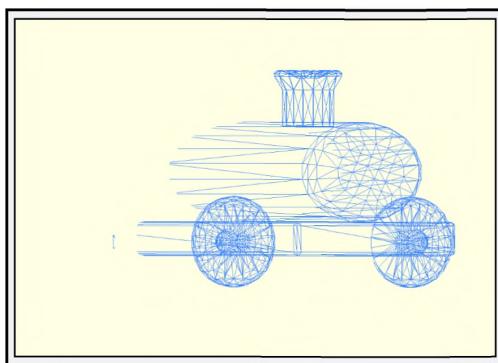
```
renderEncoder.setVertexBytes(  
    &uniforms,  
    length: MemoryLayout<Uniforms>.stride,  
    index: 1)
```

► Add this code:

```
timer += 0.005  
uniforms.viewMatrix = float4x4.identity  
let translationMatrix = float4x4(translation: [0, -0.6, 0])  
let rotationMatrix = float4x4(rotationY: sin(timer))  
uniforms.modelMatrix = translationMatrix * rotationMatrix
```

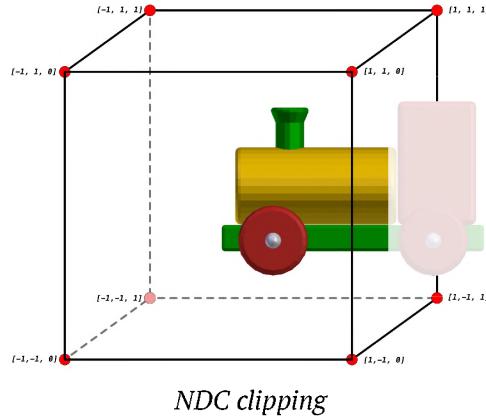
Here, you reset the camera view matrix and replace the model matrix with a rotation around the y-axis.

► Build and preview the app.



A clipped train

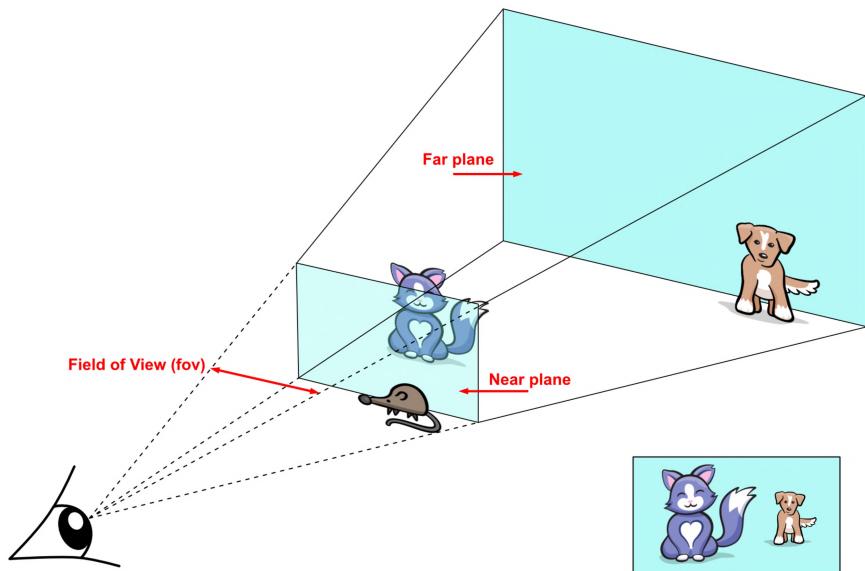
You can see that when the train rotates, any vertices greater than **1.0** on the z-axis are clipped. Any vertex outside Metal's NDC will be clipped.



## Projection

It's time to apply some perspective to your render to give your scene some depth.

The following diagram shows a 3D scene. At the bottom-right, you can see how the rendered scene will appear.



When you render a scene, you need to consider:

- How much of that scene will fit on the screen. Your eyes have a **field of view** of about 200°, and within that field of view, your computer screen takes up about 70°.
- How far you can see by having a **far plane**. Computers can't see to infinity.
- How close you can see by having a **near plane**.
- The **aspect ratio** of the screen. Currently, your train changes size when the screen size changes. When you take into account the width and height ratio, this won't happen.

The image above shows all these things. The shape created from the near to the far plane is a cut-off pyramid called a **frustum**. Anything in your scene that's located outside the frustum will not render.

Compare the rendered image again to the scene setup. The rat in the scene won't render because he's in front of the near plane.

`MathLibrary.swift` provides a projection method that returns the matrix to project objects within this frustum into clip space, ready for conversion to NDC coordinates.

## Projection Matrix

► Open `Renderer.swift`, and add this code to `mtkView(_:drawableSizeWillChange:)`:

```
let aspect =  
    Float(view.bounds.width) / Float(view.bounds.height)  
let projectionMatrix =  
    float4x4(  
        projectionFov: Float(45).degreesToRadians,  
        near: 0.1,  
        far: 100,  
        aspect: aspect)  
uniforms.projectionMatrix = projectionMatrix
```

This delegate method gets called whenever the view size changes. Because the aspect ratio will change, you must reset the projection matrix.

You're using a field of view of 45°; a near plane of 0.1, and a far plane of 100 units.



- At the end of `init(metalView:)`, add this:

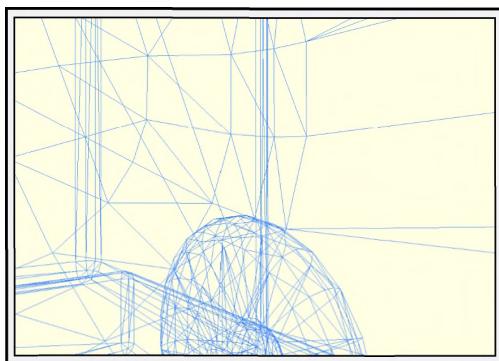
```
mtkView(  
    metalView,  
    drawableSizeWillChange: metalView.bounds.size)
```

This code ensures that you set up the projection matrix at the start of the app.

- In the vertex function of `Shaders.metal`, change the position matrix calculation to:

```
float4 position =  
    uniforms.projectionMatrix * uniforms.viewMatrix  
    * uniforms.modelMatrix * in.position;
```

- Build and preview the app.



*Zoomed in*

Because of the projection matrix, the z-coordinates measure differently now, so you're zoomed in on the train.

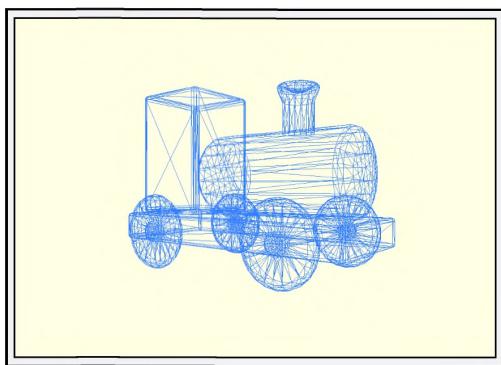
- In `Renderer.swift` in `draw(in:)`, replace:

```
uniforms.viewMatrix = float4x4.identity
```

- With:

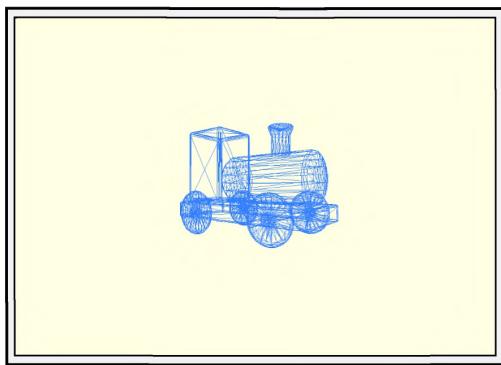
```
uniforms.viewMatrix = float4x4(translation: [0, 0, -3]).inverse
```

This moves the camera back into the scene by three units. Build and preview:



*Camera moved back*

► In `mtkView(_:drawableSizeWillChange:)`, change the projection matrix's `projectionFOV` parameter to **70°**, then build and preview the app.



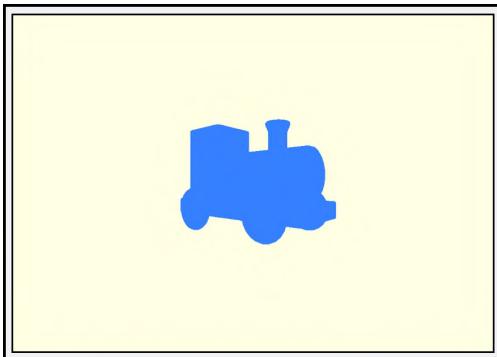
*A greater field of view*

The train appears smaller because the field of view is wider, and more objects horizontally can fit into the rendered scene.

**Note:** Experiment with the projection values and the model transformation. In `draw(in:)`, set `translationMatrix`'s `z` translation value to a distance of `97`, and the front of the train is just visible. At `z = 98`, the train is no longer visible. The projection far value is `100` units, and the camera is back `3` units. If you change the projection's `far` parameter to `1000`, the train is visible again.

- To render a solid train, in `draw(in:)`, remove:

```
renderEncoder.setTriangleFillMode(.lines)
```



*The train positioned in a scene*

## Perspective Divide

Now that you've converted your vertices from object space through world space, camera space and clip space, the GPU takes over to convert to NDC coordinates (that's  $-1$  to  $1$  in the  $x$  and  $y$  directions and  $0$  to  $1$  in the  $z$  direction). The ultimate aim is to scale all the vertices from clip space into NDC space, and by using the fourth  $w$  component, that task gets a lot easier.

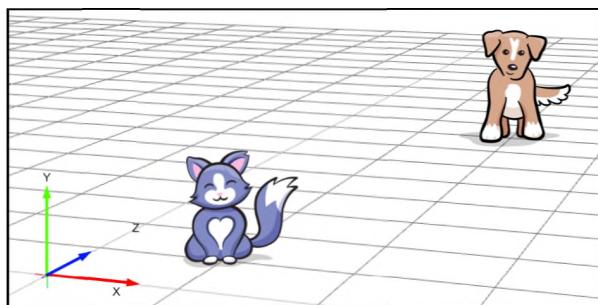
To scale a point, such as  $(1, 2, 3)$ , you can have a fourth component:  $(1, 2, 3, 3)$ . Divide by that last  $w$  component to get  $(1/3, 2/3, 3/3, 1)$ . The  $xyz$  values are now scaled down. These coordinates are known as **homogeneous**, which means *of the same kind*.

The projection matrix projected the vertices from a frustum to a cube in the range  $-w$  to  $w$ . After the vertex leaves the vertex function along the pipeline, the GPU performs a **perspective divide** and divides the  $x$ ,  $y$  and  $z$  values by their  $w$  value. The higher the  $w$  value, the further back the coordinate is. The result of this calculation is that all visible vertices will now be within NDC.

**Note:** To avoid a divide by zero, the projection near plane should always be a value slightly more than zero.

The  $w$  value is the main difference between a `float4` vector direction and a `float4` position. Because of the perspective divide, the position must have a value, generally 1, in  $w$ . Whereas a vector should have 0 in the  $w$  value as it doesn't go through the perspective divide.

In the following picture, the dog and cat are the same height – perhaps a  $y$  value of 2, for example. With projection, since the dog is farther back, it should appear smaller in the final render.



*The dog should appear smaller.*

After projection, the cat might have a  $w$  value of  $\sim 1$ , and the dog might have a  $w$  value of  $\sim 8$ . Dividing by  $w$  would give the cat a height of 2 and the dog a height of  $1/4$ , which will make the dog appear smaller.

## NDC to Screen

Finally, the GPU converts from normalized coordinates to whatever the device screen size is. You may already have done something like this at some time in your career when converting between normalized coordinates and screen coordinates.

To convert Metal NDC (Normalized Device Coordinates), which are between -1 and 1 to a device, you *can* use something like this:

```
converted.x = point.x * screenWidth/2 + screenWidth/2
converted.y = point.y * screenHeight/2 + screenHeight/2
```

However, you can also do this with a matrix by scaling half the screen size and translating by half the screen size. The clear advantage of this method is that you can set up a transformation matrix once and multiply any normalized point by the matrix to convert it into the correct screen space using code like this:

```
converted = matrix * point
```

The rasterizer on the GPU takes care of the matrix calculation for you.

## Refactoring the Model Matrix

Currently, you set all the matrices in `Renderer`. Later, you'll create a `Camera` structure to calculate the view and projection matrices.

For the model matrix, rather than updating it directly, any object that you can move — such as a model or a camera — can hold a position, rotation and scale. From this information, you can construct the model matrix.

► Create a new Swift file named `Transform.swift`

► Add the new structure:

```
struct Transform {
    var position: float3 = [0, 0, 0]
    var rotation: float3 = [0, 0, 0]
    var scale: Float = 1
}
```

This struct will hold the transformation information for any object that you can move.

► Add an extension with a computed property:

```
extension Transform {
    var modelMatrix: matrix_float4x4 {
        let translation = float4x4(translation: position)
        let rotation = float4x4(rotation: rotation)
        let scale = float4x4(scaling: scale)
        let modelMatrix = translation * rotation * scale
        return modelMatrix
    }
}
```

This code automatically creates a model matrix from any transformable object.

► Add a new protocol so that you can mark objects as transformable:

```
protocol Transformable {
    var transform: Transform { get set }
}
```

► Because it's a bit longwinded to type `model.transform.position`, add a new extension to `Transformable`:

```
extension Transformable {
    var position: float3 {
```

```
    get { transform.position }
    set { transform.position = newValue }
}
var rotation: float3 {
    get { transform.rotation }
    set { transform.rotation = newValue }
}
var scale: Float {
    get { transform.scale }
    set { transform.scale = newValue }
}
}
```

This code provides computed properties to allow you to use `model.position` directly, and the model's `transform` will update from this value.

- Open `Model.swift`, and mark `Model` as `Transformable`.

```
class Model: Transformable {
```

- Add the new `transform` property to `Model`:

```
var transform = Transform()
```

- Open `Renderer.swift`, and from `init(metalView:)`, remove:

```
let translation = float4x4(translation: [0.5, -0.4, 0])
let rotation =
    float4x4(rotation: [0, 0, Float(45).degreesToRadians])
uniforms.modelMatrix = translation * rotation
```

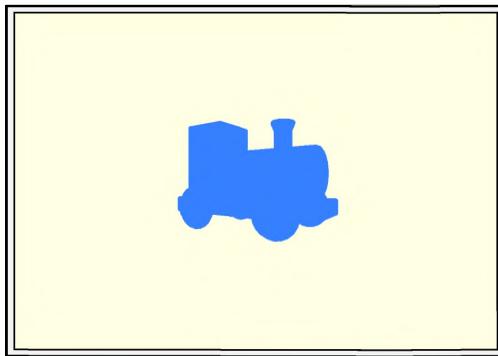
- In `draw(in:)`, replace:

```
let translationMatrix = float4x4(translation: [0, -0.6, 0])
let rotationMatrix = float4x4(rotationY: sin(timer))
uniforms.modelMatrix = translationMatrix * rotationMatrix
```

- With:

```
model.position.y = -0.6
model.rotation.y = sin(timer)
uniforms.modelMatrix = model.transform.modelMatrix
```

- Build and preview the app.



*Using a transform in Model*

The result is exactly the same, but the code is much easier to read — and changing a model's position, rotation and scale is more accessible. Later, you'll extract this code into a `GameScene` so that `Renderer` is left only to render models rather than manipulate them.

## Key Points

- Coordinate spaces map different coordinate systems. To convert from one space to another, you can use matrix multiplication.
- Model vertices start off in object space. These are generally held in the file that comes from your 3D app, such as Blender, but you can procedurally generate them too.
- The **model matrix** converts object space vertices to world space. These are the positions that the vertices hold in the scene's world. The origin at  $[0, 0, 0]$  is the center of the scene.
- The **view matrix** moves vertices into camera space. Generally, your matrix will be the inverse of the position of the camera in world space.
- The **projection matrix** applies three-dimensional perspective to your vertices.

## Where to Go From Here?

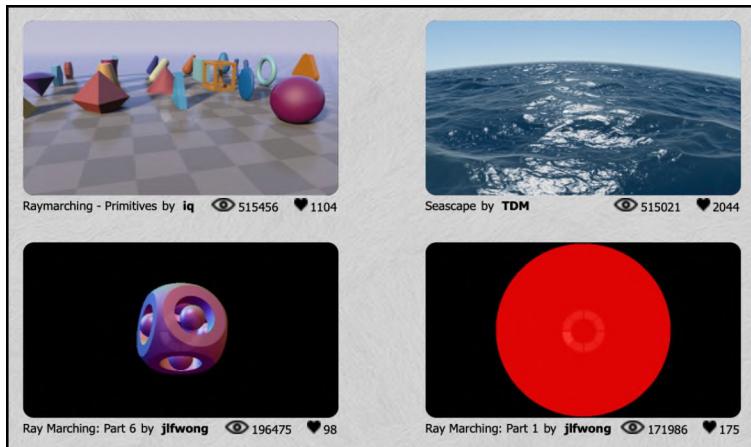
You've covered a lot of mathematical concepts in this chapter without diving too far into the underlying mathematical principles. To get started in computer graphics, you can fill your transform matrices and continue multiplying them at the usual times, but to be sufficiently creative, you'll need to understand some linear algebra. A great place to start is Grant Sanderson's Essence of Linear Algebra at <https://bit.ly/3iYnkN1>. This video treats vectors and matrices visually. You'll also find some additional references in **references.markdown** in the resources folder for this chapter.



# Chapter 7: The Fragment Function

Knowing how to render triangles, lines and points by sending vertex data to the vertex function is a pretty neat skill to have — especially since you’re able to create shapes using only simple, one-line fragment functions. However, fragment shaders are capable of doing a lot more.

- Open the website <https://shadertoy.com>, where you’ll find a dizzying number of brilliant community-created shaders.



*shadertoy.com examples*

These examples may look like renderings of complex 3D models, but looks are deceiving! Every “model” you see here is entirely generated using mathematics, written in a GLSL fragment shader. GLSL is the **Graphics Library Shading Language** for OpenGL — and in this chapter, you’ll begin to understand the principles that all *shading masters* use.

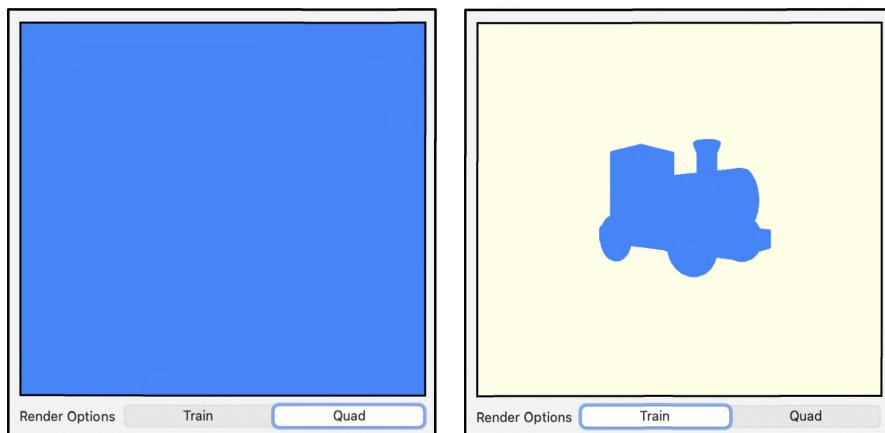


**Note:** Every graphics API uses its own shader language. The principles are the same, so if you find a GLSL shader you like, you can recreate it in Metal's MSL.

## The Starter Project

The starter project shows an example of using multiple pipeline states with different vertex functions, depending on whether you render the rotating train or the full-screen quad.

- Open the starter project for this chapter.
- Build and run the project. (You can choose to render the train or the quad. You'll start with the quad first.)



*The starter project*

Let's have a closer look at code.

- Open **Vertex.metal**, and you'll see two vertex functions:
  - **vertex\_main**: This function renders the train, just as it did in the previous chapter.
  - **vertex\_quad**: This function renders the full-screen quad using an array defined in the shader.

Both functions output a `VertexOut`, containing only the vertex's position.

► Open `Renderer.swift`.

In `init(metalView:options:)`, you'll see two pipeline state objects — one for each of the two vertex functions.

Depending on the value of `options.renderChoice`, `draw(in:)` renders either the train model or the quad. SwiftUI views handle updating `Options`. If you prefer previews to building and running the project each time, change the initial rendering value in `Options.swift`.

- Ensure you understand how this project works before you continue.

## Screen Space

One of the many things a fragment function can do is create complex patterns that fill the screen on a rendered quad. At the moment, the fragment function has only the interpolated position output from the vertex function available to it. So first, you'll learn what you can do with this position and what its limitations are.

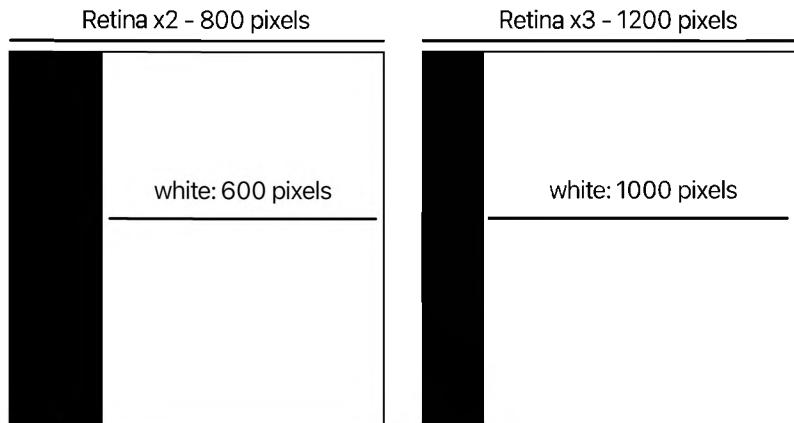
► Open `Fragment.metal`, and change the fragment function contents to:

```
float color;
in.position.x < 200 ? color = 0 : color = 1;
return float4(color, color, color, 1);
```

Here, the rasterizer interpolates `in.position` and converts it into screen space. You defined the width of the Metal view in `ContentView.swift` as 400 points. With the newly added code, you say that if the x position is less than 200, make the color black. Otherwise, make the color white.

**Note:** Although you can use an `if` statement, the compiler optimizes the ternary statement better, so it makes more sense to use that instead.

- Build and preview the change.



*MacBook Pro vs iPhone 12 Pro Max*

Did you expect half the screen to be black? The view is 400 points wide, so it would make sense. But there's something you might not have considered: Apple Retina displays have varying pixel resolutions or pixel densities. For example, a MacBook Pro has a 2x Retina display, whereas the iPhone 12 Pro Max has a 3x Retina display. These varying displays mean that the 400 point Metal view on a MacBook Pro creates an 800x800 pixel drawable texture and the iPhone view creates a 1200x1200 pixel drawable texture.

Your quad fills up the screen, and you're writing to the view's drawable render target texture (the size of which matches the device's display), but there's no easy way to find out in the fragment function what size the current render target texture is.

- Open **Common.h**, and add a new structure:

```
typedef struct {
    uint width;
    uint height;
} Params;
```

This code holds parameters that you can send to the fragment function. You can add parameters to this structure as you need them.

- Open **Renderer.swift**, and add a new property to Renderer:

```
var params = Params()
```

You'll store the current render target size in the new property.

- Add the following code to the end of `mtkView(_:drawableSizeWillChange:)`:

```
params.width = UInt32(size.width)
params.height = UInt32(size.height)
```

`size` contains the drawable texture size of the view. In other words, the view's bounds scaled by the device's scale factor.

- In `draw(in:)`, before calling the methods to render the model or quad, send the parameters to the fragment function:

```
renderEncoder.setFragmentBytes(
    &params,
    length: MemoryLayout<Uniforms>.stride,
    index: 12)
```

Notice that you're using `setFragmentBytes(_:length:index:)` to send data to the fragment function the same way you previously used `setVertexBytes(_:length:index:)`.

- Open `Fragment.metal`, and change the signature of `fragment_main` to:

```
fragment float4 fragment_main(
    constant Params &params [[buffer(12)]],
    VertexOut in [[stage_in]])
```

`Params` with the target drawing texture size is now available to the fragment function.

- Change the code that sets the value of `color` — based on the value of `in.position.x` — to:

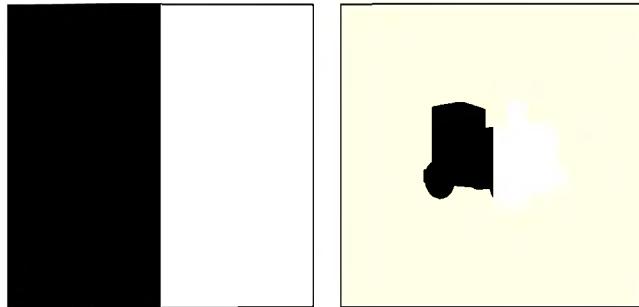
```
in.position.x < params.width * 0.5 ? color = 0 : color = 1;
```

Here, you're using the target render size for the calculation.

- Preview the app in both macOS and the Retina x3 device, iPhone 12 Pro Max.



Fantastic, the render now looks the same for both devices.



*Corrected for retina devices*

## Metal Standard Library Functions

In addition to standard mathematical functions such as `sin`, `abs` and `length`, there are a few other useful functions. Let's have a look.

### step

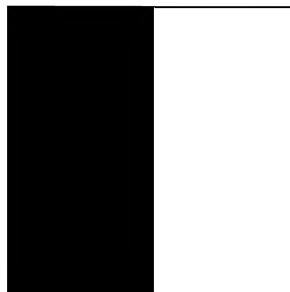
`step(edge, x)` returns `0` if `x` is less than `edge`. Otherwise, it returns `1`. This evaluation is exactly what you're doing with your current fragment function.

- Replace the contents of the fragment function with:

```
float color = step(params.width * 0.5, in.position.x);
return float4(color, color, color, 1);
```

This code produces the same result as before but with slightly less code.

- Build and run.



*step*

The result is that you get black on the left where the result of `step` is `0`, and white on the right where the result of `step` is `1`.

Let's take this further with a checkerboard pattern.

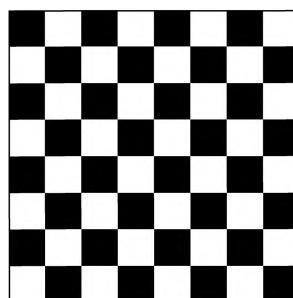
► Replace the contents of the fragment function with:

```
uint checks = 8;  
// 1  
float2 uv = in.position.xy / params.width;  
// 2  
uv = fract(uv * checks * 0.5) - 0.5;  
// 3  
float3 color = step(uv.x * uv.y, 0.0);  
return float4(color, 1.0);
```

Here's what's happening:

1. **UV coordinates** form a grid with values between `0` and `1`. The center of the grid is at `[0.5, 0.5]`. UV coordinates are most often associated with mapping vertices to textures, as you'll see in Chapter 8, "Textures".
2. `fract(x)` returns the fractional part of `x`. You take the fractional value of the UVs multiplied by half the number of checks, which gives you a value between `0` and `1`. To center the UVs, you subtract `0.5`.
3. If the result of the `xy` multiplication is less than zero, then the result is `0` or black. Otherwise, it's `1` or white.

► Build and run the app.



*Checker board*

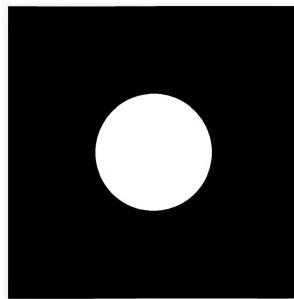
## length

Creating squares is a lot of fun, but let's create some circles using a length function.

- Replace the fragment function with:

```
float center = 0.5;
float radius = 0.2;
float2 uv = in.position.xy / params.width - center;
float3 color = step(length(uv), radius);
return float4(color, 1.0);
```

- Build and run the app.



*Circle*

To resize and move the shape around the screen, you change the circle's center and radius.

## smoothstep

`smoothstep(edge0, edge1, x)` returns a smooth Hermite interpolation between 0 and 1.

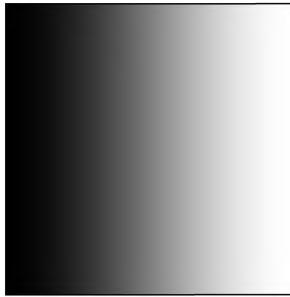
**Note:** `edge1` must be greater than `edge0`, and `x` should be `edge0 <= x <= edge1`.

- Change the fragment function to:

```
float color = smoothstep(0, params.width, in.position.x);
return float4(color, color, color, 1);
```

color contains a value between 0 and 1. When the position is the same as the screen width, the color is 0 or white. When the position is at the very left of the screen, the color is 0 or black.

- Build and run the app.



*smoothstep gradient*

Between the two edge cases, the color is a gradient interpolating between black and white. Here, you use `smoothstep` to calculate a color, but you can also use it to interpolate between any two values. For example, you can use `smoothstep` to animate a position in the vertex function.

## mix

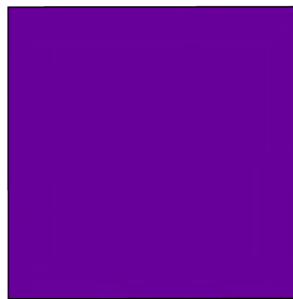
`mix(x, y, a)` produces the same result as  $x + (y - x) * a$ .

- Change the fragment function to:

```
float3 red = float3(1, 0, 0);
float3 blue = float3(0, 0, 1);
float3 color = mix(red, blue, 0.6);
return float4(color, 1);
```

A mix of 0 produces full red. A mix of 1 produces full blue. Together, these colors produce a 60% blend between red and blue.

- Build and run the app.



*A blend between red and blue*

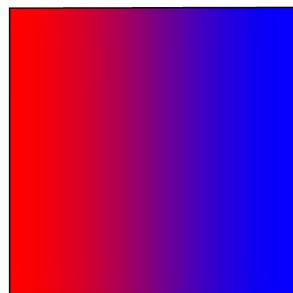
You can combine `mix` with `smoothstep` to produce a color gradient.

- Replace the fragment function with:

```
float3 red = float3(1, 0, 0);
float3 blue = float3(0, 0, 1);
float result = smoothstep(0, params.width, in.position.x);
float3 color = mix(red, blue, result);
return float4(color, 1);
```

This code takes the interpolated `result` and uses it as the amount to `mix` red and blue.

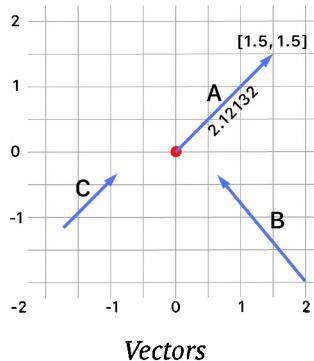
- Build and run the app.



*Combining smoothstep and mix*

## normalize

The process of normalization means to rescale data to use a standard range. For example, a vector has both direction and magnitude. In the following image, vector A has a length of 2.12132 and a direction of 45 degrees. Vector B has the same length but a different direction. Vector C has a different length but the same direction.



Vectors

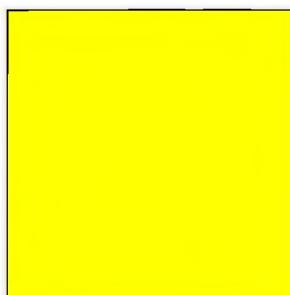
It's easier to compare the direction of two vectors if they have the same magnitude, so you **normalize** the vectors to a unit length. `normalize(x)` returns the vector `x` in the same direction but with a length of 1.

Let's look at another example of normalizing. Say you want to visualize the vertex positions using colors so that you can better debug some of your code.

► Change the fragment function to:

```
return in.position;
```

► Build and run the app.



Visualizing positions

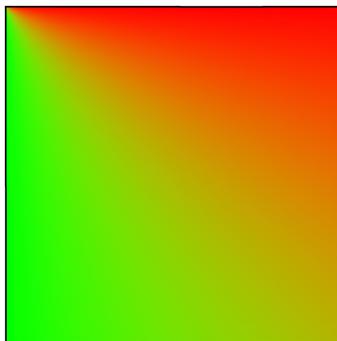
The fragment function *should* return an RGBA color with each element between 0 and 1. However, because the position is in screen space, each position varies between [0, 0, 0] and [800, 800, 0], which is why the quad renders yellow (it's only between 0 and 1 at the top-left corner).

► Now, change the code to:

```
float3 color = normalize(in.position.xyz);
return float4(color, 1);
```

Here, you normalize the vector `in.position.xyz` to have a length of 1. All of the colors are now guaranteed to be between 0 and 1. When normalized, the position (800, 0, 0) at the far top-right contains 1, 0, 0, which is red.

► Build and run the app to see the result.

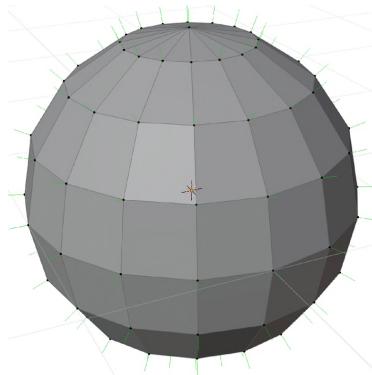


*Normalized positions*

## Normals

Although visualizing positions is helpful for debugging, it's not generally helpful in creating a 3D render. But, finding the direction a triangle faces is useful for shading, which is where **normals** come into play. Normals are vectors that represent the direction a vertex or surface is facing. In the next chapter, you'll learn how to light your models. But first, you need to understand normals.

The following image captured from Blender shows vertex normals pointing out. Each of the sphere's verteces points in a different direction.



*Vertex normals*

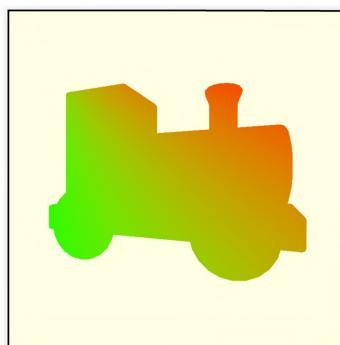
The shading of the sphere depends upon these normals. If a normal points toward the light source, Blender will shade brighter.

A quad isn't very interesting for shading purposes, so switch the default render to the train.

- Open **Options.swift**, and change the initialization of `renderChoice` to:

```
@Published var renderChoice = RenderChoice.train
```

- Preview the app to check your train render.



*Train render*

Unlike the full-screen quad, only fragments covered by the train will render. The color of each fragment, however, is still dependent upon the fragment's screen position and not the position of the train vertices.

## Loading the Train Model With Normals

3D model files generally contain surface normal values, and you can load these values with your model. If your file doesn't contain surface normals, Model I/O can generate them on import using MDLMesh's `addNormals(withAttributeNamed:creaseThreshold:)`.

## Adding Normals to the Vertex Descriptor

- Open `VertexDescriptor.swift`.

At the moment, you load only the position attribute. It's time to add the normal to the vertex descriptor.

- After the code that sets up `offset`, and before the code that sets `layouts [0]`, add the following code to `MDLVertexDescriptor`'s `defaultLayout`:

```
vertexDescriptor.attributes[1] = MDLVertexAttribute(  
    name: MDLVertexAttributeNormal,  
    format: .float3,  
    offset: offset,  
    bufferIndex: 0)  
offset += MemoryLayout<float3>.stride
```

Here, a normal is a `float3`, and interleaved with the position in buffer 0. `float3` is a typealias of `SIMD3<Float>` defined in `MathLibrary.swift`. Each vertex takes up two `float3`s, which is 32 bytes, in buffer index 0. `layouts [0]` describes buffer index 0 with the stride.

## Updating the Shader Functions

- Open `Vertex.metal`.

The pipeline state for the train model uses this vertex descriptor so that the vertex function can process the attributes, and you match the attributes with those in `VertexIn`.

- Build and run the app, and you'll see that everything still works as expected.

Even though you added a new attribute to the vertex buffer, the pipeline ignores it since you haven't included it as an `attribute(n)` in `VertexIn`. Time to fix that.

- Add the following code to `VertexIn`:

```
float3 normal [[attribute(1)]];
```

Here, you match `attribute(1)` with the vertex descriptor's attribute 1. So now you'll be able to access the normal attribute in the vertex function.

- Next, add the following code to `VertexOut`:

```
float3 normal;
```

By including the normal here, you can now pass the data on to the fragment function.

- In `vertex_main`, change the assignment to `out`:

```
float3 normal = in.normal;
VertexOut out {
    .position = position,
    .normal = normal
};
```

Perfect! With that change, you can now return both the position and the normal from the vertex function.

- Open `Fragment.metal`, and replace the contents of `fragment_main` with:

```
return float4(in.normal, 1);
```

Don't worry, that compile error is expected. Even though you updated `VertexOut` in `Vertex.metal`, the scope of that structure was only in that one file.

## Adding a Header

It's common to require structures and functions in multiple shader files. So, just as you did with the bridging header `Common.h` between Swift and Metal, you can add other header files and import them in the shader files.

- Create a new file using the macOS **Header File** template and name it `ShaderDefs.h`.

- Replace the code with:

```
#include <metal_stdlib>
using namespace metal;
```



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

Here, you define `VertexOut` within the `metal` namespace.

- Open `Vertex.metal`, and delete the `VertexOut` structure.
- After importing `Common.h`, add:

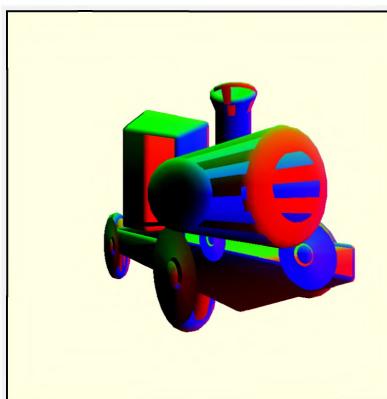
```
#import "ShaderDefs.h"
```

- Open `Fragment.metal`, and delete the `VertexOut` structure.
- Again, after importing `Common.h`, add:

```
#import "ShaderDefs.h"
```

- Build and run the app.

Oh, now that looks a little odd!



*Normals with rendering weirdness*

Your normals appear as if they are displaying correctly — red normals are at the train's right, green is up and blue is at the back — but as the train rotates, parts of it seem almost transparent.

The problem here is that the rasterizer is jumbling up the depth order of the vertices. When you look at a train from the front, you shouldn't be able to see the back of the train; it should be occluded.

# Depth

The rasterizer doesn't process depth order by default, so you need to give the rasterizer the information it needs with a **depth stencil state**.

As you may remember from Chapter 3, "The Rendering Pipeline", the Stencil Test unit checks whether fragments are visible after the fragment function, during the rendering pipeline. If a fragment is determined to be behind another fragment, it's discarded.

Let's give the render encoder an `MTLDepthStencilState` property to describe how to do this testing.

► Open `Renderer.swift`.

► Toward the end of `init(metalView:options:)`, after setting `metalView.clearColor`, add:

```
metalView.depthStencilPixelFormat = .depth32Float
```

This code tells the view that it needs to hold the depth information. The default pixel format is `.invalid`, which informs the view that it doesn't need to create a depth and stencil texture.

The pipeline state that the render command encoder uses has to have the same depth pixel format.

► In `init(metalView:options:)`, after setting `pipelineDescriptor.colorAttachments[0].pixelFormat`, before do {}, add:

```
pipelineDescriptor.depthAttachmentPixelFormat = .depth32Float
```

If you were to build and run the app now, you'd get the same result as before. However, behind the scenes, the view creates a texture to which the rasterizer can write depth values.

Next, you need to set how you want the rasterizer to calculate your depth values.

► Add a new property to `Renderer`:

```
let depthStencilState: MTLDepthStencilState?
```

This property holds the depth stencil state with the correct render settings.



- Create this method in Renderer to instantiate the depth stencil state:

```
static func buildDepthStencilState() -> MTLDepthStencilState? {  
    // 1  
    let descriptor = MTLDepthStencilDescriptor()  
    // 2  
    descriptor.depthCompareFunction = .less  
    // 3  
    descriptor.isDepthWriteEnabled = true  
    return Renderer.device.makeDepthStencilState(  
        descriptor: descriptor)  
}
```

Going through this code:

1. Create a descriptor that you'll use to initialize the depth stencil state, just as you did the pipeline state objects.
2. Specify how to compare the current and already processed fragments. With a compare function of `less`, if the current fragment depth is less than the depth of the previous fragment in the framebuffer, the current fragment replaces that previous fragment.
3. State whether to write depth values. If you have multiple passes, as you will in Chapter 12, “Render Passes”, sometimes you’ll want to read the already drawn fragments. In that case, set `isDepthWriteEnabled` to `false`. Note that `isDepthWriteEnabled` is always `true` when you’re drawing objects that require depth.

- Call the method from `init(metalView:options:)` before `super.init()`:

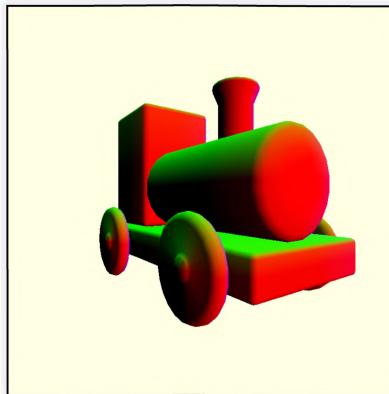
```
depthStencilState = Renderer.buildDepthStencilState()
```

- In `draw(in:)`, add this to the top of the method after `guard { }`:

```
renderEncoder.setDepthStencilState(depthStencilState)
```

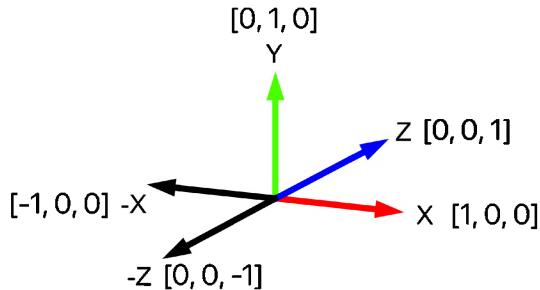
- Build and run the app to see your train in glorious 3D.

As the train rotates, it appears in shades of red, green, blue and black.



*Normals*

Consider what you see in this render. The normals are currently in object space. So, even though the train rotates in world space, the colors/normals don't change as the model changes its rotation.



*Normal colors along axes*

When a normal points to the right along the model's x-axis, the value is  $[1, 0, 0]$ . That's the same as red in RGB values, so the fragment is colored red for those normals pointing to the right.

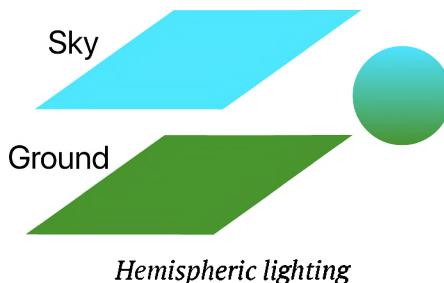
The normals pointing upwards are 1 on the y-axis, so the color is green.

The normals pointing toward the camera are negative. They're black when a color is  $[0, 0, 0]$  or less. When you see the back of the train as it rotates, you'll see that the normals pointing in the z direction are blue  $[0, 0, 1]$ .

Now that you have normals in the fragment function, you can start manipulating colors depending on the direction they're facing. Manipulating colors is important when you start playing with lighting.

## Hemispheric Lighting

Hemispheric lighting uses ambient light. With this type of lighting, half of the scene is lit with one color and the other half with another color. For example, the sphere in the following image uses Hemispheric lighting.



Notice how the sphere appears to take on the color reflected from the sky (top) and the color reflected from the ground (bottom). To see this type of lighting in action, you'll change the fragment function so that:

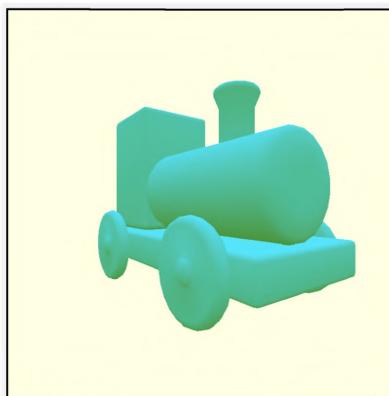
- Normals facing up are blue.
- Normals facing down are green.
- Interim values are a blue and green blend.

- Open **Fragment.metal**, and replace the contents of `fragment_main` with:

```
float4 sky = float4(0.34, 0.9, 1.0, 1.0);
float4 earth = float4(0.29, 0.58, 0.2, 1.0);
float intensity = in.normal.y * 0.5 + 0.5;
return mix(earth, sky, intensity);
```

`mix(x, y, z)` interpolates between the first two values depending on the third value, which must be between 0 and 1. Your normal values are between -1 and 1, so you convert the intensity between 0 and 1.

- Build and run the app to see your lit train. Notice how the top of the train is blue and its underside is green.



*Hemispheric lighting*

Fragment shaders are powerful, allowing you to color objects with precision. In Chapter 10, “Lighting Fundamentals”, you’ll use the power of normals to shade your scene with more realistic lighting. In Chapter 19, “Tessellation & Terrains”, you’ll create a similar effect to this one as you learn how to place snow on a terrain depending on the slope.

## Challenge

Currently, you're using hard-coded magic numbers for all of the buffer indices and attributes. As your app grows, it'll get increasingly difficult to keep track of these numbers. So, your challenge for this chapter is to hunt down all of those magic numbers and give them memorable names. For this challenge, you'll create an enum in **Common.h**.

Here's some code to get you started:

```
typedef enum {
    VertexBuffer = 0,
    UniformsBuffer = 11,
    ParamsBuffer = 12
} BufferIndices;
```

You can now use these constants in both Swift and C++ shader functions:

```
//Swift
encoder.setVertexBytes(
    &uniforms,
    length: MemoryLayout<Uniforms>.stride,
    index: Int(UniformsBuffer.rawValue))

// Shader Function
vertex VertexOut vertex_main(
    const VertexIn in [[stage_in]],
    constant Uniforms &uniforms [[buffer(UniformsBuffer)]])
```

You can even add an extension in **VertexDescriptor.swift** to prettify the code:

```
extension BufferIndices {
    var index: Int {
        return Int(self.rawValue)
    }
}
```

With this code, you can use `UniformsBuffer.index` instead of `Int(UniformsBuffer.rawValue)`.

You'll find the full solution in the challenge folder for this chapter.

## Key Points

- The fragment function is responsible for returning a color for each fragment that successfully passes through the rasterizer and the Stencil Test unit.
- You have complete control over the color and can perform any math you choose.
- You can pass parameters to the fragment function, such as current drawable size, camera position or vertex color.
- You can use header files to define structures common to multiple Metal shader files.
- Each fragment is stand-alone. You don't have access to neighboring fragments with the exception of the change of slope of the fragment. Fragments pass through the rasterizer in a 2 x 2 arrangement, and the partial derivatives of each fragment can be derived from the other fragment in the group of four. The MSL functions, `dfdx` and `dfdy`, return the horizontal and vertical changes in slope; and the function `fwidth` gives you the absolute derivative of the combined `dfdx` and `dfdy`.
- Check the Metal Shading Language Specification at <https://apple.co/3jDLQn4> for all of the MSL functions available in shader functions.
- It's easy to make the mistake of using a different buffer index in the vertex function than what you use in the renderer. Use descriptive enumerations for buffer indices.

## Where to Go From Here?

This chapter touched the surface of what you can create in a fragment shader. For more suggestions, have a look at **references.markdown**. The best source to start with is The Book of Shaders by Patricio Gonzalez (<https://thebookofshaders.com>).

# Chapter 8: Textures

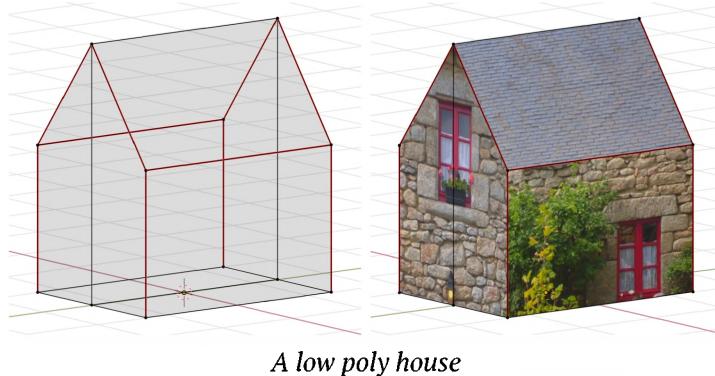
So far, you've learned how to use fragment functions and shaders to add colors and details to your models. Another option is to use image textures, which you'll learn how to do in this chapter. More specifically, you'll learn about:

- **UV coordinates:** How to unwrap a mesh so that you can apply a texture to it.
- **Texturing a model:** How to read the texture in a fragment shader.
- **Samplers:** Different ways you can read (sample) a texture.
- **Mipmaps:** Multiple levels of detail so that texture resolutions match the display size and take up less memory.
- **Asset catalog:** How to organize your textures.



## Textures and UV Maps

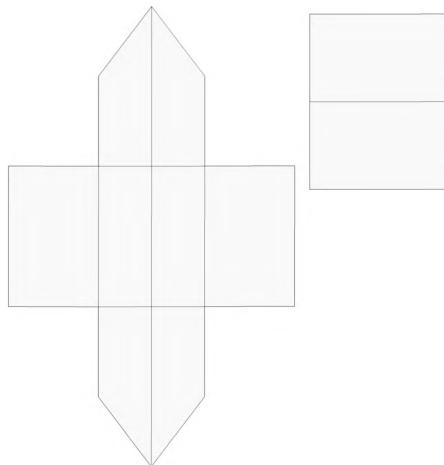
The following image shows a house model with twelve vertices. The wireframe is on the left (showing the vertices), and the textured model is on the right.



A low poly house

**Note:** If you want a closer look at this model, you'll find the Blender and .obj files in the **resources/LowPolyHouse** folder for this chapter.

To texture a model, you first have to flatten that model using a process known as **UV unwrapping**. UV unwrapping creates a **UV map** by *unfolding* the model. To unfold the model, you mark and cut seams using a modeling app. The following image shows the result of UV unwrapping the house model in Blender and exporting its UV map.

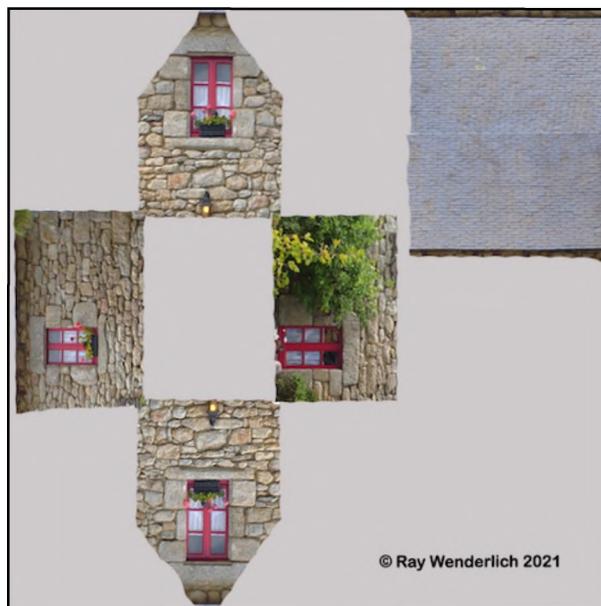


The house UV map



Notice that the roof and walls have marked seams. Seams are what make it possible for this model to lie flat. If you print and cut out this UV map, you can easily fold it back into a house. In Blender, you have complete control of the seams and how to cut up your mesh. Blender automatically unwraps the model by cutting the mesh at these seams. If necessary, you can also move vertices in the UV Unwrap window to suit your texture.

Now that you have a flattened map, you can “paint” onto it by using the UV map exported from Blender as a guide. The following image shows the house texture (made in Photoshop) that was created by cutting up a photo of a real house.



*Low poly house color texture*

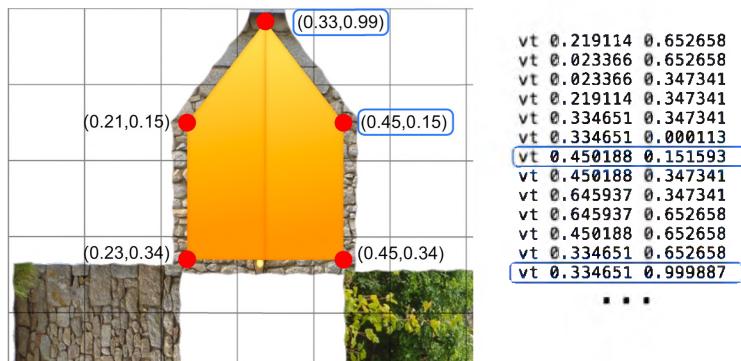
Note how the edges of the texture aren't perfect, and the copyright message is visible. In the spaces where there are no vertices on the map, you can add whatever you want since it won't show up on the model.

**Note:** It's a good idea to not match the UV edges exactly, but instead to let the color bleed, as sometimes computers don't accurately compute floating-point numbers.

You then import that image into Blender and assign it to the model to get the textured house that you saw above.

When you export a UV mapped model to an .obj file, Blender adds the UV coordinates to the file. Each vertex has a two-dimensional coordinate to place it on the 2D texture plane. The top-left is  $(0, 1)$  and the bottom-right is  $(1, 0)$ .

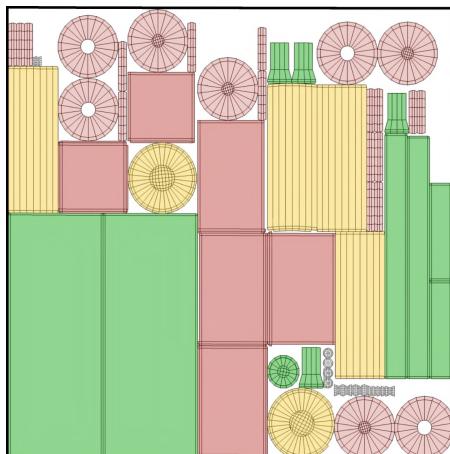
The following diagram indicates some of the house vertices, with the matching coordinates from the .obj file. You can look at the contents of the .obj file usingTextEdit.



*UV coordinates*

One of the advantages of mapping from  $0$  to  $1$  is that you can swap in lower or higher resolution textures. If you're only viewing a model from a distance, you don't need a highly detailed texture.

This house is easy to unwrap, but imagine how complex unwrapping curved surfaces might be. The following image shows the UV map of the train (which is still a simple model):



*The train's UV map*



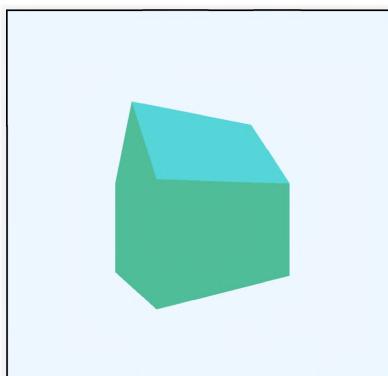
Photoshop, naturally, is not the only solution for texturing a model. You can use any image editor for painting on a flat texture. In the last few years, several other apps that allow painting directly on the model have become mainstream:

- **Blender** (free)
- **Substance Designer** and **Substance Painter** by Adobe (\$\$): In Designer, you can create complex materials procedurally. Using Substance Painter, you can paint these materials on the model.
- **3DCoat** by 3Dcoat.com (\$\$)
- **Mudbox** by Autodesk (\$\$)
- **Mari** by Foundry (\$\$\$)

In addition to texturing, using Blender, 3DCoat or Mudbox, you can sculpt models in a similar fashion to ZBrush and create low poly models from the high poly sculpt. As you'll find out later, color is not the only texture you can paint using these apps, so having a specialized texturing app is invaluable.

## The Starter App

- Open the starter project for this chapter, and build and run the app.



*The starter app*

The scene contains the low poly house. The fragment shader code is the same code from the challenge in the previous chapter, with hemispheric lighting added and a different background color. The vertex and fragment shaders are combined in **Shaders.metal**.

The other major changes are:

- **Mesh.swift** and **Submesh.swift** extract the Model I/O and MetalKit mesh buffers into custom vertex buffers and submesh groups. Model now contains an array of Meshs in place of an MTKMesh. Abstracting away from the Metal API allows for greater flexibility when generating models that don't use Model I/O and MetalKit. Remember, it's your engine, so you can choose how to hold the mesh data.
- **VertexDescriptor.swift** contains a UV attribute. Model loads UVs in the same way as you loaded normals in the previous chapter. Notice how the UVs will go into a separate buffer from the position and normal. This isn't necessary, but it makes the layout more flexible for use with custom-generated models.
- **Renderer.swift** passes uniforms and params to Model to perform the rendering code.

► Open **Shaders.metal**.

**VertexIn** and **VertexOut** contain the **uv** property. The vertex function passes the interpolated UV to the fragment function. This process is the same as adding the normal in the previous chapter.

In this chapter, you'll replace the sky and earth colors in the fragment function with colors from the texture. Initially, you'll use **lowpoly-house-color.png** located in the group **Models > LowPolyHouse**. To read the texture in the fragment function, you'll take the following steps:

1. Load and store the image texture centrally.
2. Pass the loaded texture to the fragment function before drawing the model.
3. Change the fragment function to read the appropriate pixel from the texture.

## 1. Loading the Texture

A model typically has several submeshes that reference one or more textures. Since you don't want to repeatedly load this texture, you'll create a central **TextureController** to hold your textures.



- Create a new Swift file named **TextureController.swift**. Be sure to include the new file in both the macOS and iOS targets. Replace the code with:

```
import MetalKit

enum TextureController {
    static var textures: [String: MTLTexture] = [:]
```

**TextureController** will grab the textures used by your models and hold them in this dictionary.

- Add a new method to **TextureController**:

```
static func loadTexture(filename: String) throws -> MTLTexture?
{
    // 1
    let textureLoader = MTKTextureLoader(device: Renderer.device)
    // 2
    let textureLoaderOptions: [MTKTextureLoader.Option: Any] =
        [.origin: MTKTextureLoader.Origin.bottomLeft]
    // 3
    let fileExtension =
        URL(fileURLWithPath: filename).pathExtension.isEmpty ?
            "png" : nil
    // 4
    guard let url = Bundle.main.url(
        forResource: filename,
        withExtension: fileExtension)
    else {
        print("Failed to load \(filename)")
        return nil
    }
    let texture = try textureLoader.newTexture(
        URL: url,
        options: textureLoaderOptions)
    print("loaded texture: \(url.lastPathComponent)")
    return texture
}
```

Going through the code:

1. Create a texture loader using MetalKit's **MTKTextureLoader**.
2. Change the texture's origin option to ensure that the texture loads with its origin at the bottom-left, which is what you need for **lowpoly-house-color.png**.

3. Provide a default extension for the image name.
4. Create a new `MTLTexture` using the provided image name and loader options. Also, return the newly created texture, and for debugging purposes, print the name of the loaded texture.

**Note:** Loading textures can get complicated. When Metal was first released, you had to specify everything about the image — like pixel format, dimensions and usage — using `MTLTextureDescriptor`. However, with MetalKit’s `MTKTextureLoader`, you can use the provided default values and optionally change them as needed.

- Add a new method to `TextureController`:

```
static func texture(filename: String) -> MTLTexture? {  
    if let texture = textures[filename] {  
        return texture  
    }  
    let texture = try? loadTexture(filename: filename)  
    if texture != nil {  
        textures[filename] = texture  
    }  
    return texture  
}
```

Here, you return a reference to the texture, and if the filename is new, you save the new texture to the central texture dictionary.

## Loading the Submesh Texture

Each submesh of a model’s mesh has a different material characteristic, such as roughness, base color and metallic content. For now, you’ll focus only on the base color texture. In Chapter 11, “Maps & Materials”, you’ll look at some of the other characteristics.

Conveniently, Model I/O loads a model complete with all the materials.

► Open `lowpoly-house.mtl` in the group **Models** ▶ **LowPolyHouse**. To see the text in the file, you may have to right-click the file and choose **Open with External Editor** from the menu.

The Kd value holds the diffuse material color — in this case, a light gray. At the very bottom of the file, you'll see `map_Kd lowpoly-house-color.png`. This gives Model I/O the diffuse color map file name.

```
# Blender MTL File: 'lowpoly-house.blend'
# Material Count: 1

newmtl Material
Ns 96.078431
Ka 1.000000 1.000000 1.000000
Kd 0.640000 0.640000 0.640000
Ks 0.500000 0.500000 0.500000
Ke 0.000000 0.000000 0.000000
Ni 1.000000
d 1.000000
illum 2
map_Kd lowpoly-house-color.png
```

*The .obj's material file*

- Open `Submesh.swift`, and inside `Submesh`, create a structure and a property to hold the textures:

```
struct Textures {
    let baseColor: MTLTexture?
}

let textures: Textures
```

Don't worry about compile errors; your project won't compile until you've initialized textures.

`MDLSubmesh` holds each submesh's material information in an `MDLMaterial` property. You provide the material with a `semantic` to retrieve the value for the relevant material. For example, the semantic for base color is `MDLMaterialSemantic.baseColor`.

- At the end of `Submesh.swift`, add an initializer for `Textures`:

```
private extension Submesh.Textures {
    init(material: MDLMaterial?) {
        func property(with semantic: MDLMaterialSemantic)
            -> MTLTexture? {
            guard let property = material?.property(with: semantic),
                  property.type == .string,
                  let filename = property.stringValue,
                  let texture =
                      TextureController.texture(filename: filename)
            else { return nil }
            return texture
    }
}
```



```
    baseColor = property(with: MDLMaterialSemantic.baseColor)
}
```

`property(with:)` looks up the provided property in the submesh's material, finds the filename string value of the property and returns a texture if there is one. Remember, there was another material property in the file marked `Kd`. That was the base color using floats. Material properties can also be float values where there is no texture available for the submesh.

This loads the base color texture with the submesh's material. Here, **Base color** means the same as **diffuse**. Later, you'll load other textures for the submesh in the same way.

- At the bottom of `init(mdlSubmesh:mtkSubmesh)` add:

```
textures = Textures(material: mdlSubmesh.material)
```

This code completes the initialization and removes the compiler warning.

- Build and run your app to check that everything's working. Your model will look the same as in the initial screenshot. However, you'll get a message in the console:

```
loaded texture: lowpoly-house-color.png
```

The texture loader has successfully loaded `lowpoly-house-color.png`.

## 2. Passing the Loaded Texture to the Fragment Function

In a later chapter, you'll learn about several other texture types and how to send them to the fragment function using different indices.

- Open `Common.h`, and add a new enumeration to keep track of these texture buffer index numbers:

```
typedef enum {
    BaseColor = 0
} TextureIndices;
```

- Open `VertexDescriptor.swift`, and add this code to the end of the file:

```
extension TextureIndices {
    var index: Int {
```

```
    return Int(self.rawValue)
}
```

This code allows you to use `BaseColor.index` instead of `Int(BaseColor.rawValue)`. A small touch, but it makes your code easier to read.

► Open `Model.swift`.

In `render(encoder:uniforms:params:)` where you process the submeshes, add the following code below the comment `// set the fragment texture here:`:

```
encoder.setFragmentTexture(
    submesh.textures.baseColor,
    index: BaseColor.index)
```

You're now passing the texture to the fragment function in texture buffer 0.

**Note:** Buffers, textures and sampler states are held in argument tables. As you've seen, you access these things by index numbers. On iOS, you can hold at least 31 buffers and textures, and 16 sampler states in the argument table; the number of textures on macOS increases to 128. You can find out feature availability for your device in Apple's Metal Feature Set Tables (<https://apple.co/2UpCT8r>).

### 3. Updating the Fragment Function

► Open `Shaders.metal`, and add the following new argument to `fragment_main`, immediately after `VertexOut in [[stage_in]],:`

```
texture2d<float> baseColorTexture [[texture(BaseColor)]]
```

You're now able to access the texture on the GPU.

► Replace all the code in `fragment_main` with:

```
constexpr sampler textureSampler;
```

When you read or **sample** the texture, you may not land precisely on a particular pixel. In texture space, the units that you sample are known as **texels**, and you can decide how each texel is processed using a **sampler**. You'll learn more about samplers shortly.



- Next, add this:

```
float3 baseColor = baseColorTexture.sample(  
    textureSampler,  
    in.uv).rgb;  
return float4(baseColor, 1);
```

Here, you sample the texture using the interpolated UV coordinates sent from the vertex function, and you retrieve the RGB values. In Metal Shading Language, you can use `rgb` to address the float elements as an equivalent of `xyz`. You then return the texture color from the fragment function.

- Build and run the app to see your textured house.



*The textured house*

## sRGB Color Space

You'll notice that the rendered texture looks much darker than the original image. This change in color happens because `lowpoly-house-color.png` is an sRGB texture. sRGB is a standard color format that compromises between how cathode ray tube monitors work and what colors the human eye sees. As you can see in the following example of grayscale values from 0 to 1, sRGB colors are not linear. Humans are more able to discern between lighter values than darker ones.



Unfortunately, it's not easy to do the math on colors in a non-linear space. If you multiply a color by  $0.5$  to darken it, the difference in sRGB will vary along the scale.

You're currently loading the texture as sRGB pixel data and rendering it into a linear color space. So when you're sampling a value of, say  $0.2$ , which in sRGB space is mid-gray, the linear space will read that as dark-gray.

To *approximately* convert the color, you can use the inverse of **gamma 2.2**:

```
sRGBcolor = pow(linearColor, 1.0/2.2);
```

If you use this formula on `baseColor` before returning from the fragment function, your house texture will look about the same as the original sRGB texture. However, a better way of dealing with this problem is not to load the texture as sRGB at all.

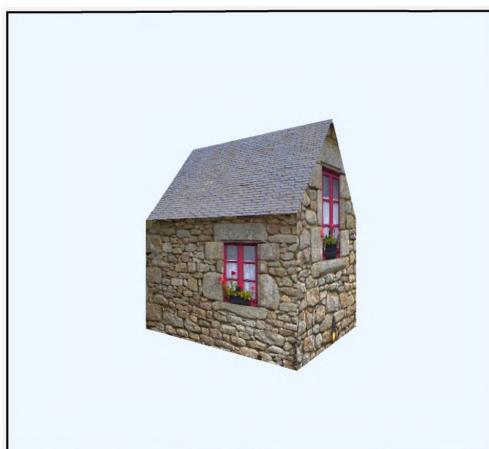
► Open `TextureController.swift`, and in `loadTexture(filename:)`, locate:

```
let textureLoaderOptions: [MTKTextureLoader.Option: Any] = [  
    .origin: MTKTextureLoader.Origin.bottomLeft]
```

► Change it to:

```
let textureLoaderOptions: [MTKTextureLoader.Option: Any] = [  
    .origin: MTKTextureLoader.Origin.bottomLeft,  
    .SRGB: false  
]
```

► Build and run the app, and the texture now loads with the linear color pixel format `bgra8Unorm`.



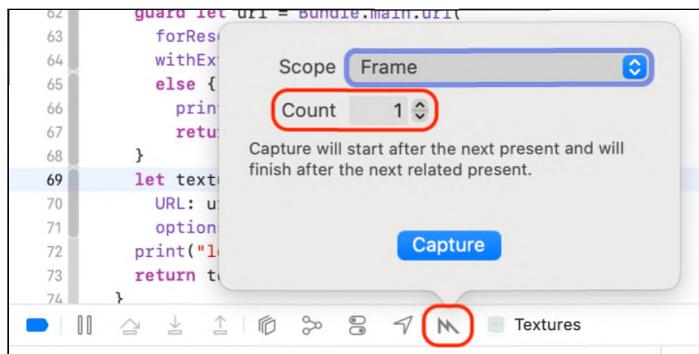
*Linear workflow*

**Note:** An alternative to loading the textures with SRGB as `false` is to change the MTKView's `colorPixelFormat` to `bgra8Unorm_srgb`. This change will affect the view's color space, and the clear color background will also change. You'll find further reading on chromaticity and color in `references.markdown` in the resources folder for this chapter.

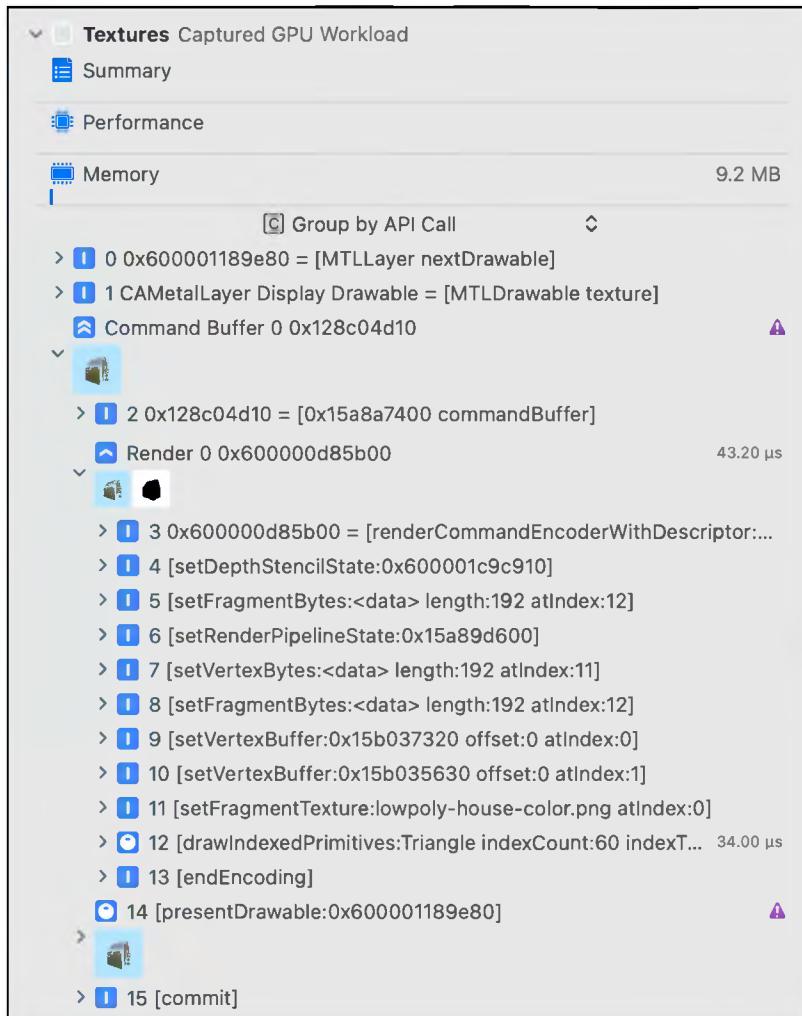
## Capture GPU Workload

There's an easy way to find out what format your texture is in on the GPU, and also to look at all the other Metal buffers currently residing there: the **Capture GPU workload** tool (also called the **GPU Debugger**).

- Run your app, and at the bottom of the Xcode window (or above the debug console if you have it open), click the **M Metal** icon, change the number of frames to count to 1, and click **Capture** in the pop-up window:



This button captures the current GPU frame. On the left in the Debug navigator, you'll see the GPU trace:



A GPU trace

**Note:** To open or close all items in a hierarchy, you can Option-click the arrow.

You can see all the commands that you've given to the render command encoder, such as `setFragmentBytes` and `setRenderPipelineState`. Later, when you have several command encoders, you'll see each one of them listed, and you can select them to see what actions or textures they have produced from their encoding.

When you select `drawIndexedPrimitives`, the **Vertex** and **Fragment** resources show.

Label	Type	Size	Details	Parameter Name	Resource...
<b>Pipeline States</b>					
Render Pipeline 0x15a89...					
<b>Vertex</b>					
MDL_OBJ-Indices	Index	240 bytes	Offset: 0x0		
Buffer 0x15b037320	Buffer 0	1 KB	Offset: 0x0	vertexBuffer.0	Read
Buffer 0x15b035630	Buffer 1	272 bytes	Offset: 0x0	vertexBuffer.1	Read
Vertex Bytes	Buffer 11 (Bytes)	192 bytes		uniforms	Read
Geometry	Post Vertex Tran...				
Vertex Attributes	Vertex Attributes				
vertex_main	Vertex Function		Library 0x600002ac0940...		
<b>Fragment</b>					
lowpoly-house-color.png	Texture 0	1024 x 1024	BGRA8Unorm	baseColorTexture	Read
Fragment Bytes	Buffer 12 (Bytes)	192 bytes		params	Read
fragment_main	Fragment Function		Library 0x600002ac0940...		
<b>Attachments</b>					
CAMetalLayer Display Dra...	Color 0	800 x 800	BGRA8Unorm		Write
MTKView Depth	Depth	800 x 800	Depth32Float		Write

### Resources on the GPU

- Double-click each vertex resource to see what's in the buffer:
  - **MDL\_OBJ-Indices:** The vertex indices.
  - **Buffer 0:** The vertex position and normal data, matching the attributes of your `VertexIn` struct and the vertex descriptor.
  - **Buffer 1:** The UV texture coordinate data.
  - **Vertex Bytes:** The uniform matrices.
  - **Vertex Attributes:** The incoming data from `VertexIn`, and the `VertexOut` return data from the vertex function.
  - **vertex\_main:** The vertex function. When you have multiple vertex functions, this is very useful to make sure that you set the correct pipeline state.

Going through the fragment resources:

- **lowpoly-house-color.png**: The house texture in texture slot **0**.
- **Fragment Bytes**: The width and height screen parameters in **params**.
- **fragment\_main**: The fragment function.

The attachments:

- **CAMetalLayer Drawable**: The result of the encoding in color attachment **0**. In this case, this is the view's current drawable. Later, you'll use multiple color attachments.
- **MTKView Depth**: The depth buffer. Black is closer. White is farther. The rasterizer uses the depth map.

You can see from this list that the GPU is holding the **lowpoly-house-color.png** texture as **BGRA8Unorm**. If you reverse the previous section's texture loading options and comment out **.SRGB: false**, you'll be able to see that the texture is now **BGRA8Unorm\_sRGB**. (Make sure you restore the option **.SRGB: false** before continuing.)

If you're ever uncertain as to what is happening in your app, capturing the GPU frame might give you the heads-up because you can examine every render encoder command and every buffer. It's a good idea to use this strategy throughout this book to examine what's happening on the GPU.

## Samplers

When sampling your texture in the fragment function, you use a default sampler. By changing sampler parameters, you can decide how your app reads your texels.

You'll now add a ground plane to your scene to see how you can control the appearance of the ground texture.

► Open **Renderer.swift**, and add a new property:

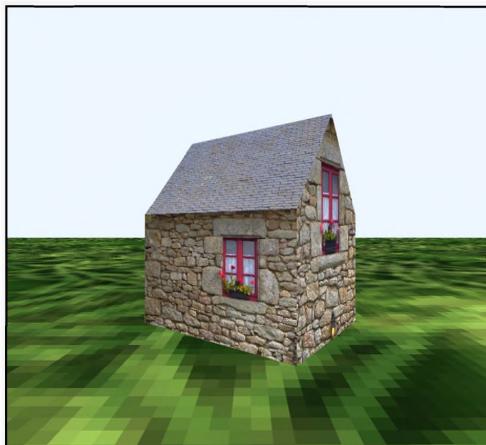
```
lazy var ground: Model = {
    Model(name: "plane.obj")
}()
```

In `draw(in:)` after rendering the house and before `renderEncoder.endEncoding()`, add:

```
ground.scale = 40
ground.rotation.y = sin(timer)
ground.render(
    encoder: renderEncoder,
    uniforms: uniforms,
    params: params)
```

This code adds a ground plane and scales it up.

- Build and run the app.



*A stretched texture*

The ground texture stretches to fit the ground plane, and each pixel in the texture may be used by several rendered fragments, giving it a pixellated look. By changing one of the sampler parameters, you can tell Metal how to process the texel where it's smaller than the assigned fragments.

- Open `Shaders.metal`. In `fragment_main`, change the `textureSampler` definition to:

```
constexpr sampler textureSampler(filter::linear);
```

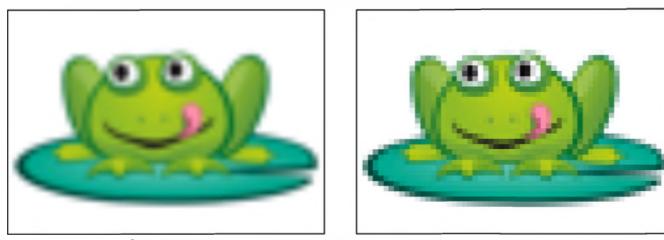
This code instructs the sampler to smooth the texture.

- Build and run the app.



*A smoothed texture*

The ground texture — although still stretched — is now smooth. There will be times, such as when you make a retro game of Frogger, that you'll want to keep the pixelation. In that case, use **nearest** filtering.



Linear

Nearest

*Filtering*

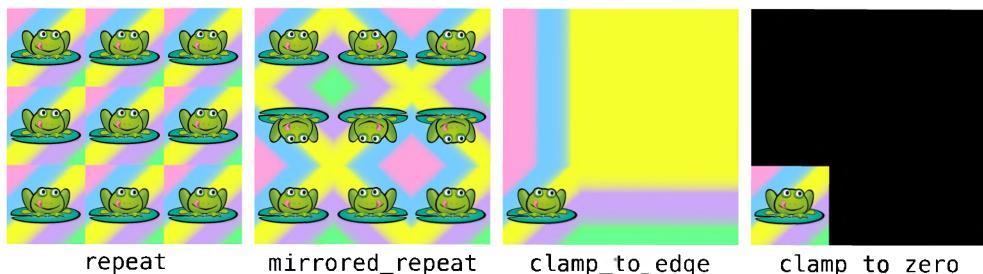
In this particular case, however, you want to tile the texture. That's easy with sampling.

- Change the sampler definition and the `baseColor` assignment to:

```
constexpr sampler textureSampler(
    filter::linear,
    address::repeat);
float3 baseColor = baseColorTexture.sample(
    textureSampler,
    in.uv * 16).rgb;
```

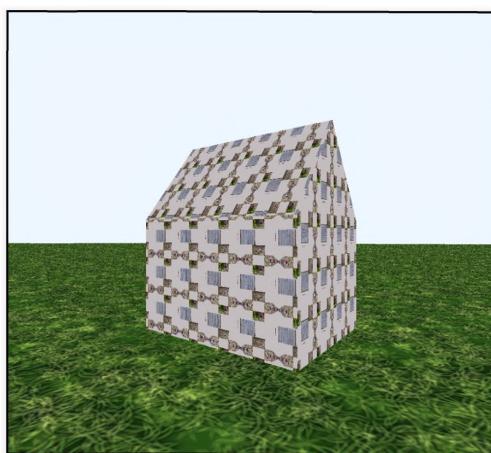
This code multiplies the UV coordinates by 16 and accesses the texture outside of the allowable limits of 0 to 1. `address::repeat` changes the sampler's addressing mode, so it'll repeat the texture 16 times across the plane.

The following image illustrates the other `address` sampling options shown with a tiling value of 3. You can use `s_address` or `t_address` to change only the width or height coordinates, respectively.



*The sampler address mode*

- Build and run your app.



*Texture tiling*

The ground looks great! The house... not so much. The shader has tiled the house texture as well. To overcome this problem, you'll create a `tiling` property on the model and send it to the fragment function with `params`.

- In `Common.h`, add this to `Params`:

```
uint tiling;
```

- In **Model.swift**, create a new property in **Model**:

```
var tiling: UInt32 = 1
```

- In **render(encoder:uniforms:params:)**, just after **var params = fragment**, add this:

```
params.tiling = tiling
```

- In **Renderer.swift**, replace the declaration of **ground** with:

```
lazy var ground: Model = {
    var ground = Model(name: "plane.obj")
    ground.tiling = 16
    return ground
}()
```

You're now sending the model's tiling factor to the **fragment** function.

- Open **Shaders.metal**. In **fragment\_main**, replace the declaration of **baseColor** with:

```
float3 baseColor = baseColorTexture.sample(
    textureSampler,
    in.uv * params.tiling).rgb;
```

- Build and run the app, and you'll see that both the ground and house now tile correctly.



*Corrected tiling*

**Note:** Creating a sampler in the shader is not the only option. You can create an `MTLSamplerState`, hold it with the model and send the sampler state to the fragment function with the `[[sampler(n)]]` attribute.

As the scene rotates, you'll notice some distracting noise. You've seen what happens on the grass when you oversample a texture. But, when you undersample a texture, you can get a rendering artifact known as **moiré**, which is occurring on the roof of the house.



*A moiré example*

In addition, the noise at the horizon almost looks as if the grass is sparkling. You can solve these artifact issues by sampling correctly using resized textures called **mipmaps**.

## Mipmaps

Check out the relative sizes of the roof texture and how it appears on the screen.



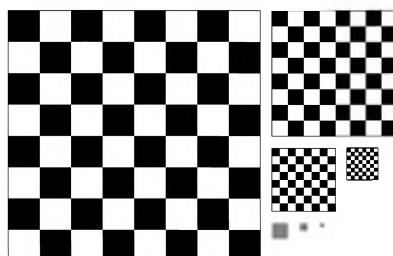
*Size of texture compared to on-screen viewing*

The pattern occurs because you're sampling more texels than you have pixels. The ideal would be to have the same number of texels to pixels, meaning that you'd require smaller and smaller textures the further away an object is. The solution is to use **mipmaps**. Mipmaps let the GPU compare the fragment on its depth texture and sample the texture at a suitable size.

**MIP** stands for *multum in parvo* — a Latin phrase meaning “*much in small*”.

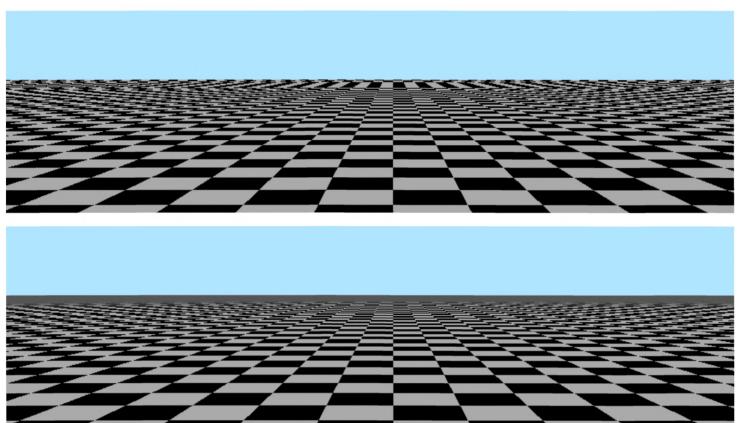
Mipmaps are texture maps resized down by a power of 2 for each level, all the way down to 1 pixel in size. If you have a texture of 64 pixels by 64 pixels, then a complete mipmap set would consist of:

**Level 0:** 64 x 64, **1:** 32 x 32, **2:** 16 x 16, **3:** 8 x 8, **4:** 4 x 4, **5:** 2 x 2, **6:** 1 x 1.



*Mipmaps*

In the following image, the top checkered texture has no mipmaps. But in the bottom image, every fragment is sampled from the appropriate MIP level. As the checkers recede, there's much less noise, and the image is cleaner. At the horizon, you can see the solid color smaller gray mipmaps.



*Mipmap comparison*

You can easily and automatically generate these mipmaps when first loading the texture.

► Open **TextureController.swift**. In `loadTexture(filename:)`, change the texture loading options to:

```
let textureLoaderOptions: [MTKTextureLoader.Option: Any] = [
    .origin: MTKTextureLoader.Origin.bottomLeft,
    .SRGB: false,
    .generateMipmaps: NSNumber(value: true)
]
```

This code will create mipmaps all the way down to the smallest pixel.

There's one more thing to change: the sampler.

► Open **Shaders.metal**, and add the following code to the construction of `textureSampler`:

```
mip_filter::linear
```

The default for `mip_filter` is `none`. However, if you provide either `.linear` or `.nearest`, then the GPU will sample the correct mipmap.

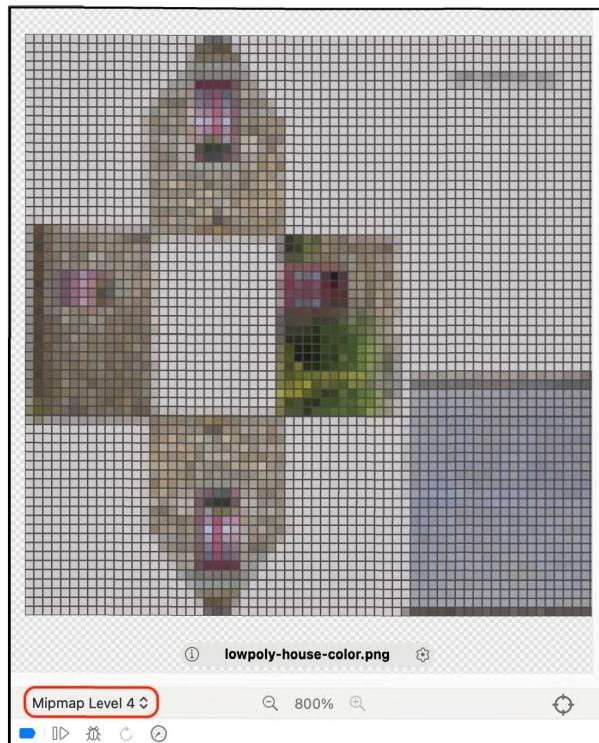
► Build and run the app.



*Mipmaps added*

The noise from both the building and the ground is gone.

Using the **Capture GPU workload** tool, you can inspect the mipmaps. Choose the draw call, and double-click a texture. At the bottom-left, you can choose the MIP level. This is MIP level 4 on the house texture:



Mipmap level 4 example

## Anisotropy

Your rendered ground is looking a bit muddy and blurred in the background. This is due to **anisotropy**. Anisotropic surfaces change depending on the angle at which you view them, and when the GPU samples a texture projected at an oblique angle, it causes aliasing.

► In **Shaders.metal**, add this to the construction of `textureSampler`:

```
max_anisotropy(8)
```

Metal will now take eight samples from the texel to construct the fragment. You can specify up to 16 samples to improve quality. Use as few as you can to obtain the quality you need because the sampling can slow down rendering.

**Note:** As mentioned before, you can hold an `MTLSamplerState` on `Model`. If you increase anisotropy sampling, you may not want it on all models, and this might be a good reason for creating the sampler state outside the fragment shader.

- Build and run, and your render should be artifact-free.



*Anisotropy*

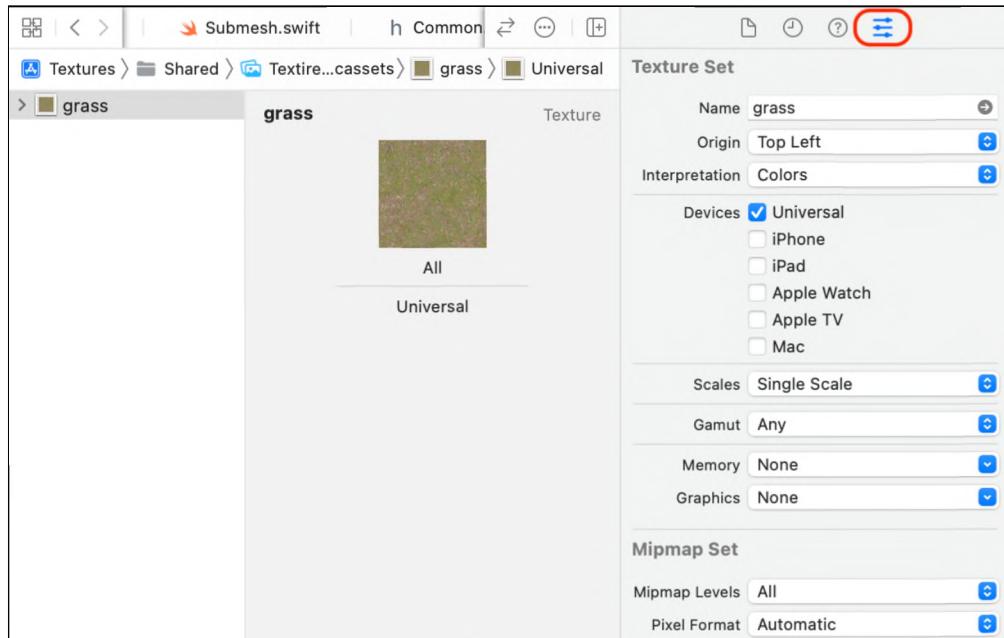
When you write your full game, you're likely to have many textures for the different models. Some models are likely to have several textures. Organizing these textures and working out which ones need mipmaps can become labor-intensive. Plus, you'll also want to compress images where you can and send textures of varying sizes and color gamuts to different devices. The asset catalog is where you'll turn.

## The Asset Catalog

As its name suggests, the asset catalog can hold all of your assets, whether they be data, images, textures or even colors. You've probably used the catalog for app icons and images. Textures differ from images in that the GPU uses them, and thus they have different attributes in the catalog. To create textures, you add a new texture set to the asset catalog.

You'll now replace the textures for the low poly house and ground and use textures from a catalog.

- Create a new file using the **Asset Catalog** template (found in the **Resource** section), and name it **Textures**. Remember to check both the iOS and macOS targets.
- With **Textures.xcassets** open, choose **Editor** ▶ **Add New Asset** ▶ **AR and Textures** ▶ **Texture Set** (or click the + at the bottom of the panel and choose **AR and Textures** ▶ **Texture Set**).
- Double-click the **Texture** name and rename it to **grass**.
- Open the **Models** ▶ **Textures** group and drag **barn-ground.png** to the **Universal** slot in your catalog. With the **Attributes inspector** open, click on the grass to see all of the texture options.



*Texture options in the asset catalog*

Here, you can see that by default, all mipmaps are created automatically. If you change **Mipmap Levels** to **Fixed**, you can choose how many levels to make. If you don't like the automatic mipmaps, you can replace them with your own custom mipmaps by dragging them to the correct slot.

Asset catalogs give you complete control of your textures without having to write cumbersome code, although you can still write the code using the `MTLTextureDescriptor` API if you want.



*Mipmap slots*

Now that you're using named textures from the asset catalog instead of .png files, you'll need to change your texture loader.

► Open `TextureController.swift`, and at the top of `loadTexture(filename:)`, after defining `textureLoader`, add this:

```
if let texture = try? textureLoader.newTexture(
    name: filename,
    scaleFactor: 1.0,
    bundle: Bundle.main,
    options: nil) {
    print("loaded texture: \(filename)")
    return texture
}
```

This now searches the bundle for the named texture and loads it if there is one. When loading from the asset catalog, the options that you set in the Attributes inspector take the place of most of the texture loading options, so these options are now `nil`.

The last thing to do is to make sure the model points to the new texture.

► Open `plane.mtl`, located in **Models > Ground**. If your file is not text-editable, you can right-click the file and choose **Open in External Editor**.

► Replace:

```
map_Kd ground.png
```

► With:

```
#map_Kd ground.png
map_Kd grass
```

Here, you commented out the old texture and added the new one. The **grass** texture will now load from the asset catalog in place of the old one.

► Repeat this for the low poly house to change it into a barn:

1. Create a new texture set in the asset catalog and rename it **barn**.
2. Drag **lowpoly-barn-color.png** into the texture set from the **Models ▶ Textures** group.
3. Change the name of the diffuse texture in **Models ▶ LowPolyHouse ▶ lowpoly-house.mtl** to barn.

**Note:** Be careful to drop the images on the texture's Universal slot. If you drag the images into the asset catalog, they are, by default, **images** and not **textures**. And you won't be able to make mipmaps on images or change the pixel format.

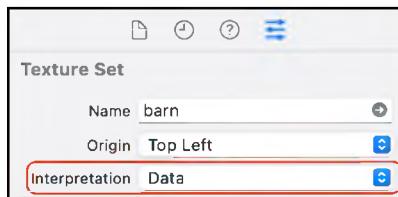
► Build and run your app to see your new textures.



*SRGB rendering*

You can see that the textures have reverted to the sRGB space because you're now loading them in their original format. You can confirm this using the **Capture GPU workload** tool.

- Open **Textures.xcassets**, click on the barn texture, and in the **Attributes inspector**, change the **Interpretation** to **Data**:



*Convert texture to data*

When your app loads the sRGB texture to a non-sRGB buffer, it automatically converts from sRGB space to linear space. (See Apple's Metal Shading Language document for the conversion rule.) By accessing as data instead of colors, your shader can treat the color data as linear.

You'll also notice in the above image that the origin — unlike loading the .png texture manually — is **Top Left**. The asset catalog loads textures differently.

- Repeat for the grass texture.
- Build and run, and your colors should now be correct.

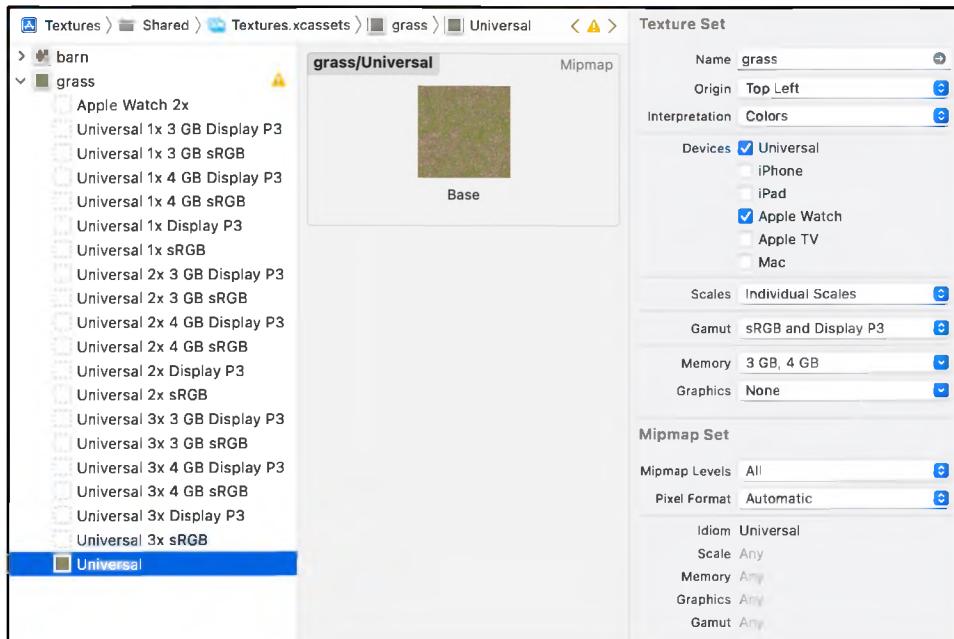


*Corrected color space*

## The Right Texture for the Right Job

Using asset catalogs gives you complete control over how to deliver your textures. Currently, you only have two color textures. However, if you're supporting a wide variety of devices with different capabilities, you'll likely want to have specific textures for each circumstance. On devices with less RAM, you'd want smaller graphics.

For example, here is a list of individual textures you can assign by checking the different options in the Attributes inspector, for the Apple Watch, devices with 3GB and 4GB memory, and sRGB and P3 displays.



*Custom textures in the asset catalog*

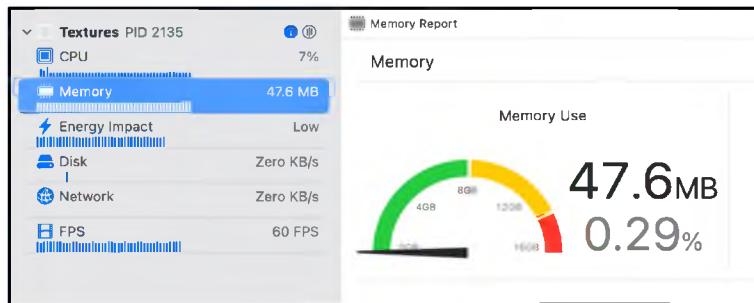
## Texture Compression

In recent years, people have put much effort into compressing textures to save both CPU and GPU memory. There are various formats you can use, such as ETC and PVRTC. Apple has embraced ASTC as being the most high-quality compressed format. ASTC is available on the A8 chip and newer.

Using texture sets within the asset catalog allows your app to determine for itself which is the best format to use.

With your app running on macOS, take a look at how much memory it's consuming.

- Click on the **Debug navigator** and select **Memory**.



*App memory usage*

This is the usage after 45 seconds — your app’s memory consumption will increase for about five minutes and then stabilize.

If you capture the frame with the **Capture GPU Workload** button, you’ll see that the texture format on the GPU is **RGBA8Unorm**. When you use asset catalogs, Apple will automatically determine the most appropriate format for your texture.

- In **Textures.xcassets**, select each of your textures, and in the **Attributes inspector**, change the Pixel Format from **Automatic** to **ASTC 8×8 Compressed - Red Green Blue Alpha**. This is a highly compressed format.

- Build and run your app, and check the memory usage again.

You’ll see that the memory footprint is slightly reduced. However, so is the quality of the render. For distant textures, this quality might be fine, but you have to balance memory usage with render quality.



Automatic

ASTC 8x8

*Compressed texture comparison*

**Note:** You may have to test the app on an iOS device to see the change in texture format in the GPU Debugger. On iOS, the automatic format will be **ASTC 4×4**, which is indistinguishable from the png render.

## Key Points

- UVs, also known as texture coordinates, match vertices to the location in a texture.
- During the modeling process, you flatten the model by marking seams. You can then paint on a texture that matches the flattened model map.
- You can load textures using either the `MTKTextureLoader` or the asset catalog.
- A model may be split into groups of vertices known as submeshes. Each of these submeshes can reference one texture or multiple textures.
- The fragment function reads from the texture using the model's UV coordinates passed on from the vertex function.
- The sRGB color space is the default color gamut. Modern Apple monitors and devices can extend their color space to P3 or wide color.
- **Capture GPU workload** is a useful debugging tool. Use it regularly to inspect what's happening on the GPU.
- Mipmaps are resized textures that match the fragment sampling. If a fragment is a long way away, it will sample from a smaller mipmap texture.
- The asset catalog is a great place to store all of your textures. Later, you'll have multiple textures per model, and it's better to keep them all in one place. Customization for different devices is easy using the asset catalog.
- Topics such as color and compression are huge. In the resources folder for this chapter, in `references.markdown`, you'll find some recommended articles to read further.

# Chapter 9: Navigating a 3D Scene

A scene can consist of one or more cameras, lights and models. Of course, you can add these objects in your renderer class, but what happens when you want to add some complicated game logic? Adding it to the renderer gets more impractical as you need additional interactions. Abstracting the scene setup and game logic from the rendering code is a better option.

Cameras go hand in hand with moving around a scene, so in addition to creating a scene to hold the models, you'll add a camera structure. Ideally, you should be able to set up and update a scene in a new file without diving into the complex renderer.

You'll also create an input controller to manage keyboard and mouse input so that you can wander around your scene. The game engines will include features such as input controllers, physics engines and sound.

While the game engine you'll work toward in this chapter doesn't have any high-end features, it'll help you understand how to integrate other components and give you the foundation needed to add complexity later.



# The Starter Project

The starter project for this chapter is the same as the final project for the previous chapter.

## Scenes

A scene holds models, cameras and lighting. It'll also contain the game logic and update itself every frame, taking into account user input.

- Open the starter project. Create a new Swift file called **GameScene.swift** and replace the code with:

```
import MetalKit

struct GameScene {
```

If you created a structure named **Scene** rather than **GameScene**, there would be a conflict with the SwiftUI Scene you use in **NavigationApp.swift**. If you really want to use **Scene**, you can add the explicit namespace to **Scene** in **NavigationApp.swift** using **SwiftUI.Scene**. But it's best to remember that Scenes belong to SwiftUI.

- Add this code to **GameScene**:

```
lazy var house: Model = {
    Model(name: "lowpoly-house.obj")
}()

lazy var ground: Model = {
    var ground = Model(name: "plane.obj")
    ground.tiling = 16
    ground.scale = 40
    return ground
}()

lazy var models: [Model] = [ground, house]
```

The scene holds all of the models you need to render.

- Open **Renderer.swift**, and remove the instantiation of **house** and **ground**.
- Add a new property:

```
lazy var scene = GameScene()
```



You still have some compile errors in `draw(in:)` because you removed house and ground. You'll fix those shortly.

At the moment, you rotate the models just before drawing them, but it's a good idea to separate **update** and **render**. `GameScene` will update the models, and `Renderer` will render them.

► Open `GameScene.swift` and add a new update method to `GameScene`:

```
mutating func update(deltaTime: Float) {
    ground.scale = 40
    ground.rotation.y = sin(deltaTime)
    house.rotation.y = sin(deltaTime)
}
```

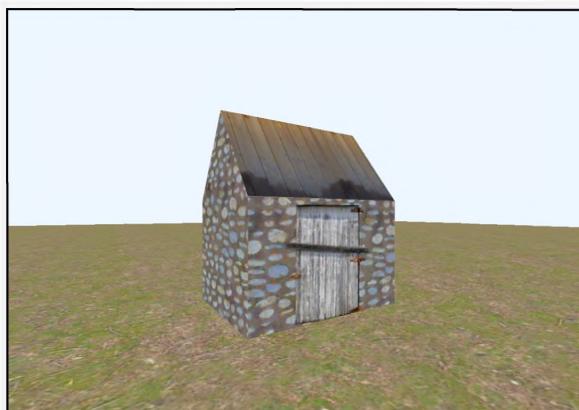
Here, you perform the rotation and scaling, which are currently in `Renderer`.

You'll calculate `deltaTime`, which is the amount of time that has passed since the previous frame soon. You'll pass this from `Renderer` to `GameScene`.

► In `Renderer.swift`, in `draw(in:)`, replace everything between `// update` and `render` to `// end update and render` with:

```
scene.update(deltaTime: timer)
for model in scene.models {
    model.render(
        encoder: renderEncoder,
        uniforms: uniforms,
        params: params)
}
```

► Build and run the app.



*The initial scene*



Here, you reduce the complexity of `draw(in:)`, separate the update from the render and set up the scene to handle its own updates. You can also more easily add and update models in `GameScene`.

## Cameras

Instead of creating view and projection matrices in the renderer, you can abstract the construction and calculation away rendering code to a `Camera` structure. Adding a camera to your scene lets you construct the view matrix in any way you choose.

Currently, you rotate the scene by rotating both house and ground in the y axis. While it looks to the viewer as if a camera is rotating around the scene, in fact, the view matrix doesn't change. Now you'll explore how to move a camera around the scene with keyboard and mouse input.

Setting up a camera is simply a way of calculating a view matrix. Miscalculating the view matrix is a frequent pain point where your carefully rendered objects result in a blank screen. So, it's worth spending time working out common camera setups.

- Create a new Swift file named `Camera.swift`, and replace the existing code with:

```
import CoreGraphics

protocol Camera: Transformable {
    var projectionMatrix: float4x4 { get }
    var viewMatrix: float4x4 { get }
    mutating func update(size: CGSize)
    mutating func update(deltaTime: Float)
}
```

Cameras have a position and rotation, so they should conform to `Transformable`. All cameras have a projection and view matrix as well as methods to perform when the window size changes and when each frame updates.

- Create a custom camera:

```
struct FPCamera: Camera {
    var transform = Transform()
}
```

You created a first-person camera. Eventually, this camera will move forward when you press the `W` key.



- Add this code to `FPCamera`:

```
var aspect: Float = 1.0
var fov = Float(70).degreesToRadians
var near: Float = 0.1
var far: Float = 100
var projectionMatrix: float4x4 {
    float4x4(
        projectionFov: fov,
        near: near,
        far: far,
        aspect: aspect)
}
```

Currently, you set up the projection matrix in `Renderer`'s mtkView(_:drawableSizeWillChange:)`. You'll remove that code in `Renderer` shortly.

- Add a new method to update the camera's aspect ratio:

```
mutating func update(size: CGSize) {
    aspect = Float(size.width / size.height)
}
```

- Add the camera's view matrix calculation:

```
var viewMatrix: float4x4 {
    (float4x4(rotation: rotation) *
     float4x4(translation: position)).inverse
}
```

Each camera calculates its own projection and view matrix. You'll change this view matrix in a moment. But first, you'll run the app to see what this matrix does.

- Add the update method:

```
mutating func update(deltaTime: Float) {
```

This method repositions the camera every frame.

You've now set up all the properties and methods required to conform to `Camera`.

- Open `GameScene.swift`, and add a new camera property:

```
var camera = FPCamera()
```

- Then add an initializer to GameScene:

```
init() {
    camera.position = [0, 1.5, -5]
```

- Create a new method:

```
mutating func update(size: CGSize) {
    camera.update(size: size)
}
```

This code will update the camera when the screen size changes.

- Open **Renderer.swift** and replace the entire contents of `mtkView(_:drawableSizeWillChange:)` with:

```
scene.update(size: size)
```

When the screen size changes, the scene update calls the camera update, which in turn updates the aspect ratio needed for the projection matrix.

- In `draw(in:)`, after `scene.update(deltaTime: timer)`, add:

```
uniforms.viewMatrix = scene.camera.viewMatrix
uniforms.projectionMatrix = scene.camera.projectionMatrix
```

Here, you set up the uniforms the vertex shader requires.

- Next, remove the previous `viewMatrix` assignment at the top of `draw(in:)`:

```
uniforms.viewMatrix =
    float4x4(translation: [0, 1.5, -5]).inverse
```

Now instead of rotating the ground and the house, you can easily rotate the camera.

- Open **GameScene.swift**. In `update(deltaTime:)`, replace:

```
ground.rotation.y = sin(deltaTime)
house.rotation.y = sin(deltaTime)
```

With:

```
camera.rotation.y = sin(deltaTime)
```

- Build and run the app.



*The camera rotating*

Instead of rotating the house and ground, the camera now rotates around the world origin. If you update the view matrix, the vertex shader code updates the final transformation of all the models in the scene.

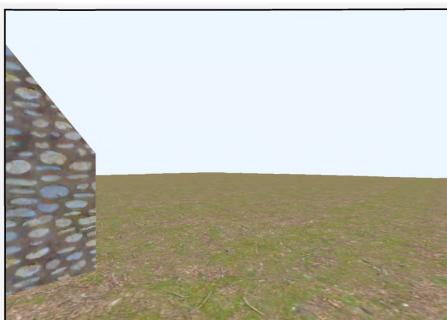
However, you don't want the camera to rotate around the world origin in a first-person camera: You want it to rotate around its own origin.

- Open **Camera.swift**, and change `viewMatrix` in **FPCamera** to:

```
var viewMatrix: float4x4 {
    (float4x4(translation: position) *
     float4x4(rotation: rotation)).inverse
}
```

Here, you reverse the order of matrix multiplication so that the camera rotates around its own origin.

- Build and run the app.



*The camera rotating around its center*

The camera now rotates in place. Next, you'll set up keys to move around the scene.

# Input

There are various forms of input, such as game controllers, keyboards, mice and trackpads. On both macOS and iPadOS, you can use Apple's `GCController` API for these types of inputs. This API helps you set your code up for:

- **Events or Interrupts:** Takes action when the user presses the key. You can set delegate methods or closures of code to run when an event occurs.
- **Polling:** Processes all pressed keys on every frame.

In this app, you'll use polling to move your cameras and players. It's a personal choice, and neither method is better than the other.

The input code you'll build works on macOS and iPadOS if you connect a keyboard and mouse to your iPad. If you want input on your iPhone or iPad without extra controllers, use `GCVirtualController`, which lets you configure on-screen controls that emulate a game controller. You can download Apple's Supporting Game Controllers sample code (<https://apple.co/3qfeL60>) that demonstrates this.

► Create a new Swift file named `InputController.swift`, and replace the code with:

```
import GameController

class InputController {
    static let shared = InputController()
}
```

This code creates a singleton input controller that you can access throughout your app.

► Add a new property to `InputController`:

```
var keysPressed: Set<GCKeyCode> = []
```

In this set, `InputController` keeps track of all keys currently pressed.

To track the keyboard, you need to set up an observer.

► Add this initializer:

```
private init() {
    let center = NotificationCenter.default
    center.addObserver(
        forName: .GCKeboardDidConnect,
        object: nil,
```



```

queue: nil) { notification in
    let keyboard = notification.object as? GKKeyboard
    keyboard?.keyboardInput?.keyChangedHandler
        = { _, _, keyCode, pressed in
            if pressed {
                self.keysPressed.insert(keyCode)
            } else {
                self.keysPressed.remove(keyCode)
            }
        }
    }
}

```

Here, you add an observer to set the keyChangedHandler when the keyboard first connects to the app. When the player presses or lifts a key, the keyChangedHandler code runs and either adds or removes the key from the set.

Now test to see if it works.

- Open **GameScene.swift**, and add this to the end of `update(deltaTime:)`:

```

if InputController.shared.keysPressed.contains(.keyH) {
    print("H key pressed")
}

```

- Build and run the app. Then press different keys.

Press the **H** key in the console and you'll see your print log.

```

loaded texture: grass
loaded texture: barn
H key pressed

```

Notice the annoying warning ‘beep’ from macOS when you press the keys.

- Open **InputController.swift**, and add this to the end of `init()`:

```

#if os(macOS)
    NSEvent.addLocalMonitorForEvents(
        matching: [.keyUp, .keyDown]) { _ in nil }
#endif

```

Here, on macOS only, you interrupt the view’s responder chain by handling any key presses and telling the system that it doesn’t need to take action when a key is pressed. You don’t need to do this for iPadOS, as the iPad doesn’t make the keyboard noise.



**Note:** You could add keys to `keysPressed` in this code instead of using the observer. However, that wouldn't work on iPadOS, and `GCKekeyCode` is easier to read than the raw key values that `NSEvent` gives you.

- Build and rerun the app. Test pressing keys.

Now the noise is gone. If you have a Bluetooth keyboard and an iPad device, connect the keyboard to the iPad and test that it also works on iPadOS.

- Open `GameScene.swift`, and remove the key testing code.

You can now capture any pressed key. Soon, you'll set up standard movement keys to move the camera.

## Delta Time

First, you'll set up the left and right arrows on the keyboard to control the camera's rotation.

When considering movement, think about how much time has passed since the last movement occurred. In an ideal world, at 60 frames per second, a frame should take 0.01667 milliseconds to execute. However, some displays can produce 120 frames per second or even a variable refresh rate.

If you get a choppy frame rate, you can smooth out the movement by calculating delta time, which is the amount of time since the previous execution of the code.

- Open `Renderer.swift`, and replace `var timer: Float = 0` with:

```
var lastTime: Double = CFAbsoluteTimeGetCurrent()
```

`lastTime` holds the time from the previous frame. You initialize it with the current time.

- In `draw(in:)`, remove:

```
timer += 0.005
```



- Replace `scene.update(deltaTime: timer)` with:

```
let currentTime = CFAbsoluteTimeGetCurrent()
let deltaTime = Float(currentTime - lastTime)
lastTime = currentTime
scene.update(deltaTime: deltaTime)
```

Here, you get the current time and calculate the difference from the last time.

## Camera Rotation

- Open `GameScene.swift`. In `update(deltaTime:)`, replace:

```
camera.rotation.y = sin(deltaTime)
```

With:

```
camera.update(deltaTime: deltaTime)
```

- Create a new Swift file named **Movement.swift**, and add:

```
enum Settings {
    static var rotationSpeed: Float { 2.0 }
    static var translationSpeed: Float { 3.0 }
    static var mouseScrollSensitivity: Float { 0.1 }
    static var mousePanSensitivity: Float { 0.008 }
}
```

You can tweak these settings to make your camera and mouse movement smooth. Eventually, you might choose to replace `Settings` with a user interface that sets the values.

- **rotationSpeed**: How many radians the camera should rotate in one second. You'll calculate the amount the camera should rotate in delta time.
- **translationSpeed**: The distance per second that your camera should travel.
- **mouseScrollSensitivity** and **mousePanSensitivity**: Settings to adjust mouse tracking and scrolling.

- Add a new protocol:

```
protocol Movement where Self: Transformable { }
```

Your game might move a player object instead of a camera, so make the movement code as flexible as possible. Now you can give any `Transformable` object `Movement`.

- Create an extension with a default method:

```
extension Movement {
    func updateInput(deltaTime: Float) -> Transform {
        var transform = Transform()
        let rotationAmount = deltaTime * Settings.rotationSpeed
        let input = InputController.shared
        if input.keysPressed.contains(.leftArrow) {
            transform.rotation.y -= rotationAmount
        }
        if input.keysPressed.contains(.rightArrow) {
            transform.rotation.y += rotationAmount
        }
        return transform
    }
}
```

You already told `InputController` to add and remove key presses to a Set called `keysPressed`. Here, you find out if `keysPressed` contains the arrow keys. If it does, you change the transform rotation value.

- Open `Camera.swift` and add the protocol conformance to `FPCamera`:

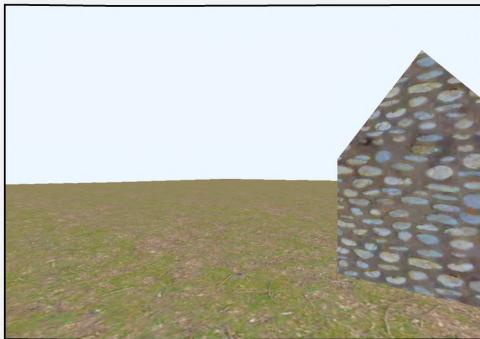
```
extension FPCamera: Movement { }
```

- Still in `Camera.swift`, add this code to `update(deltaTime:)`:

```
let transform = updateInput(deltaTime: deltaTime)
rotation += transform.rotation
```

You update the camera's rotation with the transform calculated in `Movement`.

- Build and run the app. Now, use the arrow keys to rotate the camera.



*Using arrow keys to rotate the camera*

## Camera Movement

You can implement forward and backward movement the same way using standard **WASD** keys:

- **W**: Move forward
- **A**: Strafe left
- **S**: Move backward
- **D**: Strafe right

Here's what to expect when you move the camera through the scene:

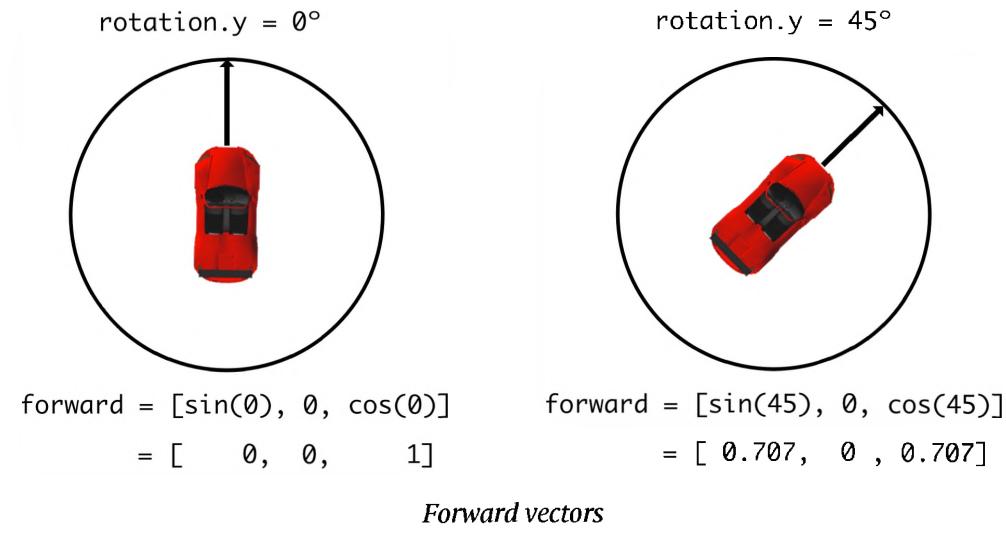
- You'll move along the **x** and **z** axes.
- Your camera will have a direction vector. When you press the **W** key, you'll move along the **z** axis in a positive direction.
- If you press the **W** and **D** keys simultaneously, you'll move diagonally.
- When you press the left and right arrow keys, you'll rotate in that direction, which changes the camera's forward direction vector.

- Open **Movement.swift**, and add a computed property to Movement's extension:

```
var forwardVector: float3 {
    normalize([sin(rotation.y), 0, cos(rotation.y)])
}
```

This is the forward vector based on the current rotation.

The following image shows an example of forward vectors when `rotation.y` is  $0^\circ$  and  $45^\circ$ :



- Add a computed property to handle strafing from side to side:

```
var rightVector: float3 {
    [forwardVector.z, forwardVector.y, -forwardVector.x]
}
```

This vector points  $90^\circ$  to the right of the forward vector.

- Still in **Movement**, at the end of `updateInput(deltaTime:)`, before `return`, add:

```
var direction: float3 = .zero
if input.keysPressed.contains(.keyW) {
    direction.z += 1
}
if input.keysPressed.contains(.keyS) {
    direction.z -= 1
}
if input.keysPressed.contains(.keyA) {
    direction.x -= 1
}
```

```
    }
    if input.keysPressed.contains(.keyD) {
        direction.x += 1
    }
```

This code processes each depressed key and creates a final desired direction vector. For instance, if the game player presses **W** and **A**, she wants to go diagonally forward and left. The final direction vector is **[−1, 0, 1]**.

► After the previous code, add:

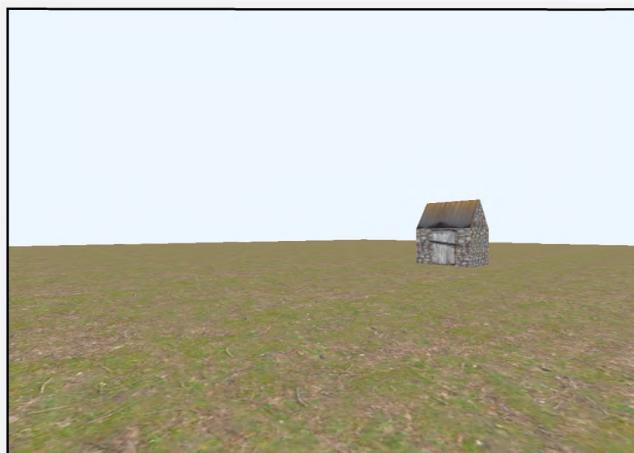
```
let translationAmount = deltaTime * Settings.translationSpeed
if direction != .zero {
    direction = normalize(direction)
    transform.position += (direction.z * forwardVector
        + direction.x * rightVector) * translationAmount
}
```

Here, you calculate the transform's position from its current forward and right vectors and the desired direction.

► Open **Camera.swift**, and add this code to the end of **update(deltaTime:)**:

```
position += transform.position
```

► Build and run the app. Now, you can move about your scene using the keyboard controls.



*Moving around the scene using the keyboard*

## Mouse and Trackpad Input

Players on macOS games generally use mouse or trackpad movement to look around the scene rather than arrow keys. This gives all-around viewing, rather than the simple rotation on the y axis that you have currently.

- Open `InputController.swift`, and add a new structure to `InputController` that you'll use in place of `CGPoint`:

```
struct Point {  
    var x: Float  
    var y: Float  
    static let zero = Point(x: 0, y: 0)  
}
```

Make sure that `Point` goes inside `InputController` to avoid future name conflicts. `Point` is the same as `CGPoint`, except it contains `FLOATS` rather than `CGFLOATS`.

- Add these properties to `InputController` to record mouse movement:

```
var leftMouseDown = false  
var mouseDelta = Point.zero  
var mouseScroll = Point.zero
```

- `leftMouseDown`: Tracks when the player does a left-click.
- `mouseDelta`: The movement since the last tracked movement.
- `mouseScroll`: Keeps track of how much the player scrolls with the mouse wheel.

- At the end of `init()`, add:

```
center.addObserver(  
    forName: .GCMouseDidConnect,  
    object: nil,  
    queue: nil) { notification in  
    let mouse = notification.object as? GCMouse  
}
```

This code to connect the mouse is similar to how you handled the keyboard connection.

- Inside the closure, after setting `mouse`, add:

```
// 1  
mouse?.mouseInput?.leftButton.pressedChangedHandler = { _, _,  
    pressed in
```

```
    self.leftMouseDown = pressed
}
// 2
mouse?.mouseInput?.mouseMovedHandler = { _, deltaX, deltaY in
    self.mouseDelta = Point(x: deltaX, y: deltaY)
}
// 3
mouse?.mouseInput?.scroll.valueChangedHandler = { _, xValue,
    yValue in
    self.mouseScroll.x = xValue
    self.mouseScroll.y = yValue
}
```

Here, you:

1. Record when the user holds down the left mouse button.
2. Track mouse movement.
3. Record scroll wheel movement. `xValue` and `yValue` are normalized values between -1 and 1. If you use a game controller instead of a mouse, the first parameter is `dpad`, which tells you which directional pad element changed.

Now you're ready to use these tracked mouse input values.

**Note:** For iOS, you should use touch values rather than mouse tracking. The challenge project sample sets up `MetalView` with gestures that update `InputController` values.

## Arcball Camera

In many apps, the camera rotates about a particular point. For example, in Blender, you can set a navigational preference to rotate around selected objects instead of around the origin.

- Open `Camera.swift`. Copy and paste the `FPCamera` structure without the extension.
- Rename the copied structure:

```
struct ArcballCamera: Camera {
```



- Since you're not implementing Movement, remove all the code from `update(deltaTime:)`.

Your code will now compile.

- Open **GameScene.swift** and change the camera initialization to:

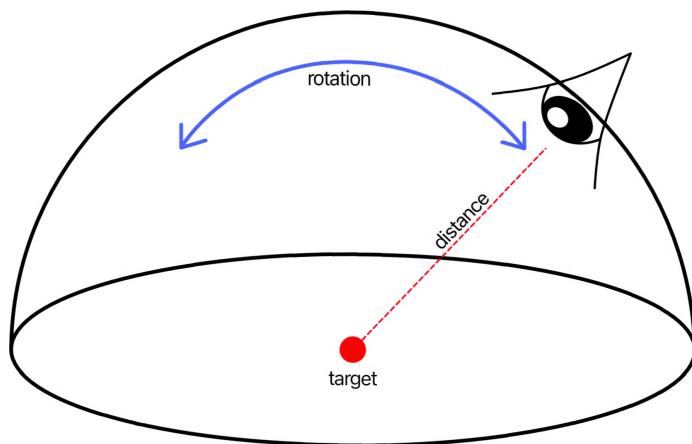
```
var camera = ArcballCamera()
```

Now, you have a camera with no movement.

## Orbiting a Point

The camera needs a track to rotate about a point:

- **Target:** The point the camera will orbit.
- **Distance:** The distance between the camera and the target. The player controls this with the mouse wheel.
- **Rotation:** The camera's rotation about the point. The player controls this by left-clicking the mouse and dragging.



*Orbiting a point*

From these three properties, you can determine the camera's world position and rotate it to always point at the target position. To do this:

1. First, rotate the distance vector around the y axis.
2. Then, rotate the distance vector around the x axis.
3. Add the target position to the rotated vector to get the new world position of the camera.
4. Rotate the camera to look at the target position.

► Open **Camera.swift**, and add the necessary properties to **ArcballCamera** to keep track of the camera's orbit:

```
let minDistance: Float = 0.0
let maxDistance: Float = 20
var target: float3 = [0, 0, 0]
var distance: Float = 2.5
```

You'll constrain `distance` with `minDistance` and `maxDistance`.

► In `update(deltaTime:)`, add:

```
let input = InputController.shared
let scrollSensitivity = Settings.mouseScrollSensitivity
distance -= (input.mouseScroll.x + input.mouseScroll.y)
    * scrollSensitivity
distance = min(maxDistance, distance)
distance = max(minDistance, distance)
input.mouseScroll = .zero
```

Here, you change `distance` depending on the mouse scroll values.

► Continue with this code:

```
if input.leftMouseDown {
    let sensitivity = Settings.mousePanSensitivity
    rotation.x += input.mouseDelta.y * sensitivity
    rotation.y += input.mouseDelta.x * sensitivity
    rotation.x = max(-.pi / 2, min(rotation.x, .pi / 2))
    input.mouseDelta = .zero
}
```

If the player drags with the left mouse button, update the camera's rotation values.

► Add this after the previous code:

```
let rotateMatrix = float4x4(  
    rotationYXZ: [-rotation.x, rotation.y, 0])  
let distanceVector = float4(0, 0, -distance, 0)  
let rotatedVector = rotateMatrix * distanceVector  
position = target + rotatedVector.xyz
```

Here, you complete the calculations to rotate the distance vector and add the target position to the new vector. In `MathLibrary.swift`, `float4x4(rotationYXZ:)` creates a matrix using rotations in Y / X / Z order.

## The lookAt Matrix

A `lookAt` matrix rotates the camera so it always points at a target. In `MathLibrary.swift`, you'll find a `float4x4` initialization `init(eye:center:up:)`. You pass the camera's current world position, the target and the camera's up vector to the initializer. In this app, the camera's up vector is always `[0, 1, 0]`.

► In `ArcballCamera`, replace `viewMatrix` with:

```
var viewMatrix: float4x4 {  
    let matrix: float4x4  
    if target == position {  
        matrix = (float4x4(translation: target) *  
            float4x4(rotationYXZ: rotation)).inverse  
    } else {  
        matrix = float4x4(eye: position, center: target, up: [0, 1,  
            0])  
    }  
    return matrix  
}
```

If the position is the same as the target, you simply rotate the camera to look around the scene at the target position. Otherwise, you rotate the camera with the `lookAt` matrix.

► Open `GameScene.swift`, and add this to the end of `init()`:

```
camera.distance = length(camera.position)  
camera.target = [0, 1.2, 0]
```

With an arcball camera, you set the target and distance as well as the position.

- Build and run the app. Zoom in to get a view of the inside of the barn.

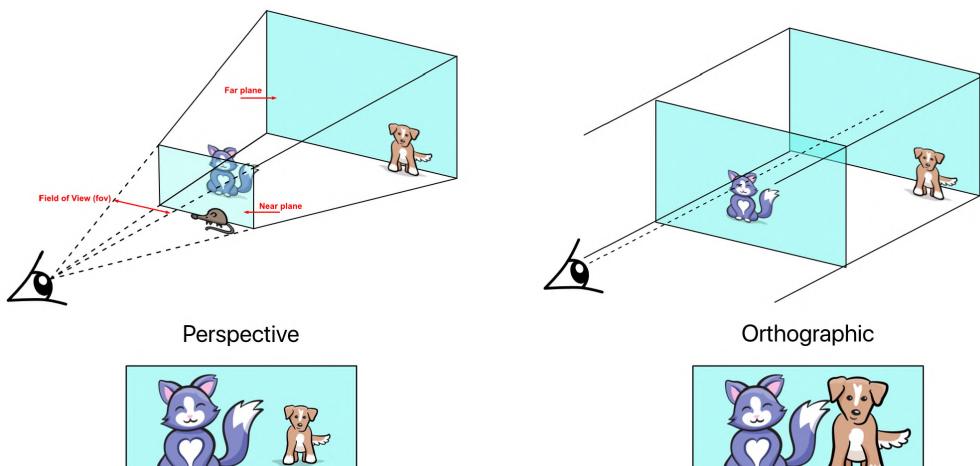


*Inside the barn*

Click and drag to orbit the barn. In **Movement.swift**, change **Settings** to suit your tracking preferences.

## Orthographic Projection

So far, you've created cameras with perspective so that objects further back in your 3D scene appear smaller than the ones closer to the camera. Orthographic projection flattens three dimensions to two dimensions without any perspective distortion.



*Orthographic projection*

Sometimes it's a little difficult to see what's happening in a large scene. To help with that, you'll build a top-down camera that shows the whole scene without any perspective distortion.

► Open **Camera.swift**, and add a new camera:

```
struct OrthographicCamera: Camera, Movement {
    var transform = Transform()
    var aspect: CGFloat = 1
    var viewSize: CGFloat = 10
    var near: Float = 0.1
    var far: Float = 100

    var viewMatrix: float4x4 {
        (float4x4(translation: position) *
        float4x4(rotation: rotation)).inverse
    }
}
```

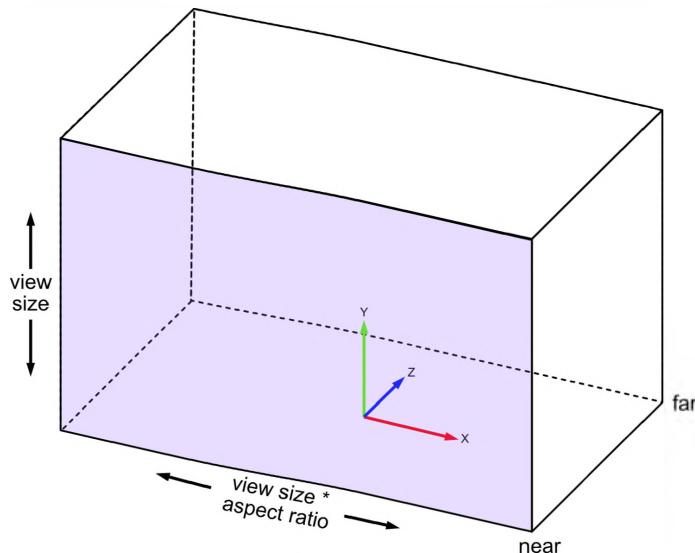
`aspect` is the ratio of the window's width to height. `viewSize` is the unit size of the scene. You'll calculate the projection frustum in the shape of a box.

► Add the projection matrix code:

```
var projectionMatrix: float4x4 {
    let rect = CGRect(
        x: -viewSize * aspect * 0.5,
        y: viewSize * 0.5,
        width: viewSize * aspect,
        height: viewSize)
    return float4x4(orthographic: rect, near: near, far: far)
}
```



Here, you calculate a rectangle for the front of the frustum using the view size and aspect ratio. Then you call the orthographic initializer defined in **MathLibrary.swift** with the frustum's near and far values.



*The orthographic projection frustum*

- Add the method to update the camera when the view size changes:

```
mutating func update(size: CGSize) {
    aspect = size.width / size.height
}
```

This code sets the aspect ratio for the orthographic projection matrix.

- Add the frame update method:

```
mutating func update(deltaTime: Float) {
    let transform = updateInput(deltaTime: deltaTime)
    position += transform.position
    let input = InputController.shared
    let zoom = input.mouseScroll.x + input.mouseScroll.y
    viewSize -= CGFloat(zoom)
    input.mouseScroll = .zero
}
```

Here, you use the previous Movement code to move around the scene using the **WASD** keys. You don't need rotation, as you're going to position the camera to be top-down. You use the mouse scroll to change the view size, which allows you to zoom in and out of the scene.

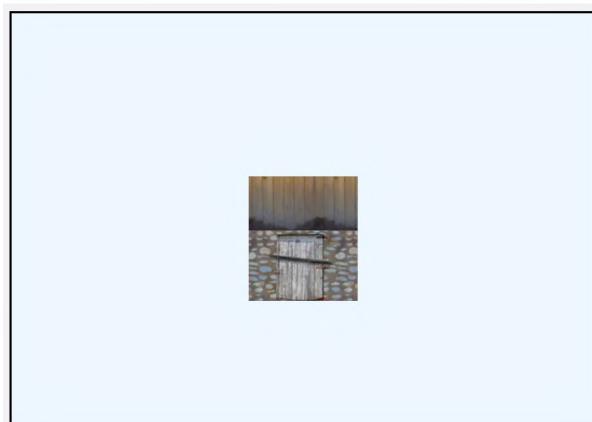
- Open **GameScene.swift** and change camera to:

```
var camera = OrthographicCamera()
```

- Replace the contents of `init()` with:

```
camera.position = [3, 2, 0]
camera.rotation.y = -.pi / 2
```

- Build and run the app.



*Orthographic viewing from the front*

You'll see a scene with no perspective. You might be surprised by this result. You can't see the ground because there's no field of view, which means you effectively "see" the ground plane side-on.

Replace the previous code in `init()` with:

```
camera.position = [0, 2, 0]
camera.rotation.x = .pi / 2
```

This code places the camera in a top-down rotation.

- Build and run the app. You can still move the WASD movement keys to move about the scene and use the mouse scroll to zoom out.



*Orthographic viewing from the top*

You'll often use an orthographic camera when creating 2D games that look down on an entire board. Later, you'll also use an orthographic camera when implementing shadows from directional lights.

## Challenge

For your challenge, combine FPCamera and ArcballCamera into one PlayerCamera. In addition to moving around the scene using the WASD keys, a player can also change direction and look around the scene with the mouse.

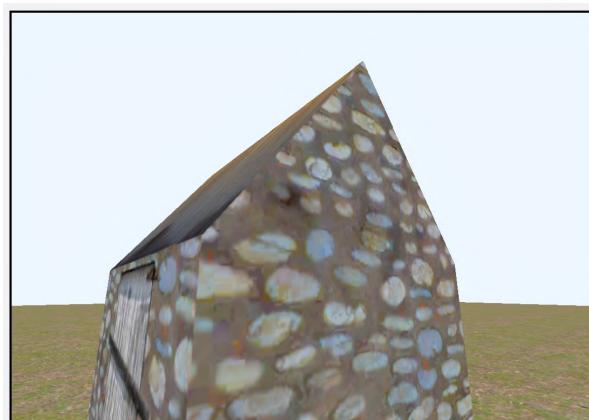
To achieve this:

- Copy FPCamera to PlayerCamera. This sets position and rotation.
- Copy the left mouse down code from ArcballCamera's `update(deltaTime:)` to the end of PlayerCamera's `update(deltaTime:)`. This sets rotation when the mouse is used. PlayerCamera won't use the scroll wheel for zooming.
- The view matrix should use rotation without the z axis because you always travel on the xy plane. So, replace PlayerCamera's `viewMatrix` with:

```
var viewMatrix: float4x4 {
    let rotateMatrix = float4x4(
        rotationYXZ: [-rotation.x, rotation.y, 0])
    return (float4x4(translation: position) *
    rotateMatrix).inverse
}
```

- Change the camera in GameScene and set its initial position.

When you finish, wander around your scene with your left hand on the keyboard to control forward motion and your right hand on the mouse to control direction. The left and right arrow keys will work for rotation, too.



*Moving around the scene*

## Key Points

- Scenes abstract game code and scene setup away from the rendering code.
- Camera structures let you calculate the view and projection matrices separately from rendering the models.
- On macOS and iPadOS, use Apple's `GCController` API to process input from game controllers, keyboards and mice.
- On iOS, `GCVirtualController` gives you onscreen D-pad controls.
- For a first-person camera, calculate position and rotation from the player's perspective.
- An arball camera orbits a target point.
- An orthographic camera renders without perspective so that all vertices rendered to the 2D screen appear at the same distance from the camera.

# Chapter 10: Lighting Fundamentals

Light and shade are important requirements for making your scenes pop. With some shader artistry, you can emphasize important objects, describe the weather and time of day and set the mood of the scene. Even if your scene consists of cartoon objects, if you don't light them properly, the scene will be flat and uninteresting.

One of the simplest methods of lighting is the **Phong reflection model**. It's named after Bui Tong Phong who published a paper in 1975 extending older lighting models. The idea is not to attempt duplication of light and reflection physics but to generate pictures that look realistic.

This model has been popular for over 40 years and is a great place to start learning how to fake lighting using a few lines of code. All computer images are fake, but there are more modern real-time rendering methods that model the physics of light.

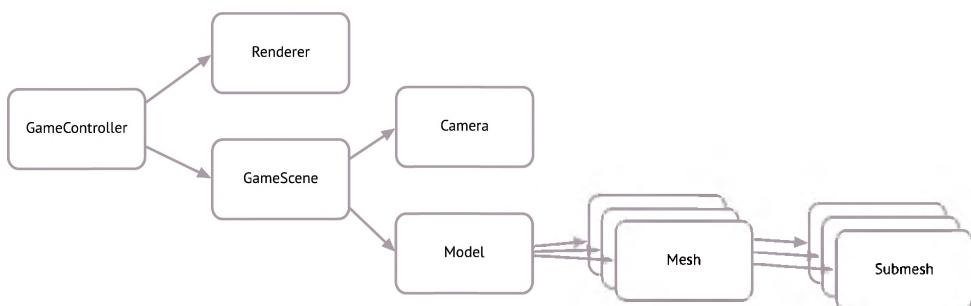
In Chapter 11, “Maps & Materials”, you’ll take a look at Physically Based Rendering (PBR), the lighting technique that your renderer will eventually use. PBR is a more realistic lighting model, but Phong is easy to understand and get started with.



# The Starter Project

- Open the starter project for this chapter.

The starter project's files are now in sensible groups. In the **Game** group, the project contains a new game controller class which further separates scene updates and rendering. Renderer is now independent from GameScene. GameController initializes and owns both Renderer and GameScene. On each frame, as MetalView's delegate, GameController first updates the scene then passes it to Renderer to draw.



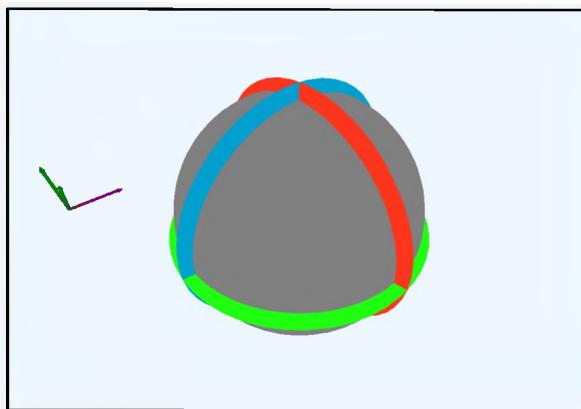
*Object ownership*

In **GameScene.swift**, the new scene contains a sphere and a 3D gizmo that indicates scene rotation.

**DebugLights.swift** in the **Utility** group contains some code that you'll use later for debugging where lights are located. Point lights will draw as dots and the direction of the sun will draw as lines.

In the **Geometry** group, the default vertex descriptor in **VertexDescriptor.swift** now includes a color buffer. The sphere model has a texture to show colors, but the 3D gizmo uses vertex colors. In the **Shaders** group, the vertex shader forwards this color to the fragment shader, and the fragment shader uses the vertex color if there is no color texture.

- Familiarize yourself with the code and build and run the project.



*The starter app*

To rotate around the sphere and fully appreciate your lighting, the camera is an `ArcballCamera` type. Press 1 to set the camera to a front view, and 2 to reset the camera to the default view. `GameScene` contains the key pressing code for this.

You can see that the sphere colors are very flat. In this chapter, you'll add shading and specular highlights.

## Representing Color

In this book, you'll learn the necessary basics to get you rendering light, color and simple shading. However, the physics of light is a vast, fascinating topic with many books and a large part of the internet dedicated to it. You can find further reading in **references.markdown** in the resources directory for this chapter.

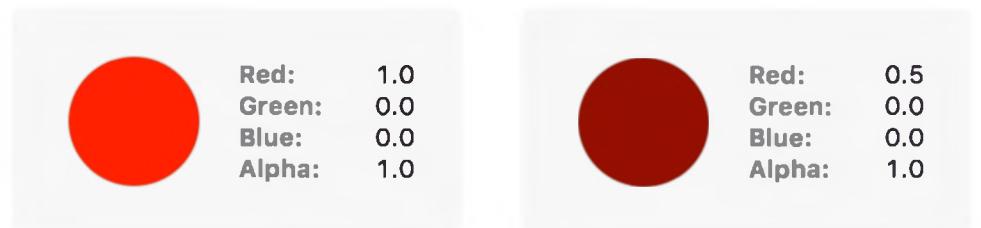
In the real world, the reflection of different wavelengths of light is what gives an object its color. A surface that absorbs all light is black. Inside the computer world, pixels display color. The more pixels, the better the resolution and this makes the resulting image clearer. Each pixel is made up of subpixels. These are a predetermined single color, either red, green or blue. By turning on and off these subpixels, depending on the color depth, the screen can display most of the colors visible to the human eye.

In Swift, you can represent a color using the RGB values for that pixel. For example, `float3(1, 0, 0)` is a red pixel, `float3(0, 0, 0)` is black and `float3(1, 1, 1)` is white.

From a shading point of view, you can combine a red surface with a gray light by multiplying the two values together:

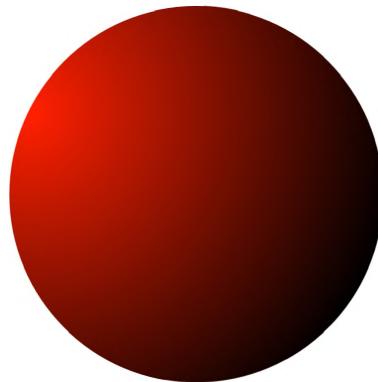
```
let result = float3(1.0, 0.0, 0.0) * float3(0.5, 0.5, 0.5)
```

The result is `(0.5, 0, 0)`, which is a darker shade of red.



### *Color shading*

For simple Phong lighting, you can use the slope of the surface. The more the surface slopes away from a light source, the darker the surface becomes.

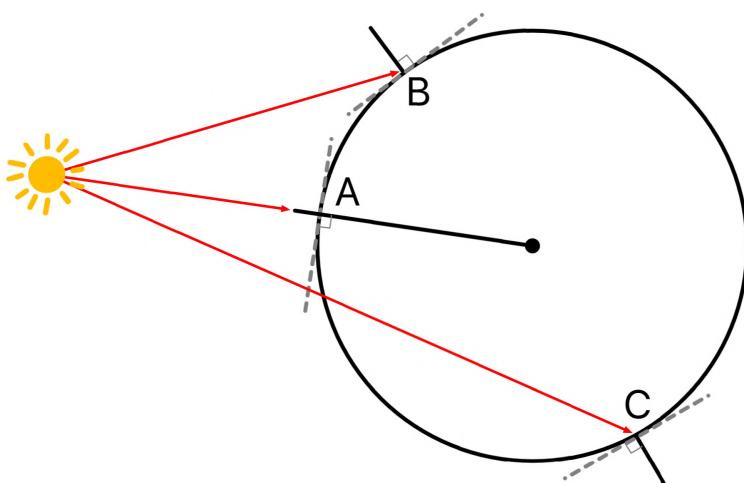


*A 3D shaded sphere*

## Normals

The slope of a surface can determine how much a surface reflects light.

In the following diagram, **point A** is facing straight toward the sun and will receive the most amount of light; **point B** is facing slightly away but will still receive some light; **point C** is facing entirely away from the sun and shouldn't receive any of the light.



*Surface normals on a sphere*

**Note:** In the real world, light bounces from surface to surface; if there's any light in the room, there will be some reflection from objects that gently lights the back surfaces of all the other objects. This is **global illumination**. The Phong lighting model lights each object individually and is called **local illumination**.

The dotted lines in the diagram are **tangent** to the surface. A tangent line is a straight line that best describes the slope of the curve at a point.

The lines coming out of the circle are at right angles to the tangent lines. These are called **surface normals**, and you first encountered these in Chapter 7, “The Fragment Function”.

## Light Types

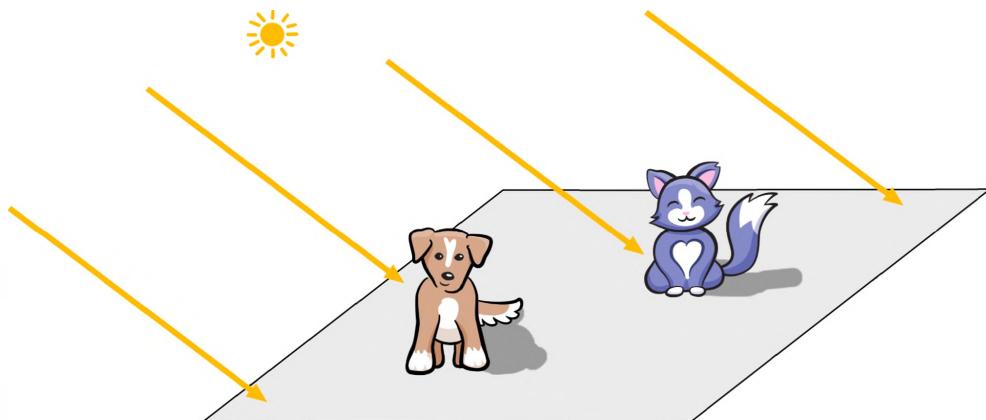
There are several standard light options in computer graphics, each of which has their origin in the real world.

- **Directional Light:** Sends light rays in a single direction. The sun is a directional light.
- **Point Light:** Sends light rays in all directions like a light bulb.
- **Spotlight:** Sends light rays in limited directions defined by a cone. A flashlight or a desk lamp would be a spotlight.

## Directional Light

A scene can have many lights. In fact, in studio photography, it would be highly unusual to have just a single light. By putting lights into a scene, you control where shadows fall and the level of darkness. You'll add several lights to your scene through the chapter.

The first light you'll create is the sun. The sun is a point light that puts out light in all directions, but for computer modeling, you can consider it a **directional** light. It's a powerful light source a long way away. By the time the light rays reach the earth, the rays appear to be parallel. Check this outside on a sunny day — everything you can see has its shadow going in the same direction.



*The direction of sunlight*

To define the light types, you'll create a `Light` structure that both the GPU and the CPU can read, and a `SceneLighting` structure that will describe the lighting for `GameScene`.

- In the `Shaders` group, open `Common.h`, and before `#endif`, create an enumeration of the light types you'll be using:

```
typedef enum {
    unused = 0,
    Sun = 1,
    Spot = 2,
    Point = 3,
    Ambient = 4
} LightType;
```

- Under this, add the structure that defines a light:

```
typedef struct {
    LightType type;
    vector_float3 position;
    vector_float3 color;
    vector_float3 specularColor;
    float radius;
    vector_float3 attenuation;
    float coneAngle;
    vector_float3 coneDirection;
    float coneAttenuation;
} Light;
```

This structure holds the position and color of the light. You'll learn about the other properties as you go through the chapter.

- Create a new Swift file in the `Game` group, and name it `SceneLighting.swift`. Then, add this:

```
struct SceneLighting {
    static func buildDefaultLight() -> Light {
        var light = Light()
        light.position = [0, 0, 0]
        light.color = [1, 1, 1]
        light.specularColor = [0.6, 0.6, 0.6]
        light.attenuation = [1, 0, 0]
        light.type = Sun
        return light
    }
}
```

This file will hold the lighting for `GameScene`. You'll have several lights, and `buildDefaultLight()` will create a basic light.

- Create a property in SceneLighting for a sun directional light:

```
let sunlight: Light = {  
    var light = Self.buildDefaultLight()  
    light.position = [1, 2, -2]  
    return light  
}()
```

position is in world space. This will place a light to the right of the scene, and forward of the sphere. The sphere is placed at the world's origin.

- Create an array to hold the various lights you'll be creating shortly:

```
var lights: [Light] = []
```

- Add the initializer:

```
init() {  
    lights.append(sunlight)  
}
```

You'll add all your lights for the scene in the initializer.

- Open **GameScene.swift**, and add the lighting property to GameScene:

```
let lighting = SceneLighting()
```

You'll do all the light shading in the fragment function so you'll need to pass the array of lights to that function. Metal Shading Language doesn't have a dynamic array feature, and there is no way to find out the number of items in an array. You'll pass this value to the fragment shader in Params.

- Open **Common.h**, and add these properties to Params:

```
uint lightCount;  
vector_float3 cameraPosition;
```

You'll need the camera position property later.

While you're in **Common.h**, add a new index to BufferIndices:

```
LightBuffer = 13
```



You'll use this to send lighting details to the fragment function.

- Open `Renderer.swift`, and add this to `updateUniforms(scene:)`:

```
params.lightCount = UInt32(scene.lighting.lights.count)
```

You'll be able to access this value in the fragment shader function.

- In `draw(scene:in:)`, just before `for model in scene.models`, add this:

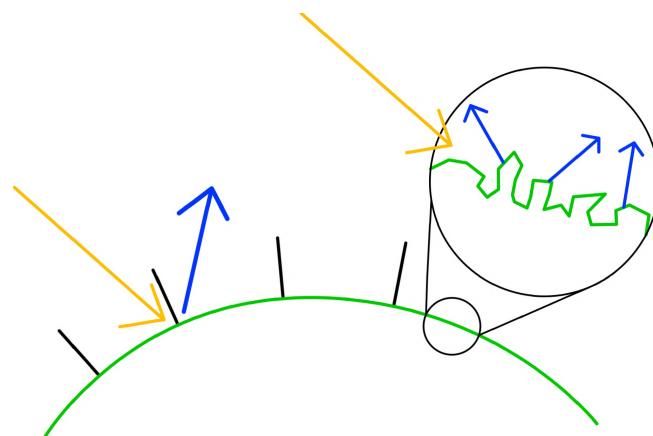
```
var lights = scene.lighting.lights
renderEncoder.setFragmentBytes(
    &lights,
    length: MemoryLayout<Light>.stride * lights.count,
    index: LightBuffer.index)
```

Here, you send the array of lights to the fragment function in buffer index 13.

You've now set up a sun light on the Swift side. You'll do all the actual light calculations in the fragment function, and you'll find out more about light properties.

## The Phong Reflection Model

In the Phong reflection model, there are three types of light reflection. You'll calculate each of these, and then add them up to produce a final color.



*Diffuse shading and micro-facets*

- **Diffuse:** In theory, light coming at a surface bounces off at an angle reflected about the surface normal at that point. However, surfaces are microscopically rough, so light bounces off in all directions as the picture above indicates. This produces a diffuse color where the light intensity is proportional to the angle between the incoming light and the surface normal. In computer graphics, this model is called **Lambertian reflectance** named after Johann Heinrich Lambert who died in 1777. In the real world, this diffuse reflection is generally true of dull, rough surfaces, but the surface with the most Lambertian property is human-made: Spectralon (<https://en.wikipedia.org/wiki/Spectralon>), which is used for optical components.
- **Specular:** The smoother the surface, the shinier it is, and the light bounces off the surface in fewer directions. A mirror completely reflects off the surface normal without deflection. Shiny objects produce a visible specular highlight, and rendering specular lighting can give your viewers hints about what sort of surface an object is — whether a car is an old wreck or fresh off the sales lot.
- **Ambient:** In the real-world, light bounces around all over the place, so a shadowed object is rarely entirely black. This is the ambient reflection.

A surface color is made up of an emissive surface color plus contributions from ambient, diffuse and specular. For diffuse and specular, to find out how much light the surface should receive at a particular point, all you have to do is find out the angle between the incoming light direction and the surface normal.

## The Dot Product

Fortunately, there's a straightforward mathematical operation to discover the angle between two vectors called the dot product.

$$A \cdot B = a_1 b_1 + a_2 b_2 + a_3 b_3$$

And:

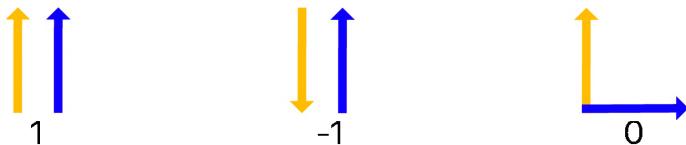
$$A \cdot B = \|A\| \|B\| \cos(\Theta)$$

Where  $\|A\|$  means the length (or magnitude) of vector A.

Even more fortunately, both SIMD and Metal Shading Language have a function `dot()` to get the dot product, so you don't have to remember the formulas.

As well as finding out the angle between two vectors, you can use the dot product for checking whether two vectors are pointing in the same direction.

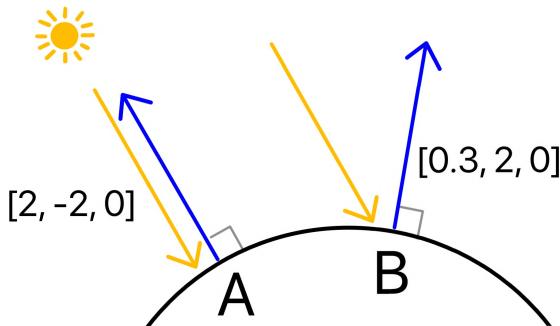
Resize the two vectors into unit vectors — that's vectors with a length of 1. You can do this using the `normalize()` function. If the unit vectors are parallel with the same direction, the dot product result will be 1. If they are parallel but opposite directions, the result will be -1. If they are at right angles (orthogonal), the result will be 0.



*The dot product*

Looking at the previous diagram, if the yellow (sun) vector is pointing straight down, and the blue (normal) vector is pointing straight up, the dot product will be -1. This value is the cosine angle between the two vectors. The great thing about cosines is that they are always values between -1 and 1 so you can use this range to determine how bright the light should be at a certain point.

Take the following example:

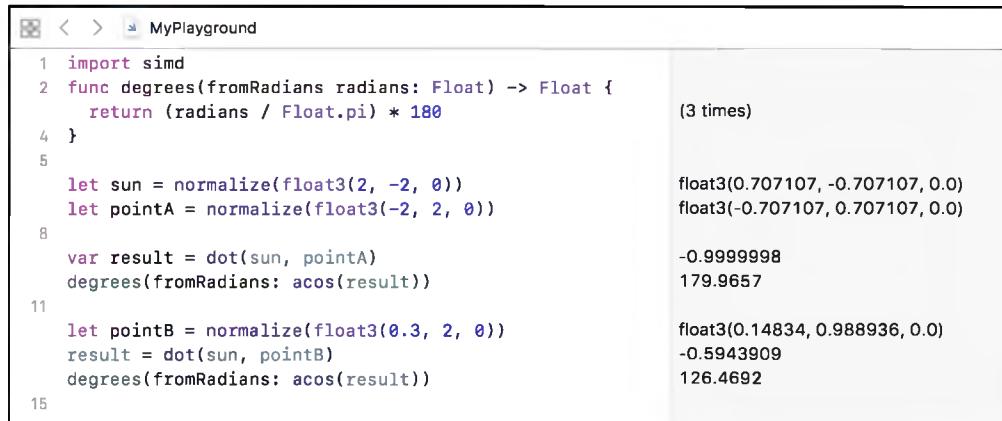


*The dot product of sunlight and normal vectors*

The sun is pouring down from the sky with a direction vector of  $[2, -2, 0]$ . Vector A is a normal vector of  $[2, -2, 0]$ . The two vectors are pointing in opposite directions, so when you turn the vectors into unit vectors (normalize them), the dot product of them will be -1.

Vector B is a normal vector of  $[0.3, 2, 0]$ . Sunlight is a directional light, so uses the same direction vector. Sunlight and B when normalized have a dot product of -0.59.

This playground code demonstrates the calculations.



The screenshot shows a Xcode playground window titled "MyPlayground". It contains the following Swift code:

```
1 import simd
2 func degrees(fromRadians radians: Float) -> Float {
3     return (radians / Float.pi) * 180
4 }
5
6 let sun = normalize(float3(2, -2, 0))
7 let pointA = normalize(float3(-2, 2, 0))
8
9 var result = dot(sun, pointA)
10 degrees(fromRadians: acos(result))
11
12 let pointB = normalize(float3(0.3, 2, 0))
13 result = dot(sun, pointB)
14 degrees(fromRadians: acos(result))
15
```

Output values are shown on the right side of the code:

- Line 6: float3(0.707107, -0.707107, 0.0)
- Line 7: float3(-0.707107, 0.707107, 0.0)
- Line 9: -0.9999998
- Line 10: 179.9657
- Line 12: float3(0.14834, 0.988936, 0.0)
- Line 13: -0.5943909
- Line 14: 126.4692

*Dot product playground code*

**Note:** The result after line 8 shows that you should always be careful when using floating points, as results are never exact. Never use an expression such as if  $(x == 1.0)$  - always check  $\leq$  or  $\geq$ .

In the fragment shader, you'll be able to take these values and multiply the fragment color by the dot product to get the brightness of the fragment.

## Diffuse Reflection

Shading from the sun does not depend on where the camera is. When you rotate the scene, you're rotating the world, including the sun. The sun's position will be in world space, and you'll put the model's normals into the same world space to be able to calculate the dot product against the sunlight direction. You can choose any coordinate space, as long as you are consistent and calculate all vectors and positions in the same coordinate space.

To be able to assess the slope of the surface in the fragment function, you'll reposition the normals in the vertex function in much the same way as you repositioned the vertex position earlier. You'll add the normals to the vertex descriptor so that the vertex function can process them.



- Open **Shaders.metal**, and add these properties to **VertexOut**:

```
float3 worldPosition;  
float3 worldNormal;
```

These will hold the vertex position and vertex normal in world space.

Calculating the new position of normals is a bit different from the vertex position calculation. **MathLibrary.swift** contains a matrix method to create a normal matrix from another matrix. This normal matrix is a  $3 \times 3$  matrix, because firstly, you'll do lighting in world space which doesn't need projection, and secondly, translating an object does not affect the slope of the normals. Therefore, you don't need the fourth W dimension. However, if you scale an object in one direction (non-linearly), then the normals of the object are no longer orthogonal and this approach won't work. As long as you decide that your engine does not allow non-linear scaling, then you can use the upper-left  $3 \times 3$  portion of the model matrix, and that's what you'll do here.

- Open **Common.h** and add this matrix property to **Uniforms**:

```
matrix_float3x3 normalMatrix;
```

This will hold the normal matrix in world space.

- In the **Geometry** group, open **Model.swift**, and in `render(encoder:uniforms:params:)`, add this after setting `uniforms.modelMatrix`:

```
uniforms.normalMatrix = uniforms.modelMatrix.upperLeft
```

This creates the normal matrix from the model matrix.

- Open **Shaders.metal**, and in **vertex\_main**, when defining **out**, populate the **VertexOut** properties:

```
.worldPosition = (uniforms.modelMatrix * in.position).xyz,  
.worldNormal = uniforms.normalMatrix * in.normal
```

Here, you convert the vertex position and normal to world space.

Earlier in the chapter, you sent Renderer's **lights** array to the fragment function in the LightBuffer index, but you haven't yet changed the fragment function to receive the array.

- Add this to `fragment_main`'s parameter list:

```
constant Light *lights [[buffer(LightBuffer)]],
```

## Creating Shared Functions in C++

Often you'll want to access C++ functions from multiple files. Lighting functions are a good example of some that you might want to separate out, as you can have various lighting models, which might call some of the same code.

To call a function from multiple `.metal` files:

1. Set up a header file with the name of the functions that you're going to create.
2. Create a new `.metal` file and import the header, and also the bridging header file `Common.h` if you're going to use a structure from that file.
3. Create the lighting functions in this new file.
4. In your existing `.metal` file, import the new header file and use the lighting functions.

In the **Shaders** group, create a new **Header File** called `Lighting.h`. Don't add it to any target.

- Add this before `#endif /* Lighting_h */`:

```
#import "Common.h"

float3 phongLighting(
    float3 normal,
    float3 position,
    constant Params &params,
    constant Light *lights,
    float3 baseColor);
```

Here, you define a C++ function that will return a `float3`.

In the **Shaders** group, create a new **Metal File** called `Lighting.metal`. Add it to both iOS and macOS targets.

- Add this new function header:

```
#import "Lighting.h"

float3 phongLighting(
```

```
float3 normal,
float3 position,
constant Params &params,
constant Light *lights,
float3 baseColor) {
    return float3(0);
}
```

You create a new function that returns a zero `float3` value. You'll build up code in `phongLighting` to calculate this final lighting value.

- Open `Shaders.metal`, and replace `#import "Common.h"` with:

```
#import "Lighting.h"
```

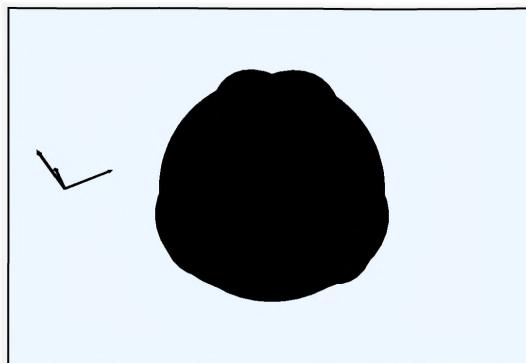
Now you'll be able to use `phongLighting` within this file.

- In `fragment_main`, replace `return float4(baseColor, 1);` with this:

```
float3 normalDirection = normalize(in.worldNormal);
float3 color = phongLighting(
    normalDirection,
    in.worldPosition,
    params,
    lights,
    baseColor
);
return float4(color, 1);
```

Here, you make the world normal a unit vector, and call the new lighting function with the necessary parameters.

If you build and run the app now, your models will render in black, as that's the color that you're currently returning from `phongLighting`.



*No lighting*

- Open **Lighting.metal**, and replace `return float3(0);` with:

```
float3 diffuseColor = 0;
float3 ambientColor = 0;
float3 specularColor = 0;
for (uint i = 0; i < params.lightCount; i++) {
    Light light = lights[i];
    switch (light.type) {
        case Sun: {
            break;
        }
        case Point: {
            break;
        }
        case Spot: {
            break;
        }
        case Ambient: {
            break;
        }
        case unused: {
            break;
        }
    }
}
return diffuseColor + specularColor + ambientColor;
```

This sets up the outline for all the lighting calculations you'll do. You'll accumulate the final fragment color, made up of diffuse, specular and ambient contributions.

- Above `break` in `case Sun`, add this:

```
// 1
float3 lightDirection = normalize(-light.position);
// 2
float diffuseIntensity =
    saturate(-dot(lightDirection, normal));
// 3
diffuseColor += light.color * baseColor * diffuseIntensity;
```

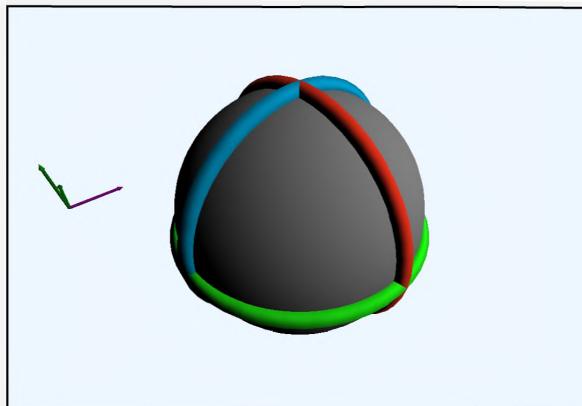
Going through this code:

1. You make the light's direction a unit vector.
2. You calculate the dot product of the two vectors. When the fragment fully points toward the light, the dot product will be -1. It's easier for further calculation to make this value positive, so you negate the dot product. `saturate` makes sure the value is between 0 and 1 by clamping the negative numbers. This gives you the slope of the surface, and therefore the intensity of the diffuse factor.



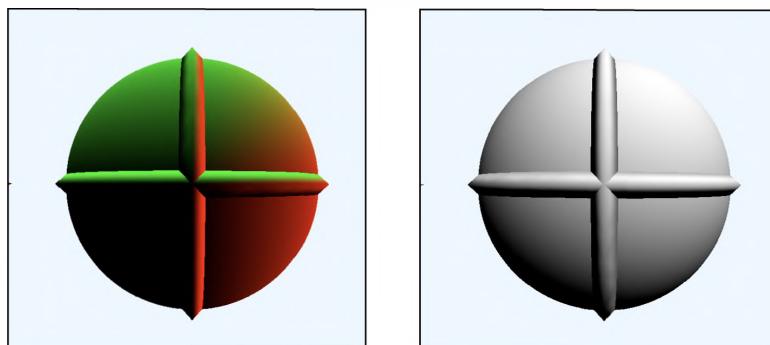
- Multiply the base color by the diffuse intensity to get the diffuse shading. If you have several sun lights, `diffuseColor` will accumulate the diffuse shading.

► Build and run the app.



*Diffuse shading*

You can sanity-check your results by returning your intermediate calculations from `phongLighting`. The following image shows `normal` and `diffuseIntensity` from the front view.



*Visualizing the normal and diffuse intensity*

**Note:** To get the front view in your app, press “1” above the alpha keys while running it. “2” will reset to the default view.

**DebugLights.swift** and **DebugLights.metal** in the **Utility** group, have some debugging methods so that you can visualize where your lights are.

- Open **DebugLights.swift**, and remove /\* and \*/ at the top and bottom of the file.

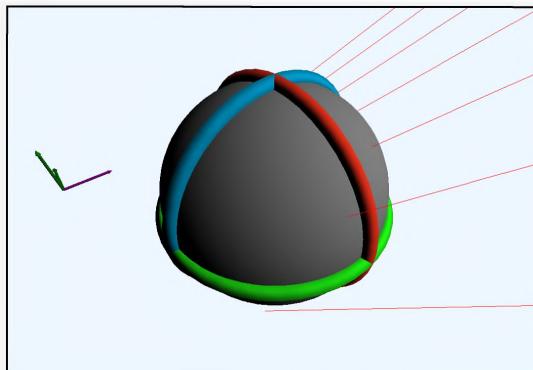
Before you added code in this chapter, this file would not compile, but does now.

- Open **Renderer.swift**, and toward the end of `draw(scene:in:)`, before `renderEncoder.endEncoding()`, add this:

```
DebugLights.draw(  
    lights: scene.lighting.lights,  
    encoder: renderEncoder,  
    uniforms: uniforms)
```

This code will display lines to visualize the direction of the sun light.

- Build and run the app.



*Debugging sunlight direction*

The red lines show the parallel sun light direction vector. As you rotate the scene, you can see that the brightest parts are the ones facing towards the sun.

**Note:** the debug method uses `.line` as the rendering type. Unfortunately line width is not configurable on the GPU, so the lines may disappear at certain angles when they are too thin to render.

This shading is pleasing, but not accurate. Take a look at the back of the sphere. The back of the sphere is black; however, you can see that the top of the green surround is bright green because it's facing up. In the real-world, the surround would be blocked by the sphere and so be in the shade. However, you're currently not taking occlusion into account, and you won't be until you master shadows in Chapter 13, “Shadows”.

## Ambient Reflection

In the real-world, colors are rarely pure black. There's light bouncing about all over the place. To simulate this, you can use ambient lighting. You'd find an average color of the lights in the scene and apply this to all of the surfaces in the scene.

- Open **SceneLighting.swift**, and add an ambient light property:

```
let ambientLight: Light = {  
    var light = Self.buildDefaultLight()  
    light.color = [0.05, 0.1, 0]  
    light.type = Ambient  
    return light  
}()
```

This light is a slightly green tint.

- Add this to the end of `init()`:

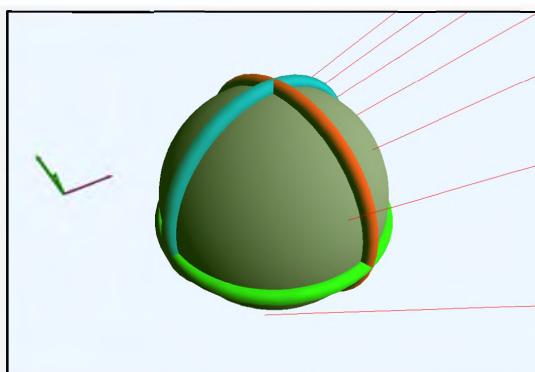
```
lights.append(ambientLight)
```

- Open **Lighting.metal**, and above break in case `Ambient`, add this:

```
ambientColor += light.color;
```

- Build and run the app. The scene is now tinged green as if there is a green light being bounced around. Change `light.intensity` in `SceneLighting` if you want less pronounced ambient light.

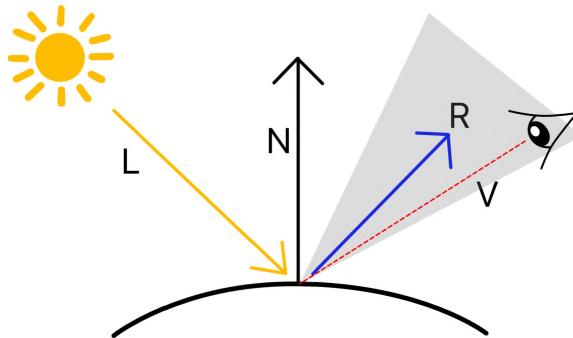
This image has an ambient color of `[0.05, 0.2, 0]`:



*Ambient lighting*

## Specular Reflection

Last but not least in the Phong reflection model, is the specular reflection. You now have a chance to put a coat of shiny varnish on the sphere. The specular highlight depends upon the position of the observer. If you pass a shiny car, you'll only see the highlight at certain angles.



*Specular reflection*

The light comes in (L) and is reflected (R) about the normal (N). If the viewer (V) is within a particular cone around the reflection (R), then the viewer will see the specular highlight. That cone is an exponential shininess parameter. The shinier the surface is, the smaller and more intense the specular highlight.

In your case, the viewer is your camera so you'll need to pass the camera coordinates, again in world position, to the fragment function. Earlier, you set up a `cameraPosition` property in `params`, and this is what you'll use to pass the camera position.

► Open `Renderer.swift`, and in `updateUniforms(scene:)`, add this:

```
params.cameraPosition = scene.camera.position
```

`scene.camera.position` is already in world space, and you're already passing `params` to the fragment function, so you don't need to take further action here.

- Open **Lighting.metal**, and in **phongLighting**, add the following variables to the top of the function:

```
float materialShininess = 32;
float3 materialSpecularColor = float3(1, 1, 1);
```

These hold the surface material properties of a shininess factor and the specular color. As these are surface properties, you should be getting these values from each model's materials, and you'll do that in the following chapter.

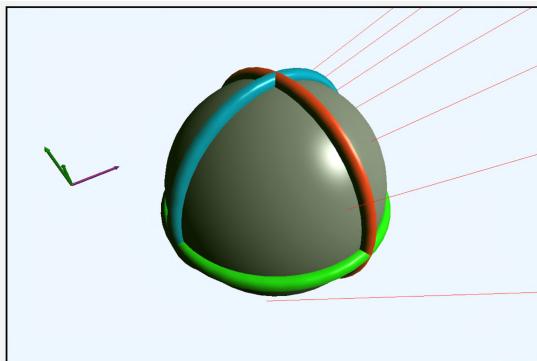
- Above **break** in case **Sun**, add the following:

```
if (diffuseIntensity > 0) {
    // 1 (R)
    float3 reflection =
        reflect(lightDirection, normal);
    // 2 (V)
    float3 viewDirection =
        normalize(params.cameraPosition);
    // 3
    float specularIntensity =
        pow(saturate(dot(reflection, viewDirection)),
            materialShininess);
    specularColor +=
        light.specularColor * materialSpecularColor
        * specularIntensity;
}
```

Going through this code:

1. For the calculation of the specular color, you'll need (L)ight, (R)election, (N)ormal and (V)iew. You already have (L) and (N), so here you use the Metal Shading Language function `reflect` to get (R).
2. You need the view vector between the fragment and the camera for (V).
3. Now you calculate the specular intensity. You find the angle between the reflection and the view using the dot product, clamp the result between 0 and 1 using `saturate`, and raise the result to a shininess power using `pow`. You then use this intensity to work out the specular color for the fragment.

- Build and run the app to see your completed lighting.



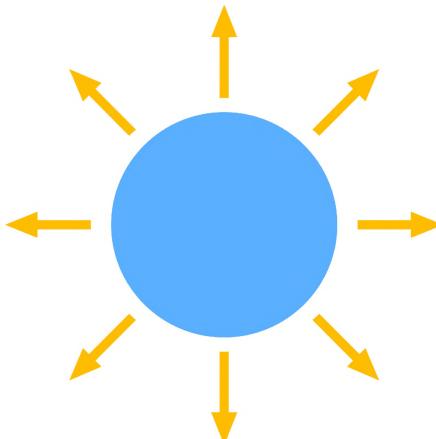
*Specular reflection*

Experiment with changing `materialShininess` from 2 to 1600. In Chapter 11, “Maps & Materials”, you’ll find out how to read in material and texture properties from the model to change its color and lighting.

You’ve created a realistic enough lighting situation for a sun. You can add more variety to your scene with point and spot lights.

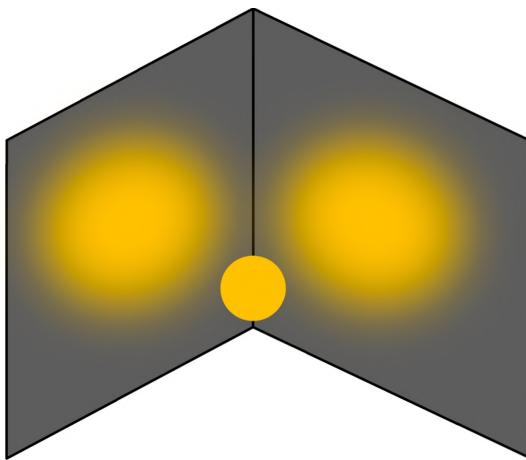
## Point Lights

As opposed to the sun light, where you converted the position into parallel direction vectors, point lights shoot out light rays in all directions.



*Point light direction*

A light bulb will only light an area of a certain radius, beyond which everything is dark. So you'll also specify **attenuation** where a ray of light doesn't travel infinitely far.



*Point light attenuation*

Light attenuation can occur abruptly or gradually. The original OpenGL formula for attenuation is:

$$\frac{1.0}{x + y * \textit{distance} + z * \textit{distance}^2}$$

Where **x** is the constant attenuation factor, **y** is the linear attenuation factor and **z** is the quadratic attenuation factor.

The formula gives a curved fall-off. You'll represent **xyz** with a **float3**. No attenuation at all will be **float3(1, 0, 0)** — substituting x, y and z into the formula results in a value of 1.

► Open **SceneLighting.swift**, and add a point light property to **SceneLighting**:

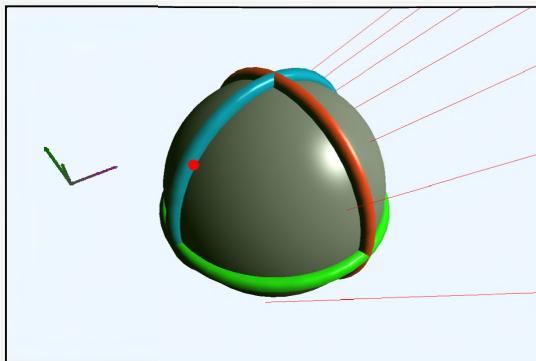
```
let redLight: Light = {
    var light = Self.buildDefaultLight()
    light.type = Point
    light.position = [-0.8, 0.76, -0.18]
    light.color = [1, 0, 0]
    light.attenuation = [0.5, 2, 1]
    return light
}()
```

Here, you create a red point light with a position and attenuation. You can experiment with the attenuation values to change radius and fall-off.

- Add the light to `lights` in `init()`:

```
lights.append(redLight)
```

- Build and run the app.



*Debugging a point light*

You'll see a small red dot which marks the position of the point light rendered by `DebugLights`.

**Note:** The shader for the point light debug dot is worth looking at. In `DebugLights.metal`, in `fragment_debug_point`, the default square point is turned into a circle by discarding fragments greater than a certain radius from the center of the point.

The debug lights function shows you where the point light is, but it does not produce any light yet. You'll do this in the fragment shader.

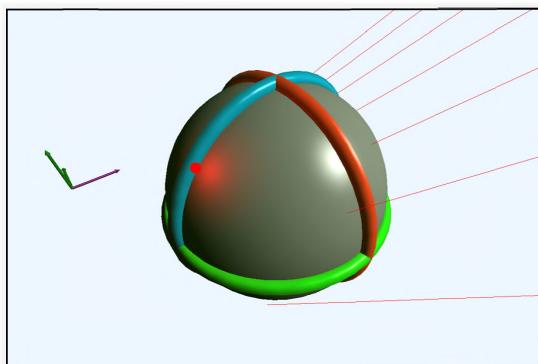
- Open `Lighting.metal`, and in `phongLighting`, add this above `break` in case Point:

```
// 1
float d = distance(light.position, position);
// 2
float3 lightDirection = normalize(light.position - position);
// 3
float attenuation = 1.0 / (light.attenuation.x +
    light.attenuation.y * d + light.attenuation.z * d * d);
```

```
float diffuseIntensity =  
    saturate(dot(lightDirection, normal));  
float3 color = light.color * baseColor * diffuseIntensity;  
// 4  
color *= attenuation;  
diffuseColor += color;
```

Going through this code:

1. You find out the distance between the light and the fragment position.
  2. With the directional sun light, you used the position as the light direction. Here, you calculate the direction from the fragment position to the light position.
  3. Calculate the attenuation using the attenuation formula and the distance to see how bright the fragment will be.
  4. After calculating the diffuse color as you did for the sun light, multiply this color by the attenuation.
- Build and run the app, and you'll see the full effect of the red point light.



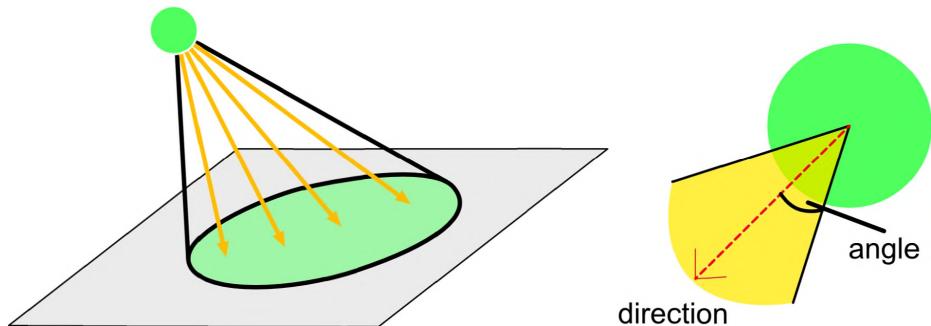
*Rendering a point light*

Remember the sphere is slightly green because of the ambient light.

## Spotlights

The last type of light you'll create in this chapter is the spotlight. This sends light rays in limited directions. Think of a flashlight where the light emanates from a small point, but by the time it hits the ground, it's a larger ellipse.

You define a cone angle to contain the light rays with a cone direction. You also define a cone power to control the attenuation at the edge of the ellipse.



*Spotlight angle and attenuation*

► Open **SceneLighting.swift**, and add a new light:

```
lazy var spotlight: Light = {
    var light = Self.buildDefaultLight()
    light.type = Spot
    light.position = [-0.64, 0.64, -1.07]
    light.color = [1, 0, 1]
    light.attenuation = [1, 0.5, 0]
    light.coneAngle = Float(40).degreesToRadians
    light.coneDirection = [0.5, -0.7, 1]
    light.coneAttenuation = 8
    return light
}()
```

This light is similar to the point light with the added cone angle, direction and cone attenuation.

► Add the light to lights in `init()`:

```
lights.append(spotlight)
```

► Open **Lighting.metal**, and in `phongLighting`, add this code above break in case Spot:

```
// 1
float d = distance(light.position, position);
float3 lightDirection = normalize(light.position - position);
// 2
float3 coneDirection = normalize(light.coneDirection);
float spotResult = dot(lightDirection, -coneDirection);
// 3
```

```

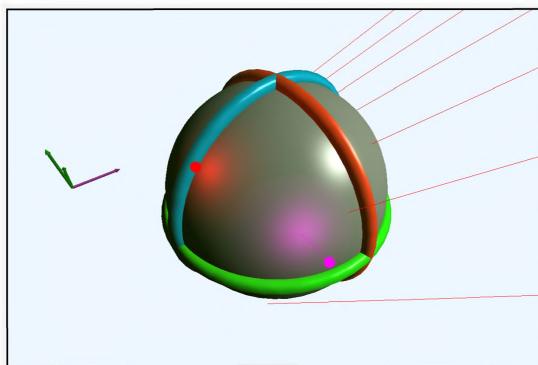
if (spotResult > cos(light.coneAngle)) {
    float attenuation = 1.0 / (light.attenuation.x +
        light.attenuation.y * d + light.attenuation.z * d * d);
    // 4
    attenuation *= pow(spotResult, light.coneAttenuation);
    float diffuseIntensity =
        saturate(dot(lightDirection, normal));
    float3 color = light.color * baseColor * diffuseIntensity;
    color *= attenuation;
    diffuseColor += color;
}

```

This code is very similar to the point light code. Going through the comments:

1. Calculate the distance and direction as you did for the point light. This ray of light may be outside of the spot cone.
2. Calculate the cosine angle (that's the dot product) between that ray direction and the direction the spot light is pointing.
3. If that result is outside of the cone angle, then ignore the ray. Otherwise, calculate the attenuation as for the point light. Vectors pointing in the same direction have a dot product of 1.0.
4. Calculate the attenuation at the edge of the spot light using coneAttenuation as the power.

► Build and run the app.



*Rendering a spotlight*

Experiment with changing the various attenuations. A cone angle of 5° with attenuation of (1, 0, 0) and a cone attenuation of 1000 will produce a very small targeted soft light; whereas a cone angle of 20° with a cone attenuation of 1 will produce a sharp-edged round light.

## Key Points

- Shading is the reason why objects don't look flat. Lights provide illumination from different directions.
- Normals describe the slope of the curve at a point. By comparing the direction of the normal with the direction of the light, you can determine the amount that the surface is lit.
- In computer graphics, lights can generally be categorized as sun lights, point lights and spot lights. In addition, you can have area lights and surfaces can emit light. These are only approximations of real-world lighting scenarios.
- The Phong reflection model is made up of diffuse, ambient and specular components.
- Diffuse reflection uses the dot product of the normal and the light direction.
- Ambient reflection is a value added to all surfaces in the scene.
- Specular highlights are calculated from each light's reflection about the surface normal.

## Where to Go From Here?

You've covered a lot of lighting information in this chapter. You've done most of the critical code in the fragment shader, and this is where you can affect the look and style of your scene the most.

You've done some weird and wonderful calculations by working out dot products between surface normals and various light directions. The formulas you used in this chapter are a small cross-section of computer graphics research that various brilliant mathematicians have come up with over the years. If you want to read more about lighting, you'll find some interesting internet sites listed in `references.markdown` in the `resources` folder for this chapter.

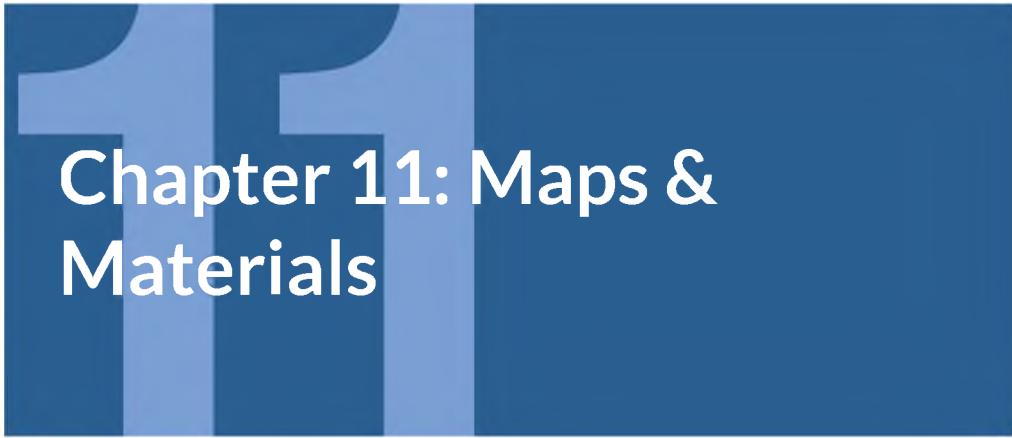
In the next chapter, you'll continue learning another important method of changing how a surface looks with texture maps and materials.



# Section II: Intermediate Metal

With the basics under your belt, you can move on to multi-pass rendering. You'll add shadows and learn several new rendering techniques. Programming the GPU using compute shaders can be intimidating, so you'll create particle systems to learn how fast multi-threaded solutions can be.





# Chapter 11: Maps & Materials

In the previous chapter, you set up a simple Phong lighting model. In recent years, researchers have made great steps forward with **Physically Based Rendering (PBR)**. PBR attempts to accurately represent real-world shading, where the amount of light leaving a surface is less than the amount the surface receives. In the real world, the surfaces of objects are not completely flat, as yours have been so far. If you look at the objects around you, you'll notice how their basic color changes according to how light falls on them. Some objects have a smooth surface, and some have a rough surface. Heck, some might even be shiny metal!

In this chapter, you'll find out how to use material groups to describe a surface, and how to design textures for micro detail.



## Normal Maps

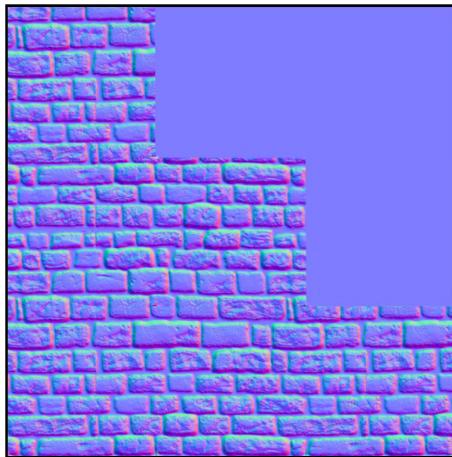
The following example best describes normal maps:



*An object rendered with a normal map*

On the left, there's a lit cube with a color texture. On the right, there's the same low-poly cube with the identical color texture and lighting. The only difference is that the cube on the right also has a second texture applied to it known as a **normal map**. This normal map makes it appear as if the cube is a high-poly model with lots of nooks and crannies. In truth, these high-end details are just an illusion.

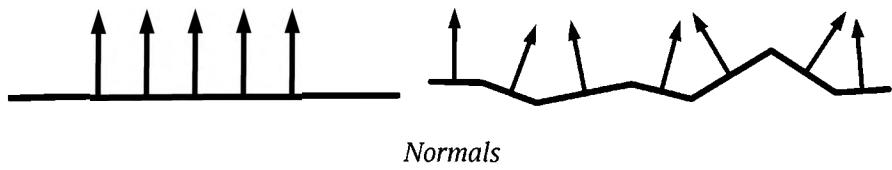
For this illusion to work, the model needs a texture, like this:



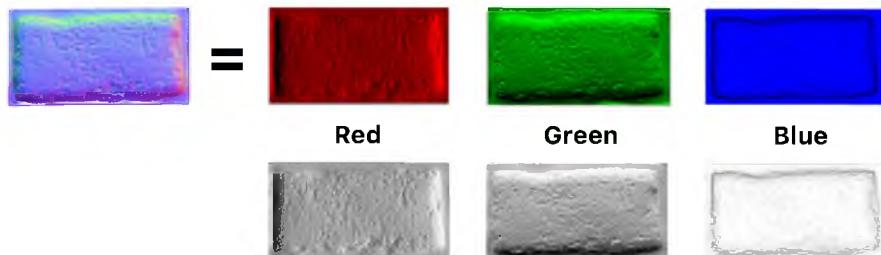
*A normal map texture*

All models have normals that stick out perpendicular to each face. A cube has six faces, and the normal for each face points in a different direction. Also, each face is flat. If you wanted to create the illusion of bumpiness, you'd need to change a normal in the fragment shader.

Look at the following image. On the left is a flat surface with normals in the fragment shader. On the right, you see perturbed normals. The texels in a normal map supply the direction vectors of these normals through the RGB channels.



Now, look at this single brick split out into the red, green and blue channels that make up an RGB image.



*Normal map channels*

Each channel has a value between 0 and 1, and you generally visualize them in grayscale as it's easier to read color values. For example, in the red channel, a value of 0 is no red at all, while a value of 1 is full red. When you convert 0 to an RGB color ( $0, 0, 0$ ), the result is black. On the opposite spectrum, ( $1, 1, 1$ ) is white. And in the middle, you have ( $0.5, 0.5, 0.5$ ), which is mid-gray. In grayscale, all three RGB values are the same, so you only need to refer to a grayscale value by a single float.

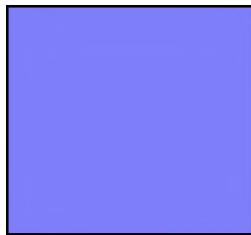
Take a closer look at the edges of the red channel's brick. Look at the left and right edges in the grayscale image. The red channel has the darkest color where the normal values of that fragment should point left ( $-X, 0, 0$ ), and the lightest color where they should point right ( $+X, 0, 0$ ).

Now look at the green channel. The left and right edges have equal value but are different for the top and bottom edges of the brick. The green channel in the grayscale image has darkest for pointing down ( $0, -Y, 0$ ) and lightest for pointing up ( $0, +Y, 0$ ).

Finally, the blue channel is mostly white in the grayscale image because the brick — except for a few irregularities in the texture — points outward. The edges of the brick are the only places where the normals should point away.

**Note:** Normal maps can be either right-handed or left-handed. Your renderer will expect positive y to be up, but some apps will generate normal maps with positive y down. To fix this, you can take the normal map into Photoshop and invert the green channel.

The base color of a normal map — where all normals are “normal” (orthogonal to the face) — is  $(0.5, 0.5, 1)$ .



*A flat normal map*

This is an attractive color but was not chosen arbitrarily. RGB colors have values between 0 and 1, whereas a model’s normal values are between -1 and 1. A color value of 0.5 in a normal map translates to a model normal of 0. The result of reading a flat texel from a normal map should be a z value of 1 and the x and y values as 0. Converting these values  $(0, 0, 1)$  into the colorspace of a normal map results in the color  $(0.5, 0.5, 1)$ . This is why most normal maps appear bluish.

## Creating Normal Maps

To create successful normal maps, you need a specialized app. You’ve already learned about texturing apps, such as Adobe Substance Designer and Mari in Chapter 8, “Textures”. Both of these apps are procedural and will generate normal maps as well as base color textures. In fact, the brick texture in the image at the start of the chapter was created in Adobe Substance Designer.

Sculpting programs, such as ZBrush, 3D-Coat, Mudbox and Blender will also generate normal maps from your sculptures. You first sculpt a detailed high-poly mesh. And then the app looks at the cavities and curvatures of your sculpt and **bakes** a normal map. Because high-poly meshes with tons of vertices aren’t resource-efficient in games, you should create a low-poly mesh and then apply the normal map to this mesh.

Photoshop CC (from 2015) and Adobe Substance 3D Sampler can generate a normal map from a photograph or diffuse texture. Because these apps look at the shading and calculate the values, they aren't as good as the sculpting or procedural apps, but it can be quite amazing to take a photograph of a real-life, personal object, run it through one of these apps, and render out a shaded model.

Here's a normal map that was created using Allegorithmic's legacy app Bitmap2Material:



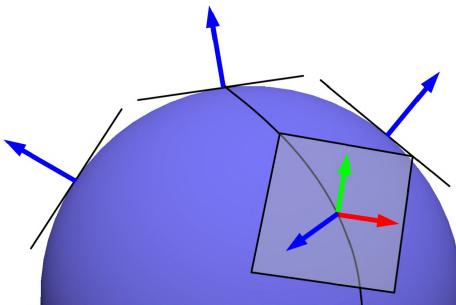
*A cross photographed and converted into a normal map*

On the right, the normal map with a white color texture is rendered on to the same cube model as before, with minimal geometry.

## Tangent Space

To render with a normal map texture, you send it to the fragment function in the same way as a color texture, and you extract the normal values using the same UVs. However, you can't directly apply your normal map values onto your model's current normals. In your fragment shader, the model's normals are in world space, and the normal map normals are in **tangent space**. Tangent space is a little hard to wrap your head around. Think of the brick cube with all its six faces pointing in different directions. Now think of the normal map with all the bricks the same color on all the six faces.

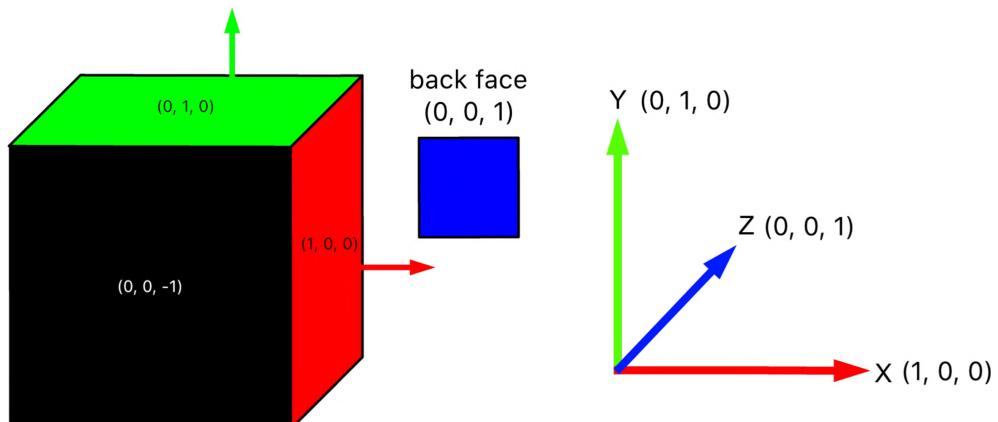
If a cube face is pointing toward negative x, how does the normal map know to point in that direction?



*Normals on a sphere*

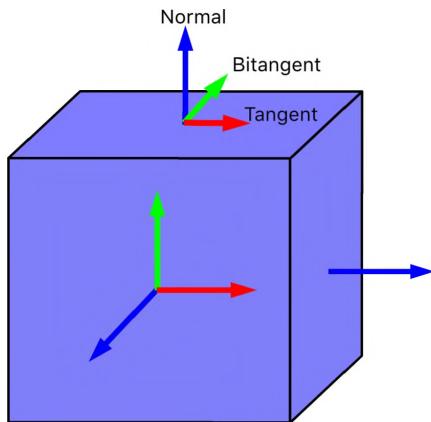
Using a sphere as an example, every fragment has a tangent — that's the line that touches the sphere at that point. The normal vector in this tangent space is thus relative to the surface. You can see that all of the arrows are at right angles to the tangent. So if you took all of the tangents and laid them out on a flat surface, the blue arrows would point upward in the same direction. That's tangent space!

The following image shows a cube's normals in world space.



*Visualizing normals in world space*

To convert the cube's normals to tangent space, you create a **TBN matrix** - that's a **Tangent Bitangent Normal matrix** that's calculated from the tangent, bitangent and normal value for each vertex.



*The TBN matrix*

In the TBN matrix, the normal is the perpendicular vector as usual; the tangent is the vector that points along the horizontal surface; and the bitangent is the vector — as calculated by the **cross product** — that is perpendicular to both the tangent and the normal.

**Note:** The cross product is an operation that gives you a vector perpendicular to two other vectors.

The tangent can be at right angles to the normal in any direction. However, to share normal maps across different parts of models, and even entirely different models, there are two standards:

1. The tangent and bitangent will represent the directions that  $u$  and  $v$  point, respectively, defined in model space.
2. The red channel will represent curvature along  $u$ , and the green channel, along  $v$ .

You could calculate these values when you load the model. However, with Model I/O, as long as you have data for both the position and texture coordinate attributes, Model I/O can calculate and store these tangent and bitangent values at each vertex for you.

Finally some code! :]

## The Starter App

- In Xcode, open the starter project for this chapter.

There are different models in the project, with accompanying textures in **Textures.xcassets**. There are two lights in the scene - a sun light and a gentle directional fill light from the back. The code is the same as at the end of the previous chapter with the exception of `GameScene` and `SceneLighting`, which simply set up the different models and lighting. Pressing keys “1” and “2” take you to the front and default views respectively.

- Build and run the app, and you’ll see a quaint cartoon cottage.



*A cartoon cottage*

It’s a bit plain, but you’re going to add a normal map to help give it some surface details.

## Using Normal Maps

- In the `Models > Cottage` group, open `cottage1.mtl` in a text editor.

There are two textures needed to render this cottage:

```
map_tangentSpaceNormal cottage-normal  
map_Kd cottage-color
```

`map_Kd` defines the color map, and `map_tangentSpaceNormal` defines the normal map. `cottage-color` and `cottage-normal` are textures in **Textures.xcassets**.

The normal map holds data instead of color. Later, you'll use other texture maps for other surface qualities. Looking at textures like **cottage-normal** in a photo editor, you'd think they are color, but the trick is to regard the RGB values as numerical data instead of color data.

- In the **Geometry** group, open **Submesh.swift**, and add a new property to **Submesh.Textures**:

```
let normal: MTLTexture?
```

- At the end of **SubMesh.Textures.init(material:)**, read in this texture:

```
normal = property(with: .tangentSpaceNormal)
```

This is the normal map property that Model I/O expects to read in from the **.mtl** file.

- In the **Shaders** group, open **Common.h**, and add this to **TextureIndices**:

```
NormalTexture = 1
```

You'll send the normal texture to the fragment shader using this index.

- Open **Model.swift**, and in **render(encoder:uniforms:params:)**, locate where you set the base color texture inside for **submesh** in **mesh.submeshes**.

- Add this:

```
encoder.setFragmentTexture(  
    submesh.textures.normal,  
    index: NormalTexture.index)
```

Here, you send the normal texture to the GPU.

- Open **Shaders.metal**, and in **fragment\_main**, add the normal texture to the list of parameters:

```
texture2d<float> normalTexture [[texture(NormalTexture)]]
```

Now that you're transferring the normal texture map, the first step is to apply it to the cottage as if it were a color texture.

- In **fragment\_main**, before calling **phongLighting**, add this:

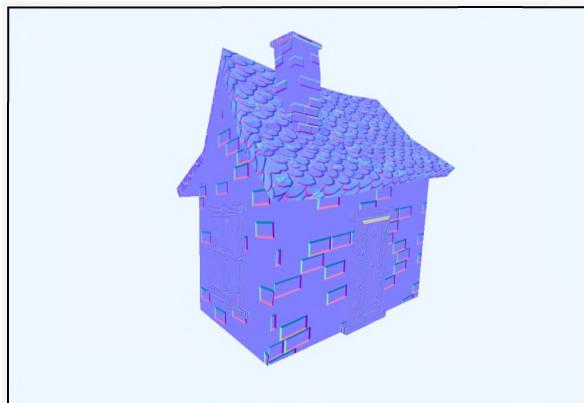
```
float3 normal;  
if (is_null_texture(normalTexture)) {  
    normal = in.worldNormal;
```



```
    } else {
        normal = normalTexture.sample(
            textureSampler,
            in.uv * params.tiling).rgb;
    }
    normal = normalize(normal);
    return float4(normal, 1);
```

This reads in `normalValue` from the texture, if there is one. If there is no normal map texture for this model, set the default normal value. The `return` is only temporary to make sure the app is loading the normal map correctly, and that the normal map and UVs match.

- Build and run to verify the normal map is providing the fragment color.



*The normal map applied as a color texture*

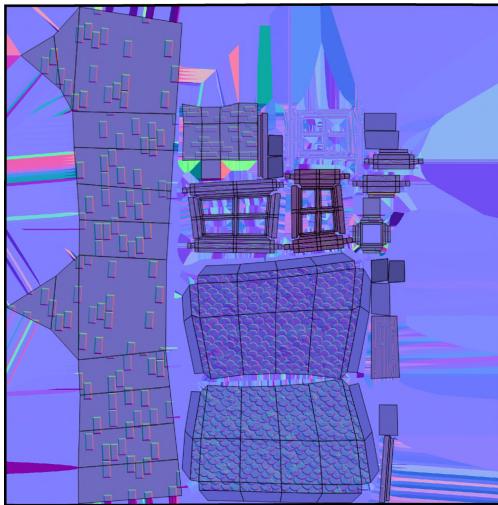
You can see all the surface details the normal map will provide. There are scattered bricks on the wall, wood grain on the door and windows and a shingle-looking roof.

- Excellent! You tested that the normal map loads, so remove this from `fragment_main`:

```
    return float4(normal, 1);
```

You may have noticed that in the normal map's bricks, along the main surface of the house, the red seems to point along negative y, and the green seems to map to negative x.

You might expect that red  $(1, 0, 0)$  maps to  $x$  and green  $(0, 1, 0)$  maps to  $y$ . This happens because the UV island for the main part of the house is rotated 90 degrees counterclockwise.



*A normal map overlaid with UVs*

Not to worry, the mesh's stored tangents will map everything correctly. They take UV rotation into account.

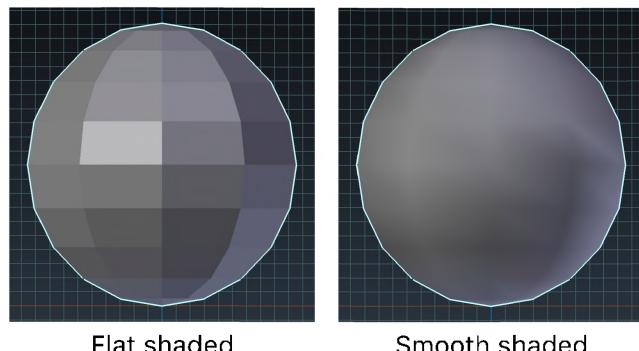
Don't celebrate just yet. You still have several tasks ahead of you. You still need to:

1. Load tangent and bitangent values using Model I/O.
2. Tell the render command encoder to send the newly created `MTLBuffers` containing the values to the GPU.
3. In the vertex shader, change the values to world space — just as you did normals — and pass the new values to the fragment shader.
4. Calculate the new normal based on these values.

## 1. Load Tangents and Bitangents

► Open `VertexDescriptor.swift`, and look at `MDLVertexDescriptor`'s `defaultLayout`. Here, you tell the vertex descriptor that there are normal values in the attribute named `MDLVertexAttribNormal`.

So far, your models have normal values included with them, but you may come across odd files where you have to generate normals. You can also override how the modeler **smoothed** the model. For example, the house model has smoothing applied in Blender so that the roof, which has very few faces, does not appear too blocky.



*Flat vs smooth shaded*

Smoothing recalculates vertex normals so that they interpolate smoothly over a surface. Blender stores smoothing groups in the .obj file, which Model I/O reads in and understands. Notice in the above image, that although the surface of the sphere is smooth, the edges are unchanged and pointy. Smoothing only changes the way the renderer evaluates the surface. Smoothing does not change the geometry.

Try reloading vertex normals and overriding the smoothing.

► Open **Model.swift**, and in `init(name:)`, replace:

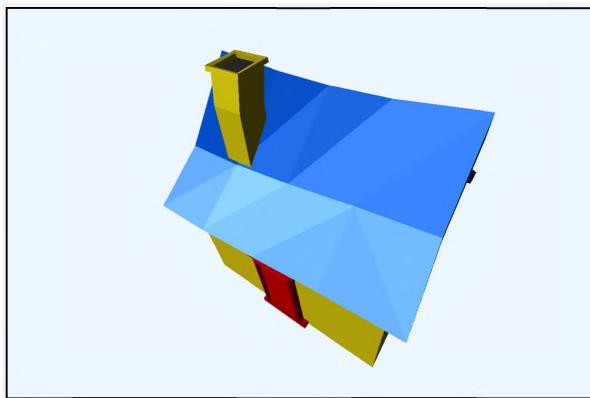
```
let (mdlMeshes, mtkMeshes) = try! MTKMesh.newMeshes(  
    asset: asset,  
    device: Renderer.device)
```

With:

```
var mtkMeshes: [MTKMesh] = []  
let mdlMeshes =  
    asset.childObjects(of: MDLMesh.self) as? [MDLMesh] ?? []  
    = mdlMeshes.map { mdlMesh in  
        mdlMesh.addNormals(  
            withAttributeNamed: MDLVertexAttributeNormal,  
            creaseThreshold: 1.0)  
        mtkMeshes.append(  
            try! MTKMesh(  
                mesh: mdlMesh,  
                device: Renderer.device))  
    }
```

You're now loading the MDLMeshes first and changing them before initializing the MTKMeshes. You ask Model I/O to recalculate normals with a crease threshold of 1. This crease threshold, between 0 and 1, determines the smoothness, where 1.0 is unsmoothed.

- Build and run the app.



*An unsmoothed model*

The cottage is now completely unsmoothed, and you can see all of its separate faces. If you were to try a `creaseThreshold` of zero, where everything is smoothed, you'd get some rendering artifacts because of the surfaces rounding too far. When dealing with smoothness remember this: Smoothness is good, but use it with caution. The artist needs to set up the model with smoothing in mind.

- Remove the line you just added that reads:

```
mdlMesh.addNormals(  
    withAttributeNamed: MDLVertexAttributeNormal,  
    creaseThreshold: 1.0)
```

- Replace it with this:

```
mdlMesh.addTangentBasis(  
    forTextureCoordinateAttributeNamed:  
        MDLVertexAttributeTextureCoordinate,  
    tangentAttributeNamed: MDLVertexAttributeTangent,  
    bitangentAttributeNamed: MDLVertexAttributeBitangent)
```

All the supplied models have normals provided by Blender, but not tangents and bitangents. This new code generates and loads the vertex tangent and bitangent values.

Model I/O does a few things behind the scenes:

- Add two named attributes to `mdlMesh`'s vertex descriptor:  
`MDLVertexAttributeTangent` and `MDLVertexAttributeBitangent`.
- Calculate the tangent and bitangent values.
- Create two new `MTLBuffers` to contain them.
- Update the layout strides on `mdlMesh`'s vertex descriptor to match the two new buffers.

With the addition of these two new attributes, you should update the default vertex descriptor so that the pipeline state in `Renderer` uses the same vertex descriptor.

First, define the new buffer attribute and buffer indices.

► Open `Common.h` and add this to `Attributes`:

```
Tangent = 4,  
Bitangent = 5
```

► Add the indices to `BufferIndices`:

```
TangentBuffer = 3,  
BitangentBuffer = 4,
```

► Open `VertexDescriptor.swift`, and add this to `MDLVertexDescriptor`'s `defaultLayout` before `return`:

```
vertexDescriptor.attributes[Tangent.index] =  
    MDLVertexAttribute(  
        name: MDLVertexAttributeTangent,  
        format: .float3,  
        offset: 0,  
        bufferIndex: TangentBuffer.index)  
vertexDescriptor.layouts[TangentBuffer.index]  
    = MDLVertexBufferLayout(stride: MemoryLayout<float3>.stride)  
vertexDescriptor.attributes[Bitangent.index] =  
    MDLVertexAttribute(  
        name: MDLVertexAttributeBitangent,  
        format: .float3,  
        offset: 0,  
        bufferIndex: BitangentBuffer.index)  
vertexDescriptor.layouts[BitangentBuffer.index]  
    = MDLVertexBufferLayout(stride: MemoryLayout<float3>.stride)
```

When you create the pipeline state in `Renderer`, the pipeline descriptor will now notify the GPU that it needs to create space for these two extra buffers. It's important that you remember that your model's vertex descriptor layout must match the one in the render encoder's pipeline state. In addition, your shader's `VertexIn` attributes should also match the vertex descriptor.

**Note:** So far, you've only created one pipeline descriptor for all models. But often models will require different vertex layouts. Or if some of your models don't contain normals, colors and tangents, you might wish to save on creating buffers for them. You can create multiple pipeline states for the different vertex descriptor layouts, and replace the render encoder's pipeline state before drawing each model.

- Build and run the app to make sure your cottage still renders.



*The cottage - no changes yet*

You've completed the necessary updates to the model's vertex layouts, and now you'll update the rendering code to match.

## 2. Send Tangent and Bitangent Values to the GPU

- Open `Model.swift`, and in `render(encoder:uniforms:params:)`, locate for `mesh` in `meshes`.

For each mesh, you're currently sending all the vertex buffers to the GPU:

```
for (index, vertexBuffer) in mesh.vertexBuffers.enumerated() {  
    encoder.setVertexBuffer(  
}
```

```
    vertexBuffer,  
    offset: 0,  
    index: index  
}
```

This code includes sending the tangent and bitangent buffers. You should be aware of the number of buffers that you send to the GPU. In **Common.h**, you've set up **UniformsBuffer** as index **11**, but if you had defined that as index **4**, you'd now have a conflict with the bitangent buffer.

### 3. Convert Tangent and Bitangent Values to World Space

Just as you converted the model's normals to world space, you need to convert the tangents and bitangents to world space in the vertex function.

- Open **Shaders.metal**, and add these new attributes to **VertexIn**:

```
float3 tangent [[attribute(Tangent)]];  
float3 bitangent [[attribute(Bitangent)]];
```

- Add new properties to **VertexOut** so that you can send the values to the fragment function:

```
float3 worldTangent;  
float3 worldBitangent;
```

- In **vertex\_main** after calculating **out.worldNormal**, add this:

```
.worldTangent = uniforms.normalMatrix * in.tangent,  
.worldBitangent = uniforms.normalMatrix * in.bitangent
```

This code moves the tangent and bitangent values into world space.

### 4. Calculate the New Normal

Now that you have everything in place, it'll be a simple matter to calculate the new normal.

Before doing the normal calculation, consider the normal color value that you're reading. Colors are between 0 and 1, but normal values range from -1 to 1.

- Still in **Shaders.metal**, in **fragment\_main**, locate where you read **normalTexture**. In the **else** part of the conditional, after reading the normal from the texture, add:



```
normal = normal * 2 - 1;
```

This code redistributes the normal value to be within the range -1 to 1.

- After the previous code, still inside the `else` part of the conditional, add this:

```
normal = float3x3(  
    in.worldTangent,  
    in.worldBitangent,  
    in.worldNormal) * normal;
```

This code recalculates the normal direction into tangent space to match the tangent space of the normal texture.

- Replace the `color` definition with:

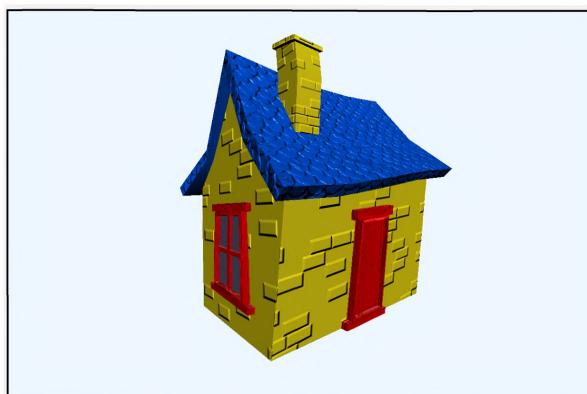
```
float3 color = phongLighting(  
    normal,  
    in.worldPosition,  
    params,  
    lights,  
    baseColor  
) ;
```

You send `phongLighting` your newly calculated normal.

- Remove `normalDirection` since you no longer need it:

```
float3 normalDirection = normalize(in.worldNormal);
```

- Build and run the app to see the normal map applied to the cottage.



*The cottage with a normal map applied*

As you rotate the cottage, notice how the lighting affects the small cavities on the model, especially on the door and roof where the specular light falls — it's almost like you created new geometry, but you didn't. That's the magic of normal apps: Adding amazing detail to simple low-poly models.

## Other Texture Map Types

Normal maps are not the only way of changing a model's surface. There are other texture maps:

- **Roughness:** Describes the smoothness or roughness of a surface. You'll add a roughness map shortly.
- **Metallic:** White for metal and black for dielectric. Metal is a conductor of electricity, whereas a dielectric material is a non-conductor.
- **Ambient Occlusion:** Describes areas that are occluded; in other words, areas that are hidden from light.
- **Reflection:** Identifies which part of the surface is reflective.
- **Opacity:** Describes the location of the transparent parts of the surface.

In fact, any value (thickness, curvature, etc.) that you can think of to describe a surface, can be stored in a texture. You just look up the relevant fragment in the texture using the UV coordinates and use the value recovered. That's one of the bonuses of writing your own renderer. You can choose what maps to use and how to apply them.

You can use all of these textures in the fragment shader, and the geometry doesn't change.

**Note:** A displacement or height map can change geometry. You'll read about displacement in Chapter 19, "Tessellation & Terrains".

## Materials

Not all models have textures. For example, the train you rendered earlier in the book has different material groups that specify a color instead of using a texture.



In the **Models ▶ Cottage** group, open **cottage1.mtl** in a text editor. This is the file that describes the visual aspects of the cottage model. There are several material groups:

- glass
- roof
- wall
- wood

Each of these groups contain different material properties. Although you're loading the same diffuse texture for all of them, you could use different textures for each group.

In the case of a property not having an associated texture, you can extract the values:

- Ns: Specular exponent (shininess)
- Kd: Diffuse color
- Ks: Specular color

**Note:** You can find a full list of the definitions at <https://bit.ly/3HBVnGn>.

The current **.mtl** file loads the color using **map\_Kd**, but for experimentation, you'll switch the rendered cottage file to one that gets its color from the material group and not a texture.

- Open **cottage2.mtl**, and see that none of the groups has a **map\_Kd** property.
- Open **GameScene.swift**, and change **cottage** to use "**cottage2.obj**" instead of "**cottage1.obj**".

The diffuse color won't be the only material property you'll be reading.

- Open **Common.h**, and add a new structure to hold material values:

```
typedef struct {
    vector_float3 baseColor;
    vector_float3 specularColor;
    float roughness;
    float metallic;
    float ambientOcclusion;
```



```
    float shininess;
} Material;
```

There are more material properties available, but these are the most common. For now, you'll read in `baseColor`, `specularColor` and `shininess`.

- Open **Submesh.swift**, and create a new property in **Submesh** under **textures** to hold the materials:

```
let material: Material
```

Your project won't compile until you've initialized `material`.

- At the bottom of **Submesh.swift**, create a new **Material** extension with initializer:

```
private extension Material {
    init(material: MDLMaterial?) {
        self.init()
        if let baseColor = material?.property(with: .baseColor),
           baseColor.type == .float3 {
            self.baseColor = baseColor.float3Value
        }
    }
}
```

In **Submesh.Textures**, you read in string values for the textures' file names from the submesh's material properties. If there's no texture available for a particular property, you can use the material base color. For example, if an object is solid red, you don't have to go to the trouble of making a texture, you can just use the material's base color of `float3(1, 0, 0)` to describe the color.

- Add the following code to the end of **Material**'s `init(material:)`:

```
if let specular = material?.property(with: .specular),
   specular.type == .float3 {
    self.specularColor = specular.float3Value
}
if let shininess = material?.property(with: .specularExponent),
   shininess.type == .float {
    self.shininess = shininess.floatValue
}
self.ambientOcclusion = 1
```

Here, you read the `specular` and `shininess` values from the submesh's materials. Currently you're not loading or using ambient occlusion, but the default value should be 1.0 (white).



- In Submesh, in `init(mdlSubmesh:mtkSubmesh:)` and after initializing textures, initialize material:

```
material = Material(material: mdlSubmesh.material)
```

You'll now send this material to the shader. This sequence of coding should be familiar to you by now.

- Open **Common.h**, and add another index to BufferIndices:

```
MaterialBuffer = 14
```

- Open **Model.swift**. In `render(encoder:uniforms:params:)`, inside `for submesh in mesh.submeshes` where you call `setFragmentTexture`, add the following:

```
var material = submesh.material
encoder.setFragmentBytes(
    &material,
    length: MemoryLayout<Material>.stride,
    index: MaterialBuffer.index)
```

This code sends the material structure to the fragment shader. As long as your material structure stride is less than 4k bytes, then you don't need to create and hold a special buffer.

- Open **Shaders.metal**, and add the following as a parameter of `fragment_main`:

```
constant Material &_material [[buffer(MaterialBuffer)]],
```

You pass the model's material properties to the fragment shader. You use `_` in front of the name, as `_material` is constant, and soon you'll need to update the structure with the texture's base color if there is one.

- At the top of `fragment_main`, add this:

```
Material material = _material;
```

- In `fragment_main`, replace:

```
float3 baseColor;
if (is_null_texture(baseColorTexture)) {
    baseColor = in.color;
} else {
    baseColor = baseColorTexture.sample(
        textureSampler,
```



```
    in.uv * params.tiling).rgb;  
}
```

With:

```
if (!is_null_texture(baseColorTexture)) {  
    material.baseColor = baseColorTexture.sample(  
        textureSampler,  
        in.uv * params.tiling).rgb;  
}
```

If the texture exists, replace the material base color with the color extracted from the texture. Otherwise, you've already loaded the base color in `material`.

- Still in `fragment_main`, replace `baseColor` with `material` in `phongLighting`'s arguments:

```
float3 color = phongLighting(  
    normal,  
    in.worldPosition,  
    params,  
    lights,  
    material  
) ;
```

Your project won't compile until you've updated `phongLighting` to match these parameters.

- Open `Lighting.h`, and replace `float3 baseColor` with:

```
Material material
```

- Open `Lighting.metal`, and in `phongLighting`'s parameters, replace the parameter `float3 baseColor` with:

```
Material material
```

You're now sending `material` to `phongLighting` instead of just the base color, so you'll be able to render the appropriate material properties for each submesh.



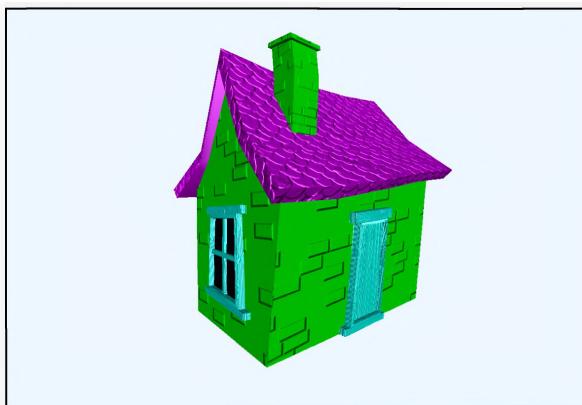
- Add the following code to the top of `phongLighting`:

```
float3 baseColor = material.baseColor;
```

- Replace the assignments of `materialShininess` and `materialSpecularColor` with:

```
float materialShininess = material.shininess;
float3 materialSpecularColor = material.specularColor;
```

- Build and run the app, and you're now loading `cottage2` with the colors coming from the material Kd values instead of a texture.



*Using color from the surface material*

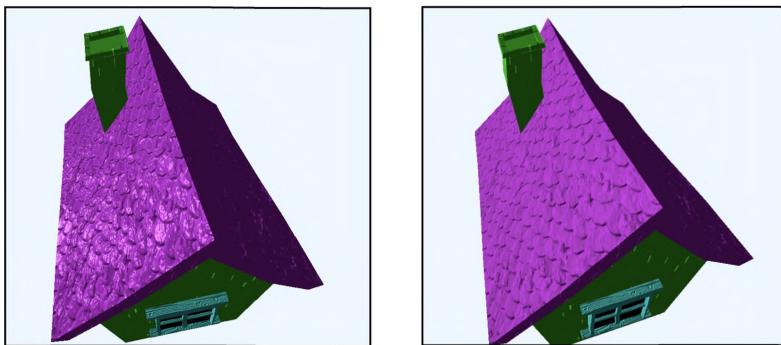
As you rotate the cottage, you can see the roof, door and window frames are shiny with strong specular highlights.

- In the **Models ▶ Cottage** group, open `cottage2.mtl` in a text editor, and in both the roof and wood groups, change:

- **Ns:** to 1.0; and
- **Ks:** to 0.2 0.2 0.2

These changes eliminate the specular highlights for those two groups.

► Build and run the app to see the difference:



*With and without specular highlight*

You can now render models with or without textures, by reading in the values in the `.mtl` file. You've also found that it's very easy to change material values by editing them in the `.mtl` file.

As you can see, models have various requirements. Some models need a color texture; some models need a roughness texture; and some models need normal maps. It's up to you to check conditionally in the fragment function whether there are textures or constant material values.

## Physically Based Rendering (PBR)

To achieve spectacular scenes, you need to have good textures, but shading plays an even more significant role. In recent years, the concept of PBR has replaced the simplistic Phong shading model. As its name suggests, PBR attempts physically realistic interaction of light with surfaces. Now that Augmented Reality has become part of our lives, it's even more important to render your models to match their physical surroundings.

The general principles of PBR are:

- Surfaces should not reflect more light than they receive.
- Surfaces can be described with known, measured physical properties.

The Bidirectional Reflectance Distribution Function (BRDF) defines how a surface responds to light. There are various highly mathematical BRDF models for both diffuse and specular, but the most common are Lambertian diffuse; and for the specular, variations on the Cook-Torrance model (presented at SIGGRAPH 1981). This takes into account:

- **micro-facet slope distribution:** You learned about micro-facets and how light bounces off surfaces in many directions in Chapter 10, “Lighting Fundamentals”.
- **Fresnel:** If you look straight down into a clear lake, you can see through it to the bottom, however, if you look across the surface of the water, you only see a reflection like a mirror. This is the Fresnel effect, where the reflectivity of the surface depends upon the viewing angle.
- **geometric attenuation:** Self-shadowing of the micro-facets.

Each of these components have different approximations, or models written by many clever people. It’s a vast and complex topic. In the resources folder for this chapter, **references.markdown** contains a few places where you can learn more about physically based rendering and the calculations involved. You’ll also learn some more about BRDF and Fresnel in Chapter 21, “Image-Based Lighting”.

Artists generally provide some textures with their models that supply the BRDF values. These are the most common:

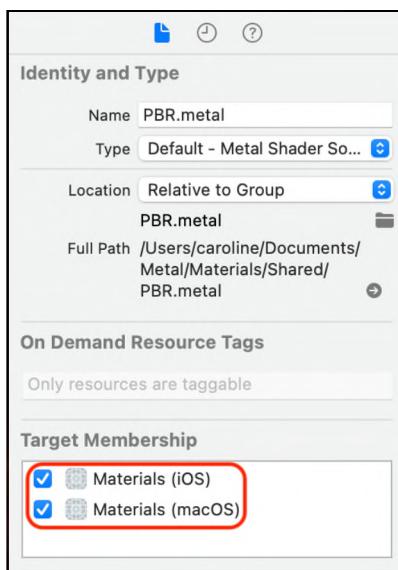
- **Albedo:** You already met the albedo map in the form of the base color map. Albedo is originally an astronomical term describing the measurement of diffuse reflection of solar radiation, but it has come to mean in computer graphics the surface color without any shading applied to it.
- **Metallic:** A surface is either a conductor of electricity — in which case it’s a metal; or it isn’t a conductor — in which case it’s a **dielectric**. Most metal textures consist of 0 (black) and 1 (white) values only: 0 for dielectric and 1 for metal.
- **Roughness:** A grayscale texture that indicates the shininess of a surface. White is rough, and black is smooth. If you have a scratched shiny surface, the texture might consist of mostly black or dark gray with light gray scratch marks.
- **Ambient Occlusion:** A grayscale texture that defines how much light reaches a surface. For example, less light will reach nooks and crannies.

Included in the starter project is a fragment function that uses a Cook-Torrance model for specular lighting. It takes as input the above textures, as well as the color and normal textures.

## PBR Workflow

First, change the fragment function to use the PBR calculations.

- Open **Renderer.swift**, and in `init(metalView:options:)`, change the name of the fragment function from "`fragment_main`" to "`fragment_PBR`".
- In the **Shaders** group, open **PBR.metal**. In the **File inspector**, add the file to the macOS and iOS targets.



*Target membership*

- Examine `fragment_PBR`.

The function starts off similar to your previous `fragment_main` but with a few more texture parameters in the function header. The function extracts values from textures when available, and calculates the normals the same as previously. For simplicity, it only processes the first light in the lights array. This is the main sun light.

`fragment_PBR` calls `computeSpecular` that works through a Cook-Torrance shading model to calculate the specular highlight. Finally, it calls `computeDiffuse` to produce the diffuse color. The final color is the result of adding together the diffuse color and the specular highlight.

To add all of the PBR textures to your project is quite long-winded, so here you'll only add roughness. You'll add metallic and ambient occlusion in the challenge.

- Open **Submesh.swift**, and create a new property for roughness in **Submesh.Textures**:

```
let roughness: MTLTexture?
```

- In the **Submesh.Textures** extension, add the following code to the end of **init(material:)**:

```
roughness = property(with: .roughness)
```

In addition to reading in a possible roughness texture, you need to read in the material value too.

- At the bottom of **Material**'s **init(material:)**, add:

```
if let roughness = material?.property(with: .roughness),  
    roughness.type == .float3 {  
    self.roughness = roughness.floatValue  
}
```

- Open **Model.swift**, and in **render(encoder:uniforms:params:)**, locate where you send the base color and normal textures to the fragment function, then add this code afterward:

```
encoder.setFragmentTexture(  
    submesh.textures.roughness,  
    index: 2)
```

- Open **GameScene.swift**, and change the name of the cottage model to “**cube.obj**”.

- In **init()**, change the camera distance and target to:

```
camera.distance = 3.5  
camera.target = .zero
```

These values fit viewing the shape and size of the cube better.

- Open **cube.mtl** in the **Models > Cube** group in a text editor. The roughness and normal maps are commented out with a **#**. The default roughness value is 1.0, which is completely rough.



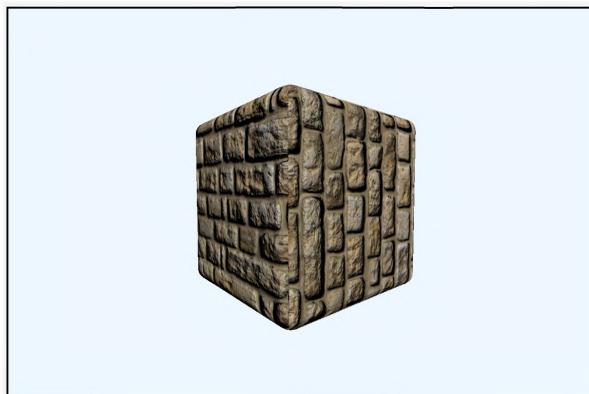
- Build and run the app to see a cube with only an albedo texture applied.



*A cube with albedo map applied*

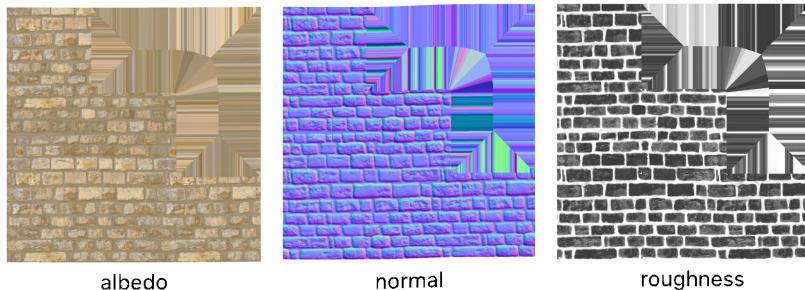
This texture has no lighting information baked into it. Textures altering the surface will change the lighting appropriately.

- In **cube.mtl**, remove the # in front of `map_tangentSpaceNormal cube-normal`.
- Build and run the app again to see the difference when the normal texture is applied.



*A cube with normal and albedo maps applied*

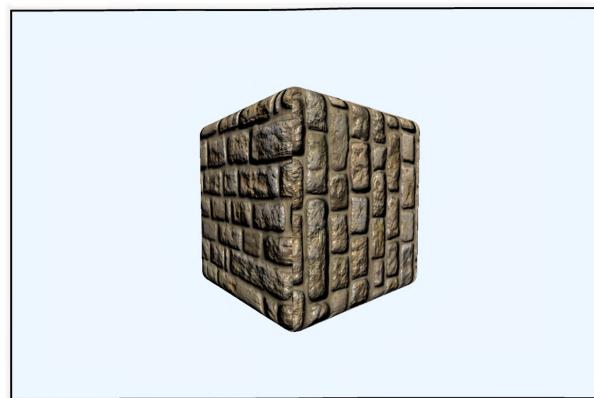
- Still in **cube.mtl**, remove the # in front of `map_roughness cube-roughness`.
- In the **Textures** group, open **Textures.xcassets**, and select **cube-roughness**. Select the image and press the spacebar to preview it. The dark gray values will be smooth and shiny (exaggerated here for effect), and the white mortar between the bricks will be completely rough (not shiny).



*The cube's texture maps*

Compare the roughness map to the cube's color and normal maps to see how the model's UV layout is used for all the textures.

- Build and run the app to see the PBR function in action. Admire how much you can affect how a model looks just by a few textures and a bit of fragment shading.



*The final rendered cube*

## Channel Packing

Later, you'll be using the PBR fragment function for rendering. Even if you don't understand the mathematics, understand the layout of the function and the concepts used.

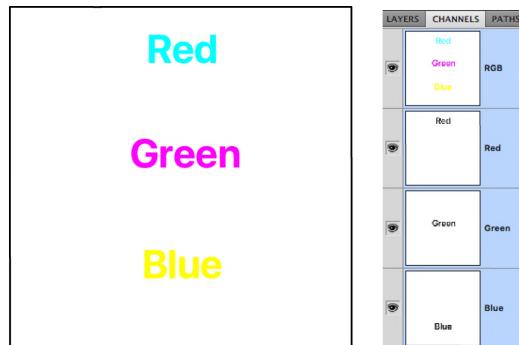
When loading models built by various artists, you're likely going to come up against a variety of standards. Textures may be a different way up; normals might point in a different direction; sometimes you may even find three textures *magically* contained in a single file, a technique known as **channel packing**. Channel packing is an efficient way of managing external textures.

To understand how it works, open **PBR.metal** and look at the code where the fragment function reads single floats: roughness, metallic and ambient occlusion. When the function reads the texture for each of these values, it's only reading the red channel. For example:

```
roughness = roughnessTexture.sample(textureSampler, in.uv).r;
```

Available within the roughness file are green and blue channels that are currently unused. As an example, you could use the green channel for metallic and the blue channel for ambient occlusion.

Included in the resources folder for this chapter is an image named **channel-packed.png**. If you have Photoshop or some other graphics application capable of reading individual channels, open this file and inspect the channels.



*Different channels in Photoshop*

A different color channel contains each of the words. Similarly, you can load your different grayscale maps to each color channel. If you receive a file like this, you can split each channel into a different file by hiding channels and saving the new file. If you're organizing your maps through an asset catalog, channel packing won't impact the memory consumption and you won't gain much advantage. However, some artists *do use it* for easy texture management.

## Challenge

In the resources folder for this chapter is a fabulous helmet model from Malopolska's Virtual Museums collection at [sketchfab.com](https://sketchfab.com). Your challenge is to render this model. There are five textures that you'll load into the asset catalog. Don't forget to change Interpretation from Color to Data, so the textures don't load as sRGB.

Just as you did with the roughness texture, you'll add metallic and ambient occlusion textures to your code. You should also update `TextureIndices` with the correct buffer index numbers.



*Rendering a helmet*

If you get stuck, you'll find the finished project in the challenge folder.

The challenge project can also render USDZ files with textures. When Model I/O loads USDZ files, the textures are loaded as `MDLTextures` instead of string filenames. In the challenge project there is an additional method in `TextureController` to cope with this, as well as extra functionality in `Submesh.Textures`. To load these textures, you also have to preload the asset textures when you load the asset in `Model`, by using `asset.loadTextures()`.

You can download USDZ samples from Apple's AR Quick Look Gallery (<https://apple.co/3qnxbQa>) to try. The animated models, such as the toy robot, still won't work properly until after you've completed the animation chapters, but the static models, such as the car and the teapot, should render with textures once you scale the model down to `0.1`.

## Where to Go From Here?

Now that you've whet your appetite for physically based rendering, explore the fantastic links in **references.markdown**, which you'll find in the resources folder for this chapter. Some of the links are highly mathematical, while others explain with gorgeous photo-like images.

Apple's sample code Using Function Specialization to Build Pipeline Variants (<https://apple.co/3mvBUhM>) is a fantastic piece of sample code to examine, complete with a gorgeous fire truck model. It uses function constants for creating different levels of detail depending on distance from the camera. As a further challenge, you can import the sample's fire truck into your renderer to see how it looks. Remember, though, that you haven't yet implemented great lighting and reflection. In Chapter 21, "Image-Based Lighting", you'll explore how to light your scene with reflection from a skycube texture. Metallic objects look much more realistic when they have something to reflect.

# Chapter 12: Render Passes

Up to this point, you've created projects that had only one render pass. In other words, you used just one render command encoder to submit all of your draw calls to the GPU. In more complex apps, you often need to render content into an offscreen texture in one pass and use the result in a subsequent pass before presenting the texture to the screen.

There are several reasons why you might do multiple passes:

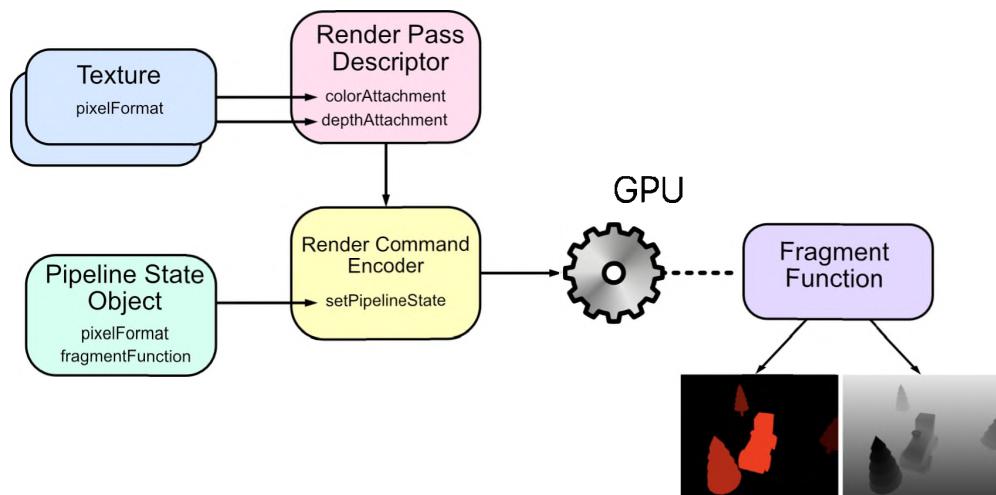
- **Shadows:** In the following chapter, you'll create a shadow pass and render a depth map from a directional light to help calculate shadows in a subsequent pass.
- **Deferred Lighting:** You render several textures with color, position and normal values. Then, in a final pass, you calculate lighting using those textures.
- **Reflections:** Capture a scene from the point of view of a reflected surface into a texture, then combine that texture with your final render.
- **Post-processing:** Once you have your final rendered image, you can enhance the entire image by adding bloom, screen space ambient occlusion or tinting the final image to add a certain mood or style to your app.



# Render Passes

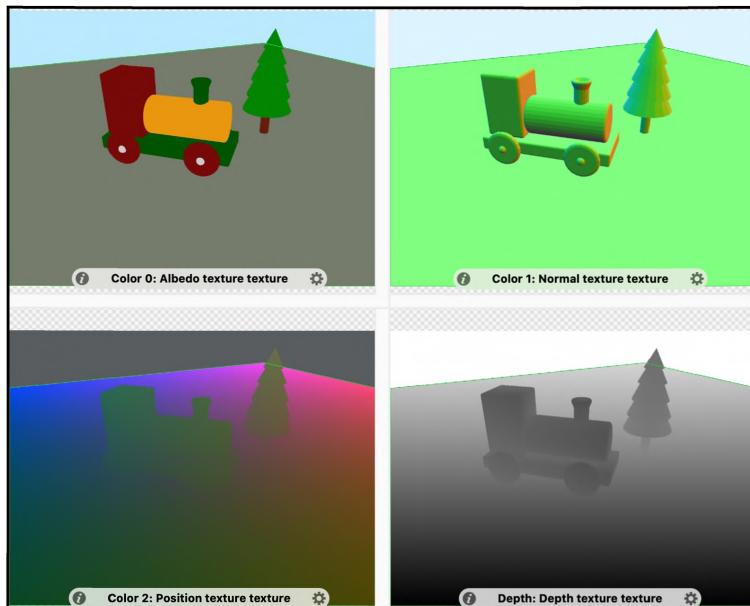
A render pass consists of sending commands to a command encoder. The pass ends when you end encoding on that command encoder.

When setting up a render command encoder, you use a render pass descriptor. So far, you've used the `MTKView currentRenderPassDescriptor`, but you can define your own descriptor or make changes to the current render pass descriptor. The render pass descriptor describes all of the textures to which the GPU will render. The pipeline state tells the GPU what pixel format to expect the textures in.



*A render pass*

For example, the following render pass writes to four textures. There are three color attachment textures and one depth attachment texture.



A render pass with four textures

## Object Picking

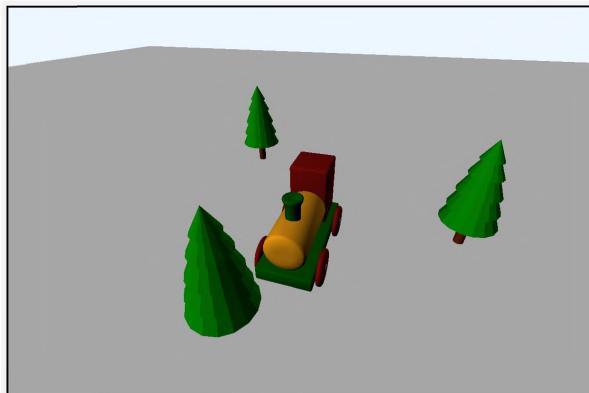
To get started with multipass rendering, you'll create a simple render pass that adds object picking to your app. When you click a model in your scene, that model will render in a slightly different shade.

There are several ways to hit-test rendered objects. For example, you could do the math to convert the 2D touch location to a 3D ray and then perform ray intersection to see which object intersects the ray. Warren Moore describes this method in his [Picking and Hit-Testing in Metal](https://bit.ly/3rlzm9b) (<https://bit.ly/3rlzm9b>) article. Alternatively, you could render a texture where each object is rendered in a different color or object ID. Then, you calculate the texture coordinate from the screen touch location and read the texture to see which object was hit.

You're going to store the model's object ID into a texture in one render pass. You'll then send the touch location to the fragment shader in the second render pass and read the texture from the first pass. If the fragment being rendered is from the selected object, you'll render that fragment in a different color.

## The Starter App

- In Xcode, open the starter app for this chapter and examine the code. It's similar to the previous chapter but refactored.
  - In the **Render Passes** group, **ForwardRenderPass.swift** contains the rendering code that used to be in **Renderer** along with the pipeline state and depth stencil state initialization. Separating this code will make it easier to have multiple render passes because you can then concentrate on getting the pipeline states and textures correct for each pass. In **Renderer**, `draw(scene:in:)` updates the uniforms, then tells the forward render pass to draw the scene.
  - **Pipelines.swift** contains pipeline state creation. Later, **PipelineStates** will contain several more pipeline states.
  - In the **Game** group, **GameScene** sets up new models in a scene.
  - In the **Geometry** group, **Model** now has an `objectId`. When **GameScene** creates the model, it allocates a unique object ID. **Model** updates `params` with its `objectId` for the fragment function.
  - In the **SwiftUI Views** group, **MetalView** has a gesture that forwards the mouse or touch location to **InputController** when the user clicks or taps the screen.
  - In the **Shaders** group, **Vertex.h** now contains **VertexIn** and **VertexOut**. **.metal** files include this header where necessary.
- Build and run the app, and familiarize yourself with the code.



*The starter app*

## Setting up Render Passes

Since you'll have multiple render passes performing similar procedures, it makes sense to have a protocol with some default methods.

- In **Render Passes**, create a new Swift file named **RenderPass.swift**, and replace the code with:

```
import MetalKit

protocol RenderPass {
    var label: String { get }
    var descriptor: MTLRenderPassDescriptor? { get set }
    mutating func resize(view: MTKView, size: CGSize)
    func draw(
        commandBuffer: MTLCommandBuffer,
        scene: GameScene,
        uniforms: Uniforms,
        params: Params
    )
}

extension RenderPass {
```

All render passes will have a render pass descriptor. The pass might create its own descriptor or use the view's current render pass descriptor. They'll all need to resize the render textures when the user resizes the window. All render passes will need a draw method.

The extension will hold default render pass methods.

- Open **ForwardRenderPass.swift**, and conform **ForwardRenderPass** to **RenderPass**:

```
struct ForwardRenderPass: RenderPass {
```

- Cut **buildDepthStencilState()** from **ForwardRenderPass**, and paste it into **RenderPass**'s extension.

Multiple render passes will use this depth stencil state initialization method.

## Creating a UInt32 Texture

Textures don't only hold color. There are many pixel formats (<https://apple.co/3Eby9oD>). So far, you've used `rgba8Unorm`, a color format that contains four 8-bit integers for red, green, blue and alpha.

Model's `objectId` is a `UInt32`, and in place of the model's color, you'll render its ID to a texture. You'll create a texture that holds `UInt32s` in a new render pass.

► In **Render Passes**, create a new Swift file named **ObjectIdRenderPass.swift** and replace the code with:

```
import MetalKit

struct ObjectIdRenderPass: RenderPass {
    let label = "Object ID Render Pass"
    var descriptor: MTLRenderPassDescriptor?
    var pipelineState: MTLRenderPipelineState

    mutating func resize(view: MTKView, size: CGSize) {}

    func draw(
        commandBuffer: MTLCommandBuffer,
        scene: GameScene,
        uniforms: Uniforms,
        params: Params
    ) {
    }
}
```

Here, you create the render pass with the required properties and methods to conform to `RenderPass`, along with a pipeline state object.

► Open **Pipelines.swift**, and add a method to `PipelineStates` to create the pipeline state object:

```
static func createObjectIdPSO() -> MTLRenderPipelineState {
    let pipelineDescriptor = MTLRenderPipelineDescriptor()
    // 1
    let vertexFunction =
        Renderer.library?.makeFunction(name: "vertex_main")
    let fragmentFunction =
        Renderer.library?.makeFunction(name: "fragment_objectId")
    pipelineDescriptor.vertexFunction = vertexFunction
    pipelineDescriptor.fragmentFunction = fragmentFunction
    // 2
    pipelineDescriptor.colorAttachments[0].pixelFormat = .r32UInt
    // 3
```



```
    pipelineDescriptor.depthAttachmentPixelFormat = .invalid
    pipelineDescriptor.vertexDescriptor =
        MTLVertexDescriptor.defaultLayout
    return Self.createPSO(descriptor: pipelineDescriptor)
}
```

Most of this code will be familiar to you, but there are some details to note:

1. You can use the same vertex function as you did to render the model because you'll render the vertices in the same position. However, you'll need a different fragment function to write the ID to the texture.
2. The color attachment's texture pixel format is a 32-bit unsigned integer. The GPU will expect you to hand it a texture in this format.
3. You'll come back and add a depth attachment, but for now, leave it invalid, which means that the GPU won't require a depth texture.

► Open **ObjectIdRenderPass.swift**, and create an initializer:

```
init() {
    pipelineState = PipelineStates.createObjectIdPSO()
    descriptor = MTLRenderPassDescriptor()
}
```

Here, you initialize the pipeline state and the render pass descriptor.

Most render passes will require you to create a texture, so you'll create one that takes several different parameters.

► Open **RenderPass.swift**, and add a new method to the extension:

```
static func makeTexture(
    size: CGSize,
    pixelFormat: MTLPixelFormat,
    label: String,
    storageMode: MTLStorageMode = .private,
    usage: MTLTextureUsage = [.shaderRead, .renderTarget]
) -> MTLTexture? {
```

In addition to a size, you'll give the texture:

- A pixel format, such as `rgba8Unorm`. In this render pass, you give it `r32UInt`.
- By default, the storage mode is `private`, meaning the texture stores in memory in a place that only the GPU can access.



- The usage. You have to configure textures used by render pass descriptors as **render targets**. Render targets are memory buffers or textures that allow offscreen rendering for cases where the rendered pixels don't need to end up in the framebuffer. You'll also want to read the texture in shader functions, so you set up that default capability, too.

► Add this code to

```
makeTexture(size:pixelFormat:label:storageMode:usage:):
```

```
let width = Int(size.width)
let height = Int(size.height)
guard width > 0 && height > 0 else { return nil }
let textureDesc =
    MTLTextureDescriptor.texture2DDescriptor(
        pixelFormat: pixelFormat,
        width: width,
        height: height,
        mipmapped: false)
textureDesc.storageMode = storageMode
textureDesc.usage = usage
guard let texture =
    Renderer.device.makeTexture(descriptor: textureDesc) else {
    fatalError("Failed to create texture")
}
texture.label = label
return texture
```

You configure a texture descriptor using the given parameters and create a texture from the descriptor.

► Open **ObjectIdRenderPass.swift**, and add a new property to **ObjectIdRenderPass** for the render texture:

```
var idTexture: MTLTexture?
```

► Add this code to **resize(view:size:)**:

```
idTexture = Self.makeTexture(
    size: size,
    pixelFormat: .r32UInt,
    label: "ID Texture")
```

Every time the view size changes, you'll rebuild the texture to match the view's size.

Now for the draw.

- Add this code to `draw(commandBuffer:scene:uniforms:params:)`:

```
guard let descriptor = descriptor else {
    return
}
descriptor.colorAttachments[0].texture = idTexture
guard let renderEncoder =
    commandBuffer.makeRenderCommandEncoder(descriptor: descriptor)
else { return }
```

You assign `idTexture` to the descriptor's first color attachment. You then create the render command encoder using this descriptor. The pixel format must match the render target textures when configuring the color attachment for the pipeline state object. In this case, you set them both to `r32UInt`.

- Add this code after the code you just added:

```
renderEncoder.label = label
renderEncoder.setRenderPipelineState(pipelineState)
for model in scene.models {
    model.render(
        encoder: renderEncoder,
        uniforms: uniforms,
        params: params)
}
renderEncoder.endEncoding()
```

Here, you set the pipeline state and render the models.

## Adding the Render Pass to Renderer

- Open `Renderer.swift`, and add the new render pass property:

```
var objectIdRenderPass: ObjectIdRenderPass
```

- In `init(metalView:options:)`, add this code before `super.init()` to initialize the render pass:

```
objectIdRenderPass = ObjectIdRenderPass()
```

- Add this code to `mtkView(_:drawableSizeWillChange:)`:

```
objectIdRenderPass.resize(view: view, size: size)
```

Here, you ensure that `idTexture`'s size matches the view size. Renderer's initializer calls `mtkView(_:drawableSizeWillChange)`, so your texture in the render pass is initialized and sized appropriately.

- Add this code to `draw(scene:in:)` immediately after `updateUniforms(scene:scene:)`:

```
objectIdRenderPass.draw(  
    commandBuffer: commandBuffer,  
    scene: scene,  
    uniforms: uniforms,  
    params: params)
```

Excellent, you've set up the render pass. Now all you have to do is create the fragment shader function to write to `idTexture`.

## Adding the Shader Function

The Object ID render pass will write the currently rendered model's object ID to a texture. You don't need any of the vertex information in the fragment function.

- In **Shaders**, create a new Metal File named **ObjectId.metal** and add:

```
#import "Common.h"  
  
// 1  
struct FragmentOut {  
    uint objectId [[color(0)]];  
};  
  
// 2  
fragment FragmentOut fragment_objectId(  
    constant Params &params [[buffer(ParamsBuffer)]])  
{  
    // 3  
    FragmentOut out {  
        .objectId = params.objectId  
    };  
    return out;  
}
```

Going through this code:

1. You create a structure that matches the render pass descriptor color attachment. Color attachment 0 contains the object ID texture.
  2. The fragment function takes in `params`, of which you only need the object ID.
  3. You create a `FragmentOut` instance and write the current object ID to it. You then return it from the fragment function, and the GPU writes the fragment into the given texture.
- Build and run the app.

You won't see a difference in your render. Currently, you're not passing on the object ID texture to the second render pass.



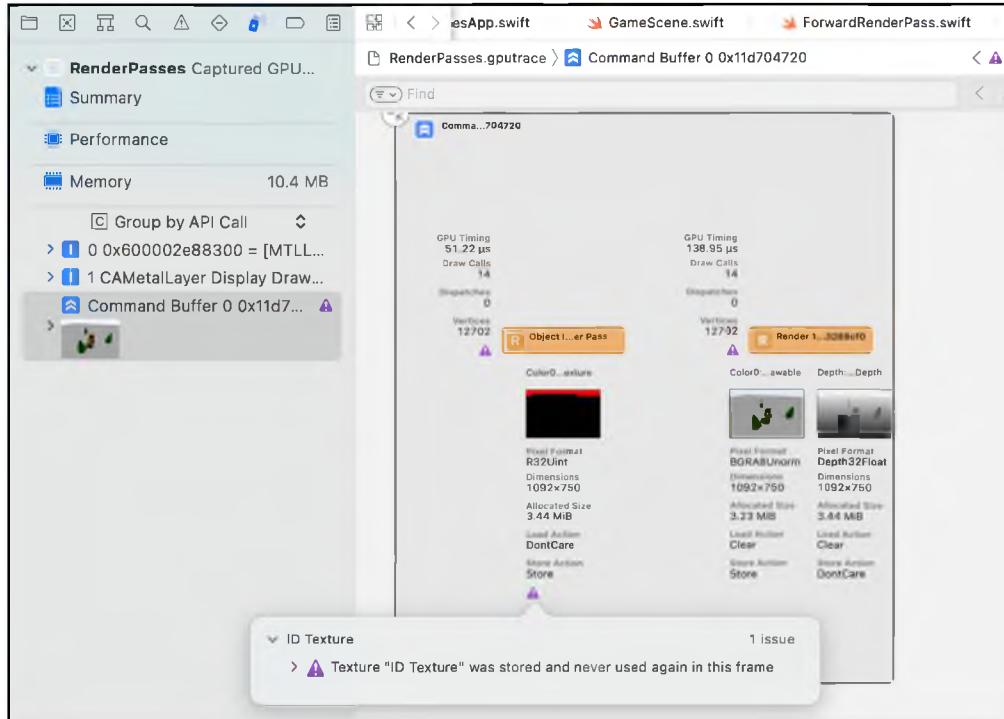
*No difference to the render*

- Capture the GPU workload by clicking the Metal icon and clicking **Capture** in the popup.



*The GPU workload capture icon*

- Click the command buffer, and you'll see two render passes. The Object ID render pass is on the left with an R32UInt pixel format texture. The usual forward render pass is on the right and has a color texture and a depth texture.

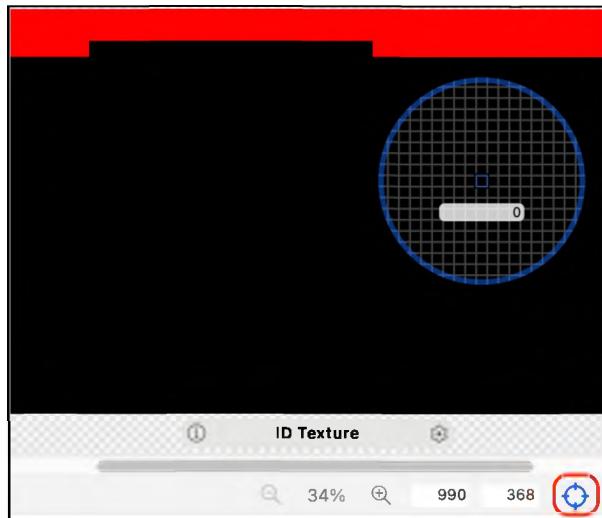


*The GPU workload capture*

Even though the store action of the Object ID's texture is **Store**, you aren't using it in the following render pass. This causes a purple exclamation mark warning.

- Double-click the red and black texture. This is **idTexture**.

Under the texture, click the target icon to show a magnifier, and move it around the texture. As you drag the magnifier over objects, it shows the value of each fragment. This should show you the object ID, but almost all of it shows an object ID of zero. The ground, which has an object ID of zero, renders on top of the other objects.



*ID texture with erroneous object ID*

To get the correct object ID, it's important to discard models' fragments that are behind other models. For this reason, you'll need to render with a depth texture.

## Adding the Depth Attachment

► Open **ObjectIdRenderPass.swift**, and add a new property to **ObjectIdRenderPass**:

```
var depthTexture: MTLTexture?
```

So far, you've used the current drawable's default depth texture. Next, you'll create a depth texture that you'll maintain.

- Add this code to `resize(view:size:)`:

```
depthTexture = Self.makeTexture(  
    size: size,  
    pixelFormat: .depth32Float,  
    label: "ID Depth Texture")
```

Here, you create the depth texture with the correct size and pixel format. This pixel format must match the render pipeline state depth texture format.

- Open `Pipelines.swift`, and in `createObjectIdPSO()`, change `pipelineDescriptor.depthAttachmentPixelFormat = .invalid` to:

```
pipelineDescriptor.depthAttachmentPixelFormat = .depth32Float
```

Now the pixel formats will match.

- Go back to `ObjectIdRenderPass.swift`. In `draw(commandBuffer:scene:uniforms:params:)`, after setting the color attachment texture, add:

```
descriptor.depthAttachment.texture = depthTexture
```

You created and stored a depth texture. If you were to build and run now and capture the GPU workload, you'd see the depth texture, but you haven't completed setting up the GPU's depth rendering yet.

## The Depth Stencil State

- Create a new property in `ObjectIdRenderPass`:

```
var depthStencilState: MTLDepthStencilState?
```

- Add this code to the end of `init()`:

```
depthStencilState = Self.buildDepthStencilState()
```

You set up a depth stencil state object with the usual depth rendering.

- In `draw(commandBuffer:scene:uniforms:params:)`, add the following code after setting the render pipeline state:

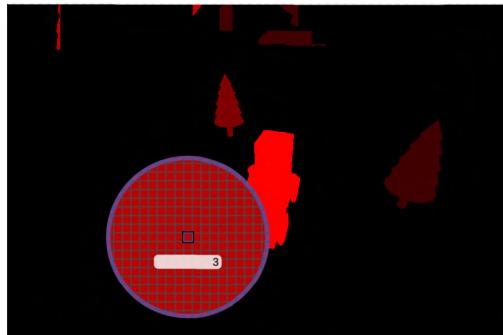
```
renderEncoder.setDepthStencilState(depthStencilState)
```



Here, you let the GPU know about the depth setting you want to render with.

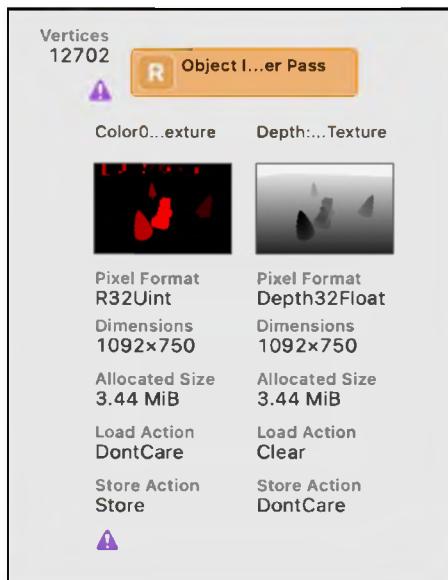
- Build and run the app. Capture the GPU workload and take a look at your color texture now.

Now when you run the magnifier over each object, you'll clearly see the object IDs.



*ID texture with Object IDs*

You may see some random pixels at the top of the render. The render pass load action for the texture is `dontCare`, so wherever you're not rendering an object, the pixels will be random. You'll need to clear the texture before you render to know exactly what object ID is in the area you click to select.



*Load and store actions*

**Note:** In reality, it doesn't matter whether you clear on load in this example. As you'll see shortly, the change of color of each fragment on a picked object will only occur during the fragment function. Since the non-rendered pixels at the top of the screen aren't being processed through a fragment function, a change of color will never happen. However, it's good practice to know what's happening in your textures. At some point, you might decide to pass back the texture to the CPU for further processing.

## Load & Store Actions

You set up load and store actions in the render pass descriptor attachments. Only set the store action to store if you need the attachment texture down the line.

Additionally, you should only clear the texture if you need to. If your fragment function writes to every fragment that appears on-screen, you generally don't need to clear. For example, you don't need to clear if you render a full-screen quad.

► Open **ObjectIdRenderPass.swift**. In

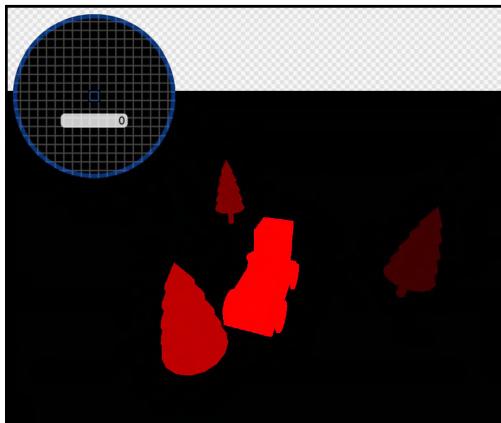
```
draw(commandBuffer:scene:uniforms:params:), after setting  
descriptor.colorAttachments[0].texture, add:
```

```
descriptor.colorAttachments[0].loadAction = .clear  
descriptor.colorAttachments[0].storeAction = .store
```

The load action can be `clear`, `load` or `dontCare`. The most common store actions are `store` or `dontCare`.



- Build and run the app, and capture the GPU workload again. Recheck the object ID texture.



*No random pixels*

The pixels at the top of the screen are now cleared with zeros. If you want a non-zero clear value, set `colorAttachments[0]. clearColor`.

## Reading the Object ID Texture

You now have a choice. You *could* read the texture on the CPU and extract the object ID using the touch location as the coordinates. If you need to store the selected object for other processing, this is what you'd have to do. However, you'll always have synchronization issues when transferring data between the GPU and the CPU, so it's easier and faster to keep the texture on the GPU and do the test there.

- Open **ForwardRenderPass.swift**, and add this new property to `ForwardRenderPass`:

```
weak var idTexture: MTLTexture?
```

`idTexture` will hold the ID texture from the object ID render pass.

- Open **Renderer.swift**. In `draw(scene:in:)`, add this code after `objectIdRenderPass.draw(...):`

```
forwardRenderPass.idTexture = objectIdRenderPass.idTexture
```

You pass the ID texture from one render pass to the next.

- Open **ForwardRenderPass.swift**. In `draw(commandBuffer:scene:uniforms:params:)`, before the `for` render loop, add:

```
renderEncoder.setFragmentTexture(idTexture, index: 11)
```

You pass `idTexture` to the forward render pass's fragment function. Be careful with your index numbers. You may want to rename this one as you did with earlier indices.

You'll also need to send the touch location to the fragment shader so you can use it to read the ID texture.

- After the previous code, add:

```
let input = InputController.shared
var params = params
params.touchX = UInt32(input.touchLocation?.x ?? 0)
params.touchY = UInt32(input.touchLocation?.y ?? 0)
```

`input.touchLocation` is the last location touched on the metal view. The SwiftUI gesture updates it on `MetalView`.

- Open **PBR.metal**. This is the complex PBR shader you used in the previous chapter.

- Add this code to the parameters of `fragment_PBR`:

```
texture2d<uint> idTexture [[texture(11)]]
```

Be mindful of the type of texture you pass and the index number.

- After the conditional that sets `material.baseColor`, add:

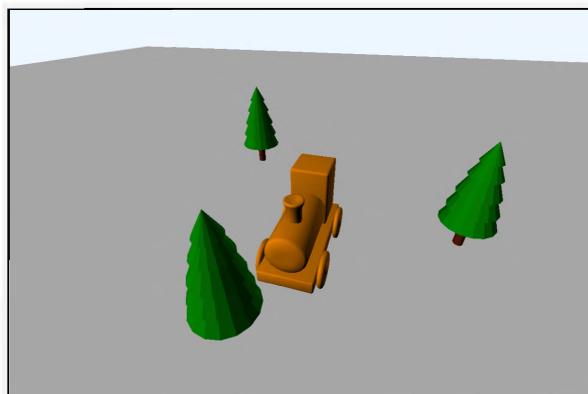
```
if (!is_null_texture(idTexture)) {
    uint2 coord = uint2(params.touchX * 2, params.touchY * 2);
    uint objectID = idTexture.read(coord).r;
    if (params.objectId != 0 && objectID == params.objectId) {
        material.baseColor = float3(0.9, 0.5, 0);
    }
}
```

Here, you read `idTexture` using the passed-in touch coordinates. This code works because you ensured that `idTexture` is the same size as the screen. Sometimes it's worthwhile to halve the texture size to save resources. You could certainly do that here, as long as you remember to halve the coordinates when reading the texture in the fragment function.

Notice that `read` differs from `sample`. `read` uses pixel coordinates rather than normalized coordinates. You don't need a sampler to read a texture, but you also can't use the various sampler options when you use `read`.

If the currently rendered object ID isn't zero and the object ID matches the fragment in `idTexture`, change the material's base color to orange.

► Build and run the app, and click any of the objects.

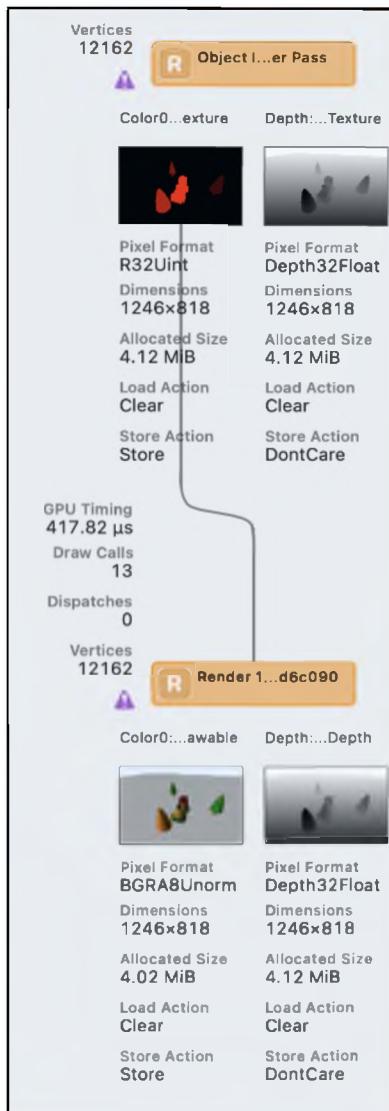


*Selected train turns orange*

The object you picked will turn orange. When you click the sky or the ground, none of the objects are picked.

This is an easy way to test whether an object is picked. It's also a good way to learn simple render pass texture chaining. However, in most circumstances, you'll need to pass back the texture to the CPU, so it's more efficient to perform ray picking as described at the beginning of the chapter.

- With the app running, capture the GPU workload, and click the command buffer.



The frame graph reflects that you're now using `idTexture` in your PBR fragment function. The GPU points out any missing textures and redundant binding errors where you bind an already bound texture.

Now that you know how to render textures in different render passes, you can move on to more complex rendering and add some shadows in the next chapter.

## Key Points

- A **render pass descriptor** describes all of the textures and load and store actions needed by a render pass.
- Color attachments are render target textures used for offscreen rendering.
- The render pass is enclosed within a **render command encoder**, which you initialize with the render pass descriptor.
- You set a **pipeline state object** on the render command encoder. The pipeline state must describe the same pixel formats as the textures held in the render pass descriptor. If there is no texture, the pixel format must be `invalid`.
- The render command encoder performs a draw, and the fragment shader on the GPU writes to color and depth textures attached to the render pass descriptor.
- Color attachments don't have to be `rgb` colors. Instead, you can write `uint` or `float` values in the fragment function.
- For each texture, you describe load and store actions. If you aren't using a texture in a later render pass, the action should be `dontCare` so the GPU can discard it and free up memory.
- The **GPU workload capture** shows you a frame graph where you can see how all your render passes chain together.

# Chapter 13: Shadows

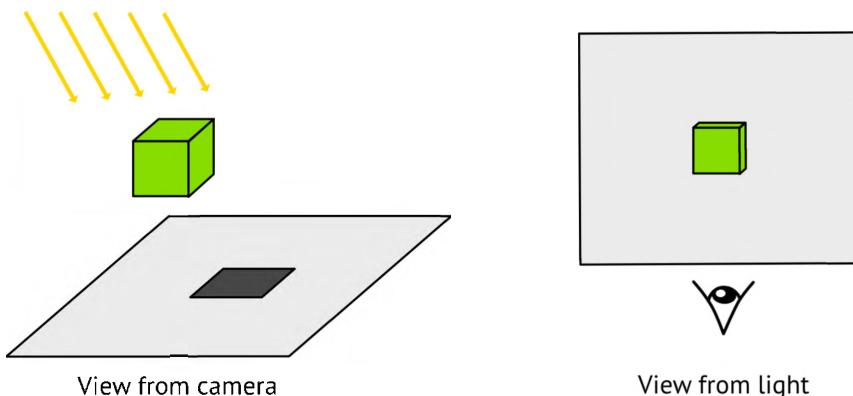
In this chapter, you'll learn about shadows. A shadow represents the absence of light on a surface. You see shadows on an object when another surface or object obscures it from light. Adding shadows in a project makes your scene look more realistic and provides a feeling of depth.



# Shadow Maps

**Shadow maps** are textures containing a scene's shadow information. When light shines on an object, it casts a shadow on anything behind it.

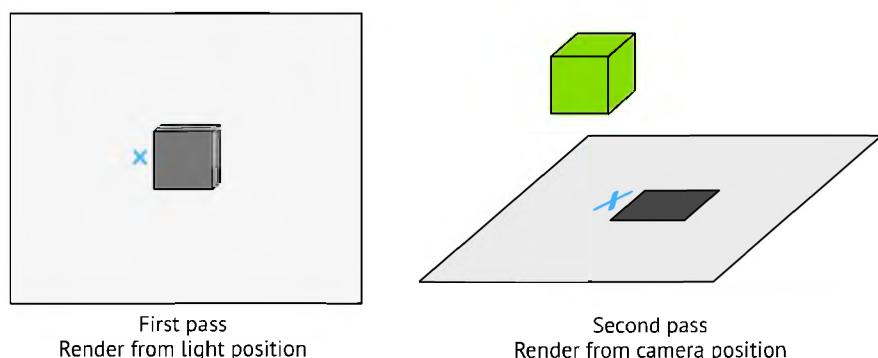
Typically, you render the scene from your camera's location. However, to build a shadow map, you need to render your scene from the light source's location - in this case, the sun.



*A scene render*

The image on the left shows a render from the camera's position with the directional light pointing down. The image on the right shows a render from the directional light's position. The eye shows the camera's position in the first image.

You'll do two render passes:



*Two render passes are needed*

- **First pass:** You'll render from the light's point of view. Since the sun is directional, you'll use an orthographic camera rather than a perspective camera. You're only interested in the depth of objects that the sun can see, so you won't render a color texture. In this pass, you'll only render the shadow map as a depth texture. This is a grayscale texture, with the gray value indicating depth. Black is close to the light, and white is farther away.
- **Second pass:** You'll render using the scene camera as usual, but you'll compare the camera fragment with each shadow map fragment. If the camera fragment's depth is less than the shadow map fragment at that position, the fragment is in the shadow. The light can see the blue x in the above image, so it isn't in shadow.

Why would you need two passes here? In this case, you'll render the shadow map from the light's position, not from the camera's position. You'll save the output to a shadow texture and give it to the next render pass, which combines the shadow with the rest of the scene to make a final image.

## The Starter Project

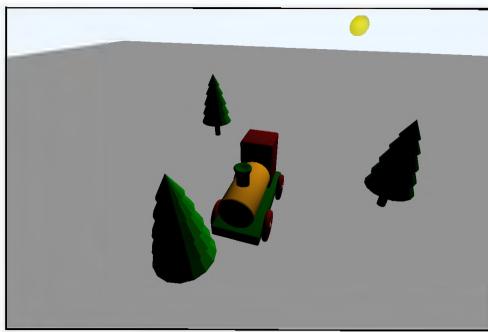
- In Xcode, open this chapter's starter project.

The code in the starter project is almost identical to the previous chapter but without the object ID and picking code. The scene now has a visible sun that rotates around the scene's center as the only light.

The code for rendering the sun is in the **Utility** group in **DebugModel.swift**. You'll render the sun separately from the scene so that the sun model, itself, isn't shaded with the rest of the scene.

There are a few extra files containing code the app doesn't need yet. You'll learn about these files as you proceed through the chapter.

- Build and run the app.



*The starter app*

Without shadows, the train and trees in this render appear to float above the ground.

The process for adding the new shadow pass is similar to adding the previous chapter's object picking render pass:

1. Create the new render pass structure and configure the render pass descriptor, depth stencil state and pipeline state.
2. Declare and draw the render pass in Renderer.
3. Set up the drawing code in the render pass.
4. Set up an orthographic camera from the light's position and calculate the necessary matrices.
5. Create the vertex shader function to draw vertices from the light's position.

Although you give the sun a position in this app, as you learned in Chapter 10, “Lighting Fundamentals”, a directional light has a direction rather than a position. So here, you'll use the sun's position as a direction.

**Note:** If you want to see directional lines to debug the sun's direction, as you did in the earlier chapter, add `DebugLights.draw(lights: scene.lighting.lights, encoder: renderEncoder, uniforms: uniforms)` to `ForwardRenderPass` before `renderEncoder.endEncoding()`.

Time to add the new shadow pass.

## 1. Creating the New Render Pass

- In the **Render Passes** group, create a new Swift file named **ShadowRenderPass.swift**, and replace the code with:

```
import MetalKit

struct ShadowRenderPass: RenderPass {
    let label: String = "Shadow Render Pass"
    var descriptor: MTLRenderPassDescriptor?
        = MTLRenderPassDescriptor()
    var depthStencilState: MTLDepthStencilState?
        = Self.buildDepthStencilState()
    var pipelineState: MTLRenderPipelineState
    var shadowTexture: MTLTexture?

    mutating func resize(view: MTKView, size: CGSize) {}

    func draw(
        commandBuffer: MTLCommandBuffer,
        scene: GameScene,
        uniforms: Uniforms,
        params: Params
    ) {
    }
}
```

This code creates a render pass that conforms to `RenderPass`, with a pipeline state and a texture property for the shadow map.

- Open **Pipelines.swift**, and create a new method to create the pipeline state object:

```
static func createShadowPSO() -> MTLRenderPipelineState {
    let vertexFunction =
        Renderer.library?.makeFunction(name: "vertex_depth")
    let pipelineDescriptor = MTLRenderPipelineDescriptor()
    pipelineDescriptor.vertexFunction = vertexFunction
    pipelineDescriptor.colorAttachments[0].pixelFormat = .invalid
    pipelineDescriptor.depthAttachmentPixelFormat = .depth32Float
    pipelineDescriptor.vertexDescriptor = .defaultLayout
    return createPSO(descriptor: pipelineDescriptor)
}
```

Here, you create a pipeline state without a color attachment or fragment function. You’re only interested in the depth information for the shadow — not the color information — so you set the color attachment pixel format to `invalid`. You’ll still render models, so you still need to hold the vertex descriptor and transform all of the models’ vertices in a vertex function.

- Open `ShadowRenderPass.swift`, and create the initializer:

```
init() {
    pipelineState =
        PipelineStates.createShadowPS0()
    shadowTexture = Self.makeTexture(
        size: CGSize(
            width: 2048,
            height: 2048),
        pixelFormat: .depth32Float,
        label: "Shadow Depth Texture")
}
```

This code initializes the pipeline state object and builds the depth texture with the required pixel format. Unlike other render passes, where you match the view’s size, shadow maps are usually square to match the light’s cuboid orthographic camera, so you don’t need to resize the texture when the window resizes. The resolution should be as much as your game resources budget allows to produce sharper shadows.

## 2. Declaring and Drawing the Render Pass

- In the `Game` group, open `Renderer.swift`, and add the new render pass property to `Renderer`:

```
var shadowRenderPass: ShadowRenderPass
```

- In `init(metalView:options:)`, initialize the render pass before calling `super.init()`:

```
shadowRenderPass = ShadowRenderPass()
```

- In `mtkView(_:drawableSizeWillChange:)`, add:

```
shadowRenderPass.resize(view: view, size: size)
```

At the moment, you’re not resizing any textures in `shadowRenderPass`, but since you’ve already created `resize(view:size:)` as a required method to conform to `RenderPass`, and you may add textures later, you should call the method here.



- In `draw(scene:in:)`, after `updateUniforms(scene: scene)`, add:

```
shadowRenderPass.draw(  
    commandBuffer: commandBuffer,  
    scene: scene,  
    uniforms: uniforms,  
    params: params)
```

This code performs the render pass.

### 3. Setting up the Render Pass Drawing Code

- Open `ShadowRenderPass.swift`, and add the following code to `draw(commandBuffer:scene:uniforms:params:)`:

```
guard let descriptor = descriptor else { return }  
descriptor.depthAttachment.texture = shadowTexture  
descriptor.depthAttachment.loadAction = .clear  
descriptor.depthAttachment.storeAction = .store  
  
guard let renderEncoder =  
    commandBuffer.makeRenderCommandEncoder(descriptor: descriptor)  
else {  
    return  
}  
renderEncoder.label = "Shadow Encoder"  
renderEncoder.setDepthStencilState(depthStencilState)  
renderEncoder.setRenderPipelineState(pipelineState)  
for model in scene.models {  
    renderEncoder.pushDebugGroup(model.name)  
    model.render(  
        encoder: renderEncoder,  
        uniforms: uniforms,  
        params: params)  
    renderEncoder.popDebugGroup()  
}  
renderEncoder.endEncoding()
```

Here, you set the depth attachment texture on the descriptor's depth attachment. The GPU will clear the texture on loading it and store it so that the following render pass can use it. You then create the render command encoder, using the descriptor, and render the scene as usual.

You surround the model render with `renderEncoder.pushDebugGroup(_:)` and `renderEncoder.popDebugGroup()`, which gathers the render commands into groups on the GPU workload capture. Now, you can more easily debug what's happening.

## 4. Setting up the Light Camera

During the shadow pass, you'll render from the point of view of the sun, so you'll need a new camera and some new shader matrices.

- In **Common.h**, in the **Shaders** group, add these properties to **Uniforms**:

```
matrix_float4x4 shadowProjectionMatrix;
matrix_float4x4 shadowViewMatrix;
```

Here, you hold the projection and view matrices for the sunlight.

- Open **Renderer.swift**, and add a new property to **Renderer**:

```
var shadowCamera = OrthographicCamera()
```

Here, you create an orthographic camera. You previously used an orthographic camera in Chapter 9, “Navigating a 3D Scene”, to render the scene from above. Because sunlight is a directional light, this is the correct projection type for shadows caused by sunlight. For example, if you want shadows from a spotlight, you would use a perspective camera with a field of view that matches the spotlight's cone angle.

- Add the following code to the end of **updateUniforms(scene:)**:

```
shadowCamera.viewSize = 16
shadowCamera.far = 16
let sun = scene.lighting.lights[0]
shadowCamera.position = sun.position
```

With this code, you set up the orthographic camera with a cubic view volume of 16 units.

- Continue with some more code:

```
uniforms.shadowProjectionMatrix = shadowCamera.projectionMatrix
uniforms.shadowViewMatrix = float4x4(
    eye: sun.position,
    center: .zero,
    up: [0, 1, 0])
```

`shadowViewMatrix` is a `lookAt` matrix that ensures the sun is *looking at* the center of the scene. `float4x4(eye:center:up)` is defined in `MathLibrary.swift`. It takes the camera's position, the point that the camera should look at, and the camera's up vector. This matrix rotates the camera to look at the target by providing these parameters.

**Note:** Here's a useful debugging tip. Temporarily set `uniforms.viewMatrix` to `uniforms.shadowViewMatrix` and `uniforms.projectionMatrix` to `uniforms.shadowProjectionMatrix` at the end of `updateUniforms(scene:)`. Developers commonly get the shadow matrices wrong, and it's useful to visualize the scene render through the light.

## 5. Creating the Shader Function

As you may have noticed when you set up the shadow pipeline state object in `Pipelines.swift`, it references a shader function named `vertex_depth`, which doesn't exist yet.

- In the **Shaders** group, using the **Metal File** template, create a new file named `Shadow.metal`. Make sure to check both **macOS** and **iOS** targets.
- Add the following code to the new file:

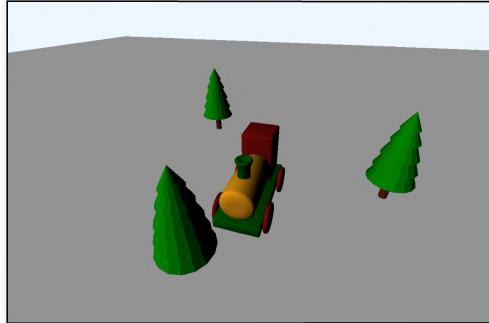
```
#import "Common.h"

struct VertexIn {
    float4 position [[attribute(0)]];
};

vertex float4
vertex_depth(const VertexIn in [[stage_in]],
             constant Uniforms &uniforms [[buffer(UniformsBuffer)]])
{
    matrix float4x4 mvp =
        uniforms.shadowProjectionMatrix * uniforms.shadowViewMatrix
        * uniforms.modelMatrix;
    return mvp * in.position;
}
```

This code receives a vertex position, transforms it by the light's projection and view matrices that you set up in `Renderer` and returns the transformed position.

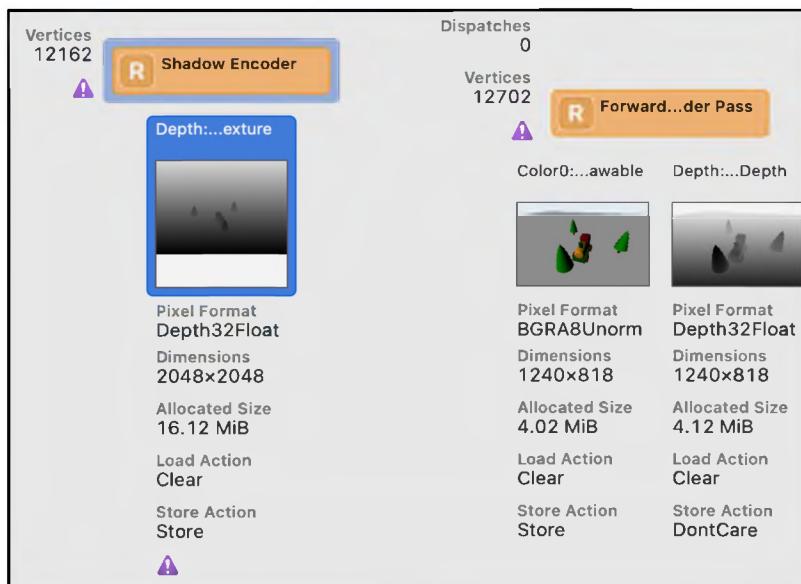
- Build and run the app.



*No shadow yet*

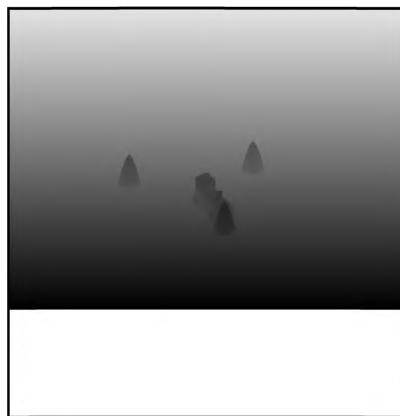
That looks nice, but where's the shadow?

- Capture the GPU workload and examine the frame capture.



*GPU frame capture*

- Double-click the shadow encoder pass texture result to show the texture in the other resource pane.



*The shadow pass depth texture*

This is the scene rendered from the light's position. You used the shadow pipeline state, which you configured not to have a fragment shader, so the color information isn't processed here at all — it's purely depth. Lighter colors are farther away, and darker colors are closer.

## The Main Pass

Now that you have the shadow map saved to a texture, you just need to send it to the main pass to use the texture in lighting calculations in the fragment function.

- Open **ForwardRenderPass.swift**, and add a new property:

```
weak var shadowTexture: MTLTexture?
```

- In `draw(commandBuffer:scene:uniforms:params:)`, before the model render for loop, add:

```
renderEncoder.setFragmentTexture(shadowTexture, index: 15)
```

You pass in the shadow texture and send it to the GPU.

- Open **Renderer.swift**, and add this code to `draw(scene:in:)` before drawing the forward render pass:

```
forwardRenderPass.shadowTexture = shadowRenderPass.shadowTexture
```

You pass the shadow texture from the previous shadow pass to the forward render pass.

- In the **Shaders** group, open **Vertex.h**, and add a new member to **VertexOut**:

```
float4 shadowPosition;
```

This holds the vertex position transformed by the shadow matrices.

- Open **Shaders.metal**, and add this line in **vertex\_main**, when creating **out**:

```
.shadowPosition =  
    uniforms.shadowProjectionMatrix * uniforms.shadowViewMatrix  
    * uniforms.modelMatrix * in.position
```

You hold two transformed positions for each vertex. One transformed within the scene from the camera's point of view, and the other from the light's point of view. You'll be able to compare the shadow position with the fragment from the shadow map.

- Open **PBR.metal**.

This file is where the lighting happens, so the rest of the shadow work will occur in **fragment\_PBR**.

- First, add one more function parameter after **aoTexture**:

```
depth2d<float> shadowTexture [[texture(15)]]
```

Unlike the textures you've used in the past, which have a type of **texture2d**, the texture type of a depth texture is **depth2d**.

- At the end of **fragment\_PBR**, before **return**, add:

```
// shadow calculation  
// 1  
float3 shadowPosition  
    = in.shadowPosition.xyz / in.shadowPosition.w;  
// 2  
float2 xy = shadowPosition.xy;  
xy = xy * 0.5 + 0.5;  
xy.y = 1 - xy.y;  
xy = saturate(xy);  
// 3  
constexpr sampler s(  
    coord::normalized, filter::linear,  
    address::clamp_to_edge,  
    compare_func::less);
```



```
float shadow_sample = shadowTexture.sample(s, xy);
// 4
if (shadowPosition.z > shadow_sample) {
    diffuseColor *= 0.5;
}
```

Here's a code breakdown:

1. `in.shadowPosition` represents the vertex's position from the light's point of view. The GPU performed a perspective divide before writing the fragment to the shadow texture when you rendered from the light's point of view. Dividing `xyz` by `w` here matches the same perspective division so that you can compare the current sample's depth value to the one in the shadow texture.
  2. Determine a coordinate pair from the shadow position to serve as a screen space pixel locator on the shadow texture. Then, you rescale the coordinates from  $[-1, 1]$  to  $[0, 1]$  to match the uv space. Finally, you reverse the Y coordinate since it's upside down.
  3. Create a sampler to use with the shadow texture, and sample the texture at the coordinates you just created. Get the depth value for the currently processed pixel. You create a new sampler, as `textureSampler`, initialized at the top of the function, repeats the texture if it's sampled off the edge. Try using `textureSampler` later to see repeated extra shadows at the back of the scene.
  4. You darken the diffuse color for pixels with a depth greater than the shadow value stored in the texture. For example, if `shadowPosition.z` is  $0.5$ , and `shadow_sample` from the stored depth texture is  $0.2$ , then from the sun's point of view, the current fragment is further away than the stored fragment. Since the sun can't see the fragment, it's in shadow.
- Build and run the app, and you'll finally see models with shadows.



*Shadows added*

## Shadow Acne

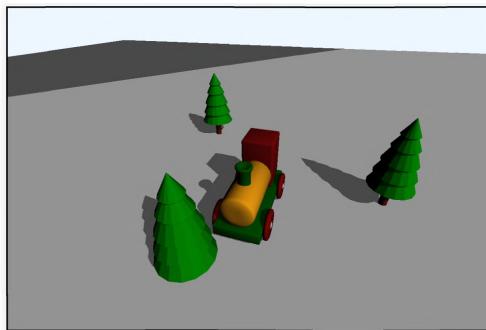
In the previous image, as the sun rotates, you'll notice a lot of flickering. This is called **shadow acne** or **surface acne**. The surface is self-shadowing because of a lack of float precision where the sampled texel doesn't match the calculated value.

You can mitigate this by adding a bias to the shadow texture, increasing the z value, thereby bringing the stored fragment closer.

- Change the conditional test at `// 4` above to:

```
if (shadowPosition.z > shadow_sample + 0.001) {
```

- Build and run the app.



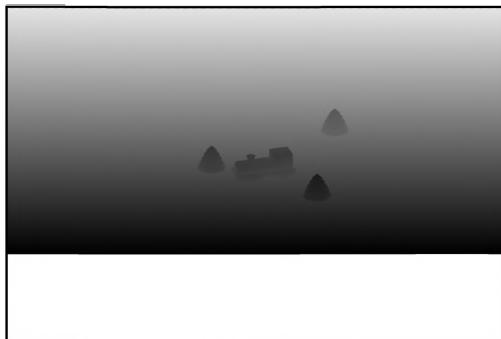
*Shadows with no acne*

The surface acne is now gone, and you have clear shadows from the sun as it rotates around the scene.

## Identifying Problems

Take a look at the previous render, and you'll see a problem. Actually, there are two problems. A large dark gray area on the plane appears to be in shadow but shouldn't be.

If you capture the scene, this is the depth texture for that sun position:



*Orthographic camera too large*

The bottom quarter of the image is white, meaning that the depth for that position is at its farthest. The light's orthographic camera cuts off that part of the plane and causes it to look as if it is in shadow.

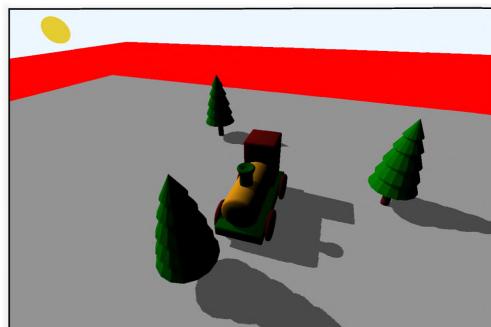
The second problem is reading the shadow map.

► Open **PBR.metal**. In `fragment_PBR`, before `xy = saturate(xy);`, add:

```
if (xy.x < 0.0 || xy.x > 1.0 || xy.y < 0.0 || xy.y > 1.0) {  
    return float4(1, 0, 0, 1);  
}
```

The `xy` texture coordinates should go from `0` to `1` to be on the texture. So if the coordinates are off the texture, you return red.

► Build and run the app.



*Reading values off the texture*

Areas in red are off the depth texture.

You can solve these two problems by setting up the light's orthographic camera to enclose everything the scene camera catches.

## Visualizing the Problems

In the **Utility** group, **DebugCameraFrustum.swift** will help you visualize this problem by rendering wireframes for the various camera frustums. When running the app, you can press various keys for debugging purposes:

- 1: The front view of the scene.
- 2: The default view where the sun rotates around the scene.
- 3: Render a wireframe of the scene camera frustum.
- 4: Render a wireframe of the light camera frustum.
- 5: Render a wireframe of the scene camera's bounding sphere.

This key code is in **GameScene**'s `update(deltaTime:)`.

► Open **ForwardRenderPass.swift**, and add this to the end of `draw(commandBuffer:scene:uniforms:params:)`, before `renderEncoder.endEncoding()`:

```
DebugCameraFrustum.draw(  
    encoder: renderEncoder,  
    scene: scene,  
    uniforms: uniforms)
```

This code sets up the debug code so that the above keypresses will work.

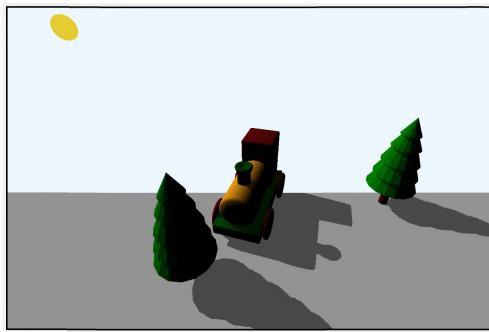
► Open **GameScene.swift**, and at the top of `init()`, add:

```
camera.far = 5
```

The default for the camera's far plane is 100, and it's difficult to visualize. A `far` of 5 is quite close and easy to visualize but will temporarily cut off a lot of the scene.



- Build and run the app.



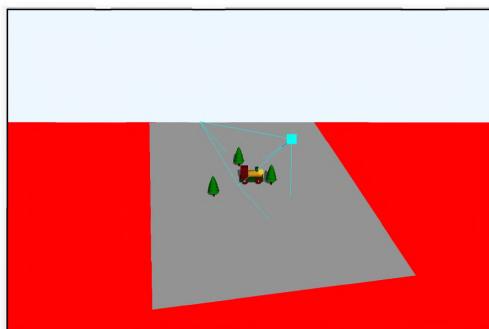
*Some of the scene is missing.*

You can see that much of the scene is missing due to the closer far plane.

- Press the number 3 key on the keyboard above the letters.

Here, you pause the sun's rotation and create a new perspective arcball camera that looks down on the scene from afar and renders the original perspective scene camera frustum in blue wireframe.

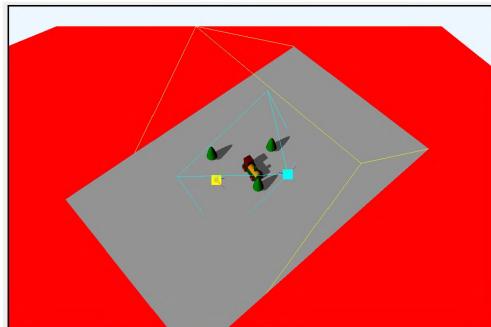
Using the mouse or trackpad, drag the scene to rotate and examine it. You'll see that the third tree lies outside the blue frustum and therefore isn't rendered. You'll also see where the shadow texture covers the scene. Red areas lie outside the shadow texture.



*The scene camera frustum*

- Press the number 4 key.

The orthographic light's camera view volume wireframe shows in yellow.



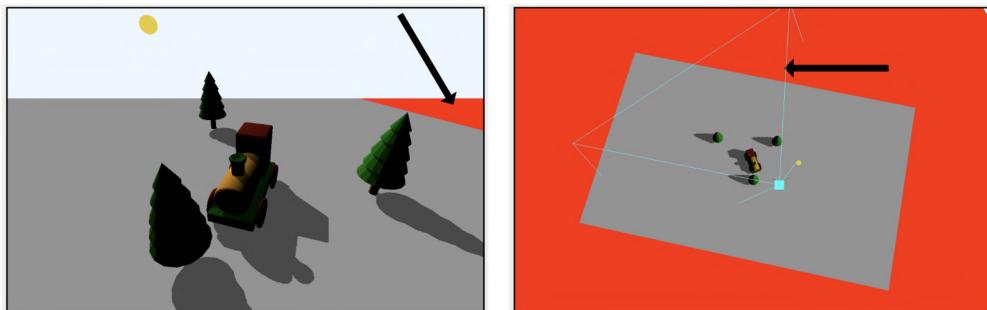
*The light view volume*

One edge of the light's view volume is from the corners of the gray plane. That's where the light's frustum doesn't reach and shows up on the shadow map texture as white.

- In GameScene, change `camera.far = 5` to:

```
camera.far = 10
```

- Build and run the app. When you see a patch of red plane, press the 3 key then the 4 key.

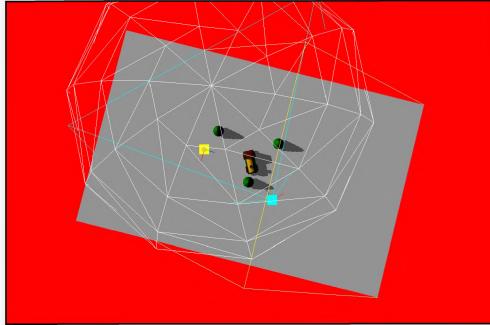


*Understanding why the scene captures area off texture*

Rotating the scene, you'll see that the blue wireframe extends into the red area. The yellow wireframe should enclose that area but currently doesn't.

- Press the number 5 key.

This shows a white bounding sphere that encloses the scene camera's frustum.



*The scene camera frustum's bounding sphere*

The light volume should enclose the white bounding sphere to get the best shadows.

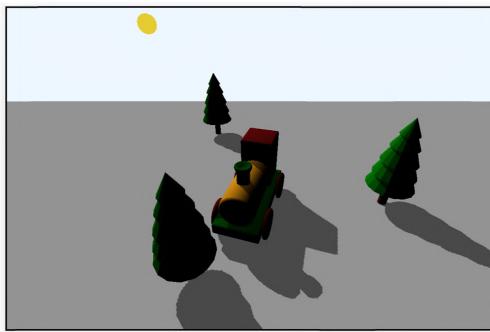
## Solving the Problems

- In the **Game** group, open **ShadowCamera.swift**. This file contains various methods to calculate the corners of the camera frustum.  
`createShadowCamera(using:lightPosition:)` creates an orthographic camera that encloses the specified camera.
- Open **Renderer.swift**. In `updateUniforms(scene:)`, replace all of the shadow code from `shadowCamera.viewSize = 16` to the end of the method with:

```
let sun = scene.lighting.lights[0]
shadowCamera = OrthographicCamera.createShadowCamera(
    using: scene.camera,
    lightPosition: sun.position)
uniforms.shadowProjectionMatrix = shadowCamera.projectionMatrix
uniforms.shadowViewMatrix = float4x4(
    eye: shadowCamera.position,
    center: shadowCamera.center,
    up: [0, 1, 0])
```

Here, you create an orthographic light camera with a view volume that completely envelopes the `scene.camera`'s frustum.

- Build and run the app.



*Light view volume encloses scene camera frustum*

Now the light camera volume encloses the whole scene so that you won't see any red errors or erroneous gray patches.

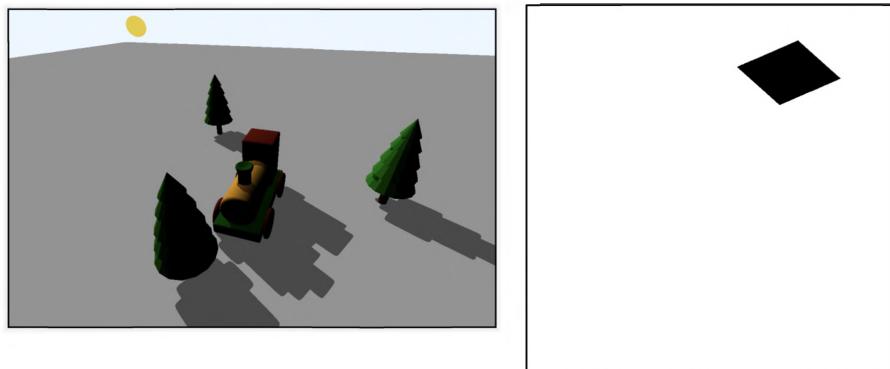
- Open **GameScene.swift**. In `init()`, remove:

```
camera.far = 10
```

You remove the assignment to `camera.far`, which restores the default `camera.far` to 100.

- Build and run the app.

Due to the huge light view volume, the shadows are very blocky. The image below shows the rendered shadow texture on the right. You can see almost no details.



*Blocky shadows when the light volume is too large*

You can change `far` back to `5` or `20` and capture the GPU workload to compare shadow texture quality.

This is one situation where you, as the game designer, would have to decide on the shadow quality. The best outcome would be to use the `far` value of `5` on shadows closer to the camera and a `far` value of `20` for shadows farther away, as the resolution won't matter so much.

## Cascaded Shadow Mapping

Modern games use a technique known as **cascaded shadow maps** to help balance performance and shadow depth. In Chapter 8, “Textures”, you learned about mip maps, textures of varying sizes used by the GPU depending on the distance from the camera. Cascaded shadow maps employ a similar idea.

With cascaded shadow maps, you render the scene to several shadow maps in a depth texture array using different near and far planes. As you've seen, the smaller `far` value creates a smaller light volume, which produces a more detailed shadow map. You sample the shadow from the shadow map with the smaller light volume for the fragments closer to the scene camera.

Farther away, you don't need as much accuracy, so you can sample the shadow from the larger light frustum that takes in more of the scene. The downside is that you have to render the scene multiple times, once for each shadow map.

Shadows can take a lot of calculation and processing time. You have to decide how much of your frame time allowance to give them. In the resources folder for this chapter, **references.markdown** contains some articles about common techniques to improve your shadows.



## Key Points

- A **shadow map** is a render taken from the point of the light casting the shadow.
- You capture a **depth map** from the perspective of the light in a first render pass.
- A second render pass then compares the depth of the rendered fragment with the stored depth map fragment. If the fragment is in shadow, you shade the diffuse color accordingly.
- The best shadows are where the light view volume exactly encases the scene camera's frustum. However, you have to know how much of the scene is being captured. If the area is large, shadows will be blocky.
- Shadows are expensive. A lot of research has gone into rendering shadows, and there are many different methods of improvements and techniques. **Cascaded shadow mapping** is the most common modern technique.

# Chapter 14: Deferred Rendering

Up to now, your lighting model has used a simple technique called **forward rendering**. With traditional forward rendering, you draw each model in turn. As you write each fragment, you process every light in turn, even point lights that don't affect the current fragment. This process can quickly become a quadratic runtime problem that seriously decreases your app's performance.

Assume you have a hundred models and a hundred lights in the scene. Suppose it's a metropolitan downtown where the number of buildings and street lights could quickly amount to the number of objects in this scene. At this point, you'd be looking for an alternative rendering technique.

**Deferred rendering**, also known as **deferred shading** or **deferred lighting**, does two things:

- In the first pass, it collects information such as material, normals and positions from the models and stores them in a special buffer for later processing in the fragment shader. Unnecessary calculations don't occur in this first pass. The special buffer is named the **G-buffer**, where **G** is for **Geometry**.
- In the second pass, it processes all lights in a fragment shader, but only where the light affects the fragment.

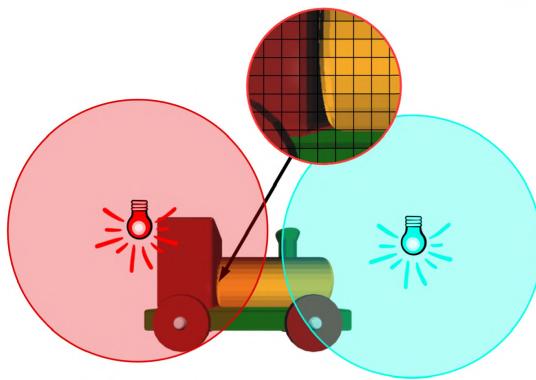


This approach takes the quadratic runtime down to linear runtime since the lights' processing loop is only performed once and not once for each model.

Look at the forward rendering algorithm:

```
// single pass
for each model {
    for each fragment {
        for each light {
            if directional { accumulate lighting }
            if point { accumulate lighting }
            if spot { accumulate lighting }
        }
    }
}
```

You effected this algorithm in Chapter 10, "Lighting Fundamentals".



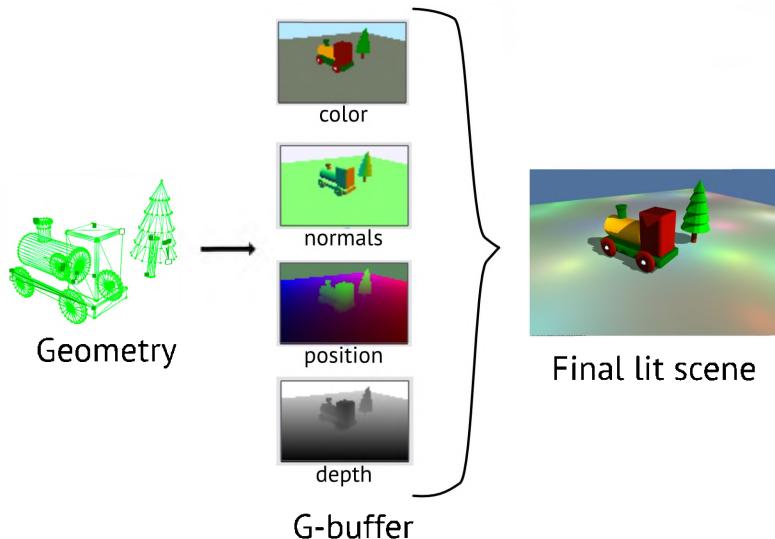
*Point lights affecting fragments*

In forward rendering, you process both lights for the magnified fragments in the image above even though the blue light on the right won't affect them.

Now, compare it to the deferred rendering algorithm:

```
// pass 1 - g-buffer capture
for each model {
    for each fragment {
        capture color, position, normal and shadow
    }
}
// pass 2 - light accumulation
render a quad
for each fragment { accumulate directional light }
render geometry for point light volumes
for each fragment { accumulate point light }
```

```
render geometry for spot light volumes
for each fragment { accumulate spot light }
```



*Four textures comprise the G-buffer*

While you have more render passes with deferred rendering, you process fewer lights. All fragments process the directional light, which shades the albedo along with adding the shadow from the directional light. But for the point light, you render special geometry that only covers the area the point light affects. The GPU will process only the affected fragments.

Here are the steps you'll take throughout this chapter:

- The first pass renders the shadow map. You've already done this.
- The second pass constructs G-buffer textures containing these values: material color (or albedo) with shadow information, world space normals and positions.
- Using a full-screen quad, the third and final pass processes the directional light. The same pass then renders point light volumes and accumulates point light information. If you have spotlights, you would repeat this process.

**Note:** Apple GPUs can combine the second and third passes. Chapter 15, “Tile-Based Deferred Rendering”, will revise this chapter’s project to take advantage of this feature.

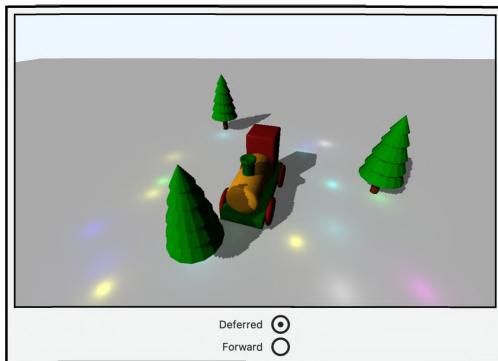
## The Starter Project

► In Xcode, open the starter project for this chapter. The project is almost the same as the end of the previous chapter, with some refactoring and reorganization. There's new lighting, with extra point lights. The camera and light debugging features from the previous chapter are gone.

Take note of the following additions:

- In the **Game** group, in **SceneLighting.swift**,  
`createPointLights(count:min:max:)` creates multiple point lights.
- Since you'll deal with many lights, the light buffer is greater than 4k. This means that you won't be able to use `setFragmentBytes(_:length:index:)`. Instead, scene lighting is now split out into three light buffers: one for sunlight, one for point lights and one that contains both sun and point lights, so that forward rendering still works as it did before. Spotlighting isn't implemented here.
- In the **Render Passes** group, **GBufferRenderPass.swift** is a copy of **ForwardRenderPass.swift** and is already set up in `Renderer`. You'll work on this render pass and change it to suit deferred rendering.
- In the app, a radio button below the metal view gives you the option to switch between render pass types. There won't be any difference in the render at this point.
- For simplicity, the renderer returns to phong shading rather than processing textures for PBR.
- In the **Shaders** group, in **Lighting.metal**, `phongLighting`'s conditional code is refactored into separate functions, one for each lighting method.
- **icosphere.obj** is a new model you'll use later in the chapter.

- Build and run the app, and ensure that you know how all of the code fits together.



*The starter app*

The twenty point lights are random, so your render may look slightly different.

**Note:** To visualize where the point lights are, uncomment the `DebugLights` draw at the end of `ForwardRenderPass.swift`. You'll see the point light positions when you choose the Forward option in the app.

## The G-buffer Pass

All right, time to build up that G-buffer!

- In the **Render Passes** group, open `GBufferRenderPass.swift`, and add four new texture properties to `GBufferRenderPass`:

```
var albedoTexture: MTLTexture?  
var normalTexture: MTLTexture?  
var positionTexture: MTLTexture?  
var depthTexture: MTLTexture?
```

These are the textures the G-buffer requires.

- Add this to `resize(view:size:)`:

```
albedoTexture = Self.makeTexture(  
    size: size,  
    pixelFormat: .bgra8Unorm,  
    label: "Albedo Texture")  
normalTexture = Self.makeTexture(  
    size: size,  
    pixelFormat: .rgba16Float,  
    label: "Normal Texture")  
positionTexture = Self.makeTexture(  
    size: size,  
    pixelFormat: .rgba16Float,  
    label: "Position Texture")  
depthTexture = Self.makeTexture(  
    size: size,  
    pixelFormat: .depth32Float,  
    label: "Depth Texture")
```

Here, you create the four textures with the desired pixel formats. `bgra8Unorm` has the format of four 8-bit unsigned components, which store integer values between 0 and 255. However, you'll need to store the position and normal values in higher precision than the color values by using `rgba16Float`.

- In the **Shaders** group, open **Common.h**, and add a new enumeration for the extra texture indices:

```
typedef enum {  
    RenderTargetAlbedo = 1,  
    RenderTargetNormal = 2,  
    RenderTargetPosition = 3  
} RenderTargetIndices;
```

These are the names for the render target texture indices.

In the **Geometry** group, open **VertexDescriptor.swift**, and add the syntactic sugar extension:

```
extension RenderTargetIndices {  
    var index: Int {  
        return Int(rawValue)  
    }  
}
```

Using values from `RenderTargetIndices` will now be easier to read.

► Open **Pipelines.swift**.

You'll create all the pipeline states here. You can compare what each pipeline state requires as you progress through the chapter.

Currently, `createGBufferPSO(colorPixelFormat:)` and `createForwardPSO(colorPixelFormat:)` are the same, but you'll need to change the G-buffer pipeline state object to specify the different texture formats.

► At the end of the file, create a new extension:

```
extension MTLRenderPipelineDescriptor {
    func setGBufferPixelFormats() {
        colorAttachments[RenderTargetAlbedo.index]
            .pixelFormat = .bgra8Unorm
        colorAttachments[RenderTargetNormal.index]
            .pixelFormat = .rgba16Float
        colorAttachments[RenderTargetPosition.index]
            .pixelFormat = .rgba16Float
    }
}
```

These color attachment pixel formats are the same as the ones you used for the albedo, normal and position textures.

► In `createGBufferPSO(colorPixelFormat:)`, replace:

```
pipelineDescriptor.colorAttachments[0].pixelFormat
= colorPixelFormat
```

► With:

```
pipelineDescriptor.colorAttachments[0].pixelFormat
= .invalid
pipelineDescriptor.setGBufferPixelFormats()
```

This sets the three color attachment pixel formats. Notice that you're not using `colorAttachments[0]` any more, as `RenderTargetIndices` starts at 1. You could use `0` for the albedo since you're not using the drawable in this pass, but you'll combine passes in the next chapter, so you leave color attachment `0` available for this event.

For the vertex function, you can reuse `vertex_main` from the main render pass, as all this does is transform the positions and normals. However, you'll need a new fragment function that stores the position and normal data into textures and doesn't process the lighting.



- Still in `createGBufferPSO(colorPixelFormat:)`, replace "fragment\_main" with:

```
"fragment_gBuffer"
```

That completes the pipeline state object setup. Next, you'll deal with the render pass descriptor.

- Open `GBufferRenderPass.swift`, and add this code to the bottom of `init(view:)`:

```
descriptor = MTLRenderPassDescriptor()
```

Here, you create a new render pass descriptor instead of using the view's automatically-generated render pass descriptor.

- At the top of `draw(commandBuffer:scene:uniforms:params:)`, add:

```
let textures = [
    albedoTexture,
    normalTexture,
    positionTexture
]
for (index, texture) in textures.enumerated() {
    let attachment =
        descriptor?.colorAttachments[RenderTargetAlbedo.index +
    index]
    attachment?.texture = texture
    attachment?.loadAction = .clear
    attachment?.storeAction = .store
    attachment?. clearColor =
        MTLClearColor(red: 0.73, green: 0.92, blue: 1, alpha: 1)
}
descriptor?.depthAttachment.texture = depthTexture
descriptor?.depthAttachment.storeAction = .dontCare
```

You iterate through each of the three textures that you'll write in the fragment function and add them to the render pass descriptor's color attachments. If the load action is `clear` when you add the color attachment, you can set the clear color. The scene depicts a sunny day with sharp shadows, so you set the color to sky blue. `store` ensures that the color textures don't clear before the next render pass. However, you won't need the depth attachment after this render pass, so you set this store action to `dontCare`.

- Still in `draw(commandBuffer:scene:uniforms:params:)`, remove:

```
renderEncoder.setFragmentBuffer(
```



```
scene.lighting.lightsBuffer,  
offset: 0,  
index: LightBuffer.index)
```

In the initial pass, you only store the albedo, or base color, and mark fragments as shadowed or not. You don't need the light buffer because you previously processed the shadow matrices in the shadow render pass.

Currently, you send the view's render pass descriptor to `GBufferRenderPass`. However, you must change this since you created a new one.

► Open `Renderer.swift`. In `draw(scene:in:)`, remove:

```
gBufferRenderPass.descriptor = descriptor
```

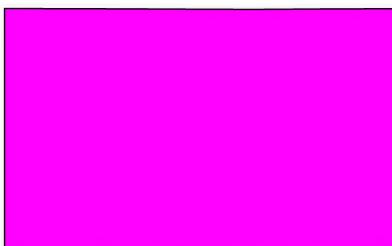
Before you test all of this code, you must create the new fragment shader.

► In the **Shaders** group, create a new Metal File named **Deferred.metal**. Add this code to the new file:

```
#import "Vertex.h"  
#import "Lighting.h"  
  
fragment float4 fragment_gBuffer(  
    VertexOut in [[stage_in]],  
    depth2d<float> shadowTexture [[texture(ShadowTexture)]],  
    constant Material &material [[buffer(MaterialBuffer)]])  
{  
    return float4(material.baseColor, 1);  
}
```

Here, you take in the results of the vertex function, the shadow texture from the shadow render pass, and the object's material. You return the base color of the material so that you'll be able to see something in the render.

► Build and run the app.



*The current drawable contains randomness*

Currently, you aren't writing anything to the view's drawable, only to the G-buffer render pass descriptor textures. So you'll get something random on your app window. Mine comes out a lovely shade of magenta.

- Capture the GPU workload, and click the Command Buffer to see what's happening there.

You'll probably get an error stating that it can't harvest a resource, but ignore that for the moment.



*Frame capture with G-buffer textures*

From this result, you can see that you successfully stored the shadow texture from the shadow pass as well as the three color and depth textures from your G-buffer pass, cleared to your sky blue color.

- Open `Deferred.metal`, and add a new structure before `fragment_gBuffer`:

```
struct GBUFFEROUT {
    float4 albedo [[color(RenderTargetAlbedo)]];
    float4 normal [[color(RenderTargetNormal)]];
    float4 position [[color(RenderTargetPosition)]];
};
```

These correspond to the pipeline states and render pass descriptor color attachment textures.

- Replace `fragment_gBuffer()` with:

```
// 1
fragment GBUFFEROUT fragment_gBuffer(
    VertexOut in [[stage_in]],
    depth2d<float> shadowTexture [[texture(ShadowTexture)]],
```

```

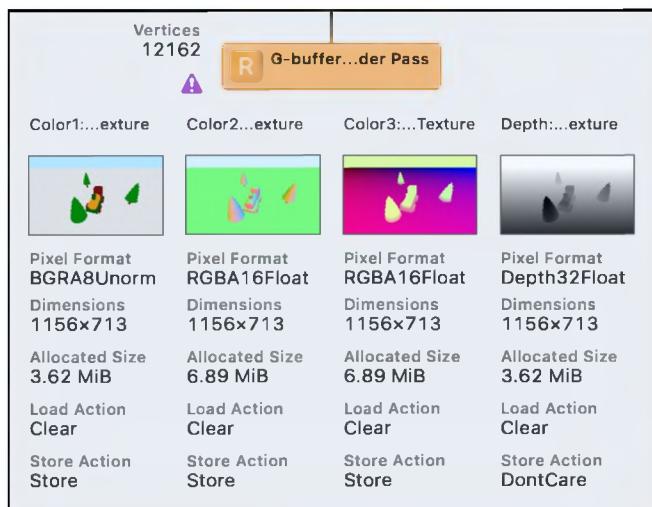
constant Material &material [[buffer(MaterialBuffer)]]
{
    GBufferOut out;
    // 2
    out.albedo = float4(material.baseColor, 1.0);
    // 3
    out.albedo.a = calculateShadow(in.shadowPosition,
        shadowTexture);
    // 4
    out.normal = float4(normalize(in.worldNormal), 1.0);
    out.position = float4(in.worldPosition, 1.0);
    return out;
}

```

Here, you

1. Return `GBufferOut` from the fragment function instead of only a single color value.
2. Set the albedo texture to the material's base color.
3. Calculate whether the fragment is in shadow, using the shadow position and the shadow texture. The shadow value is a single float. Since you don't use the alpha channel of the albedo texture, you can store the shadow value there.
4. Write the normal and position values into the corresponding texture.

► Build and run the app, and capture the GPU workload.



*G-buffer textures containing data*

`fragment_gBuffer` now writes to your three color textures.

## The Lighting Pass

Up to this point, you rendered the scene to multiple render targets, saving them for later use in the fragment shader. By rendering a full-screen quad, you can cover every pixel on the screen. This lets you process each fragment from your three textures and calculate lighting for each fragment. The results of this composition pass will end up in the view's drawable.

- Create a new Swift file named **LightingRenderPass** in the **Render Passes** group. Replace the contents with:

```
import MetalKit

struct LightingRenderPass: RenderPass {
    let label = "Lighting Render Pass"
    var descriptor: MTLRenderPassDescriptor?
    var sunLightPSO: MTLRenderPipelineState
    let depthStencilState: MTLDepthStencilState?
    weak var albedoTexture: MTLTexture?
    weak var normalTexture: MTLTexture?
    weak var positionTexture: MTLTexture?

    func resize(view: MTKView, size: CGSize) {}

    func draw(
        commandBuffer: MTLCommandBuffer,
        scene: GameScene,
        uniforms: Uniforms,
        params: Params
    ) {
    }
}
```

With this code, you add the necessary conformance to **RenderPass** and the texture properties you need for this consolidation pass.

You'll accumulate output from all light types when running the lighting pass. Each type of light needs a different fragment function, so you'll need multiple pipeline states. First, you'll create a pipeline state object for rendering the sun's directional light and return later and add a point light pipeline state object.

- Open **Pipelines.swift**, and copy `createForwardPSO(colorPixelFormat:)` to a new method named `createSunLightPSO(colorPixelFormat:)`.

Instead of rendering models, you'll render a quad for the lighting pass. You can define the vertices on the GPU and create a simple vertex function.

- In `createSunLightPSO(colorPixelFormat:)`, replace "vertex\_main" with:

```
"vertex_quad"
```

This vertex function is responsible for positioning the quad vertices.

- Replace "fragment\_main" with:

```
"fragment_deferredSun"
```

- Remove:

```
pipelineDescriptor.vertexDescriptor =
    MTLVertexDescriptor.defaultLayout
```

For this quad, you don't need the vertex descriptor. If you don't remove it, the GPU expects many buffers at the various bindings as described by the default vertex descriptor in `VertexDescriptor.swift`.

- Open `LightingRenderPass.swift`, and add the initializer to `LightingRenderPass`:

```
init(view: MTKView) {
    sunLightPSO = PipelineStates.createSunLightPSO(
        colorPixelFormat: view.colorPixelFormat)
    depthStencilState = Self.buildDepthStencilState()
}
```

Here, you initialize the pipeline state object with your new pipeline state parameters.

- In `draw(commandBuffer:scene:uniforms:params:)`, add:

```
guard let descriptor = descriptor,
    let renderEncoder =
        commandBuffer.makeRenderCommandEncoder(
            descriptor: descriptor) else {
    return
}
renderEncoder.label = label
renderEncoder.setDepthStencilState(depthStencilState)
var uniforms = uniforms
renderEncoder.setVertexBytes(
    &uniforms,
    length: MemoryLayout<Uniforms>.stride,
    index: UniformsBuffer.index)
```

Since you'll draw the quad directly to the screen, you'll use the view's current render pass descriptor. You set up the render command encoder as usual with the depth stencil state and vertex uniforms.

- After the previous code, add:

```
renderEncoder.setFragmentTexture(  
    albedoTexture,  
    index: BaseColor.index)  
renderEncoder.setFragmentTexture(  
    normalTexture,  
    index: NormalTexture.index)  
renderEncoder.setFragmentTexture(  
    positionTexture, index:  
    NormalTexture.index + 1)
```

This code passes the three attachments from the G-buffer render pass. Note the laziness of adding one to the index for the position. This is a mistake waiting to happen, and you should name the index at a later time.

- Create a new method for processing the sun light:

```
func drawSunLight(  
    renderEncoder: MTLRenderCommandEncoder,  
    scene: GameScene,  
    params: Params  
) {  
    renderEncoder.pushDebugGroup("Sun Light")  
    renderEncoder.setRenderPipelineState(sunLightPS0)  
    var params = params  
    params.lightCount = UInt32(scene.lighting.sunlights.count)  
    renderEncoder.setFragmentBytes(  
        &params,  
        length: MemoryLayout<Params>.stride,  
        index: ParamsBuffer.index)  
    renderEncoder.setFragmentBuffer(  
        scene.lighting.sunBuffer,  
        offset: 0,  
        index: LightBuffer.index)  
    renderEncoder.drawPrimitives(  
        type: .triangle,  
        vertexStart: 0,  
        vertexCount: 6)  
    renderEncoder.popDebugGroup()  
}
```

Here, you send the sun light details to the fragment function and draw the six vertices of a quad.

- Call this new method at the end of  
`draw(commandBuffer:scene:uniforms:params:)`:

```
drawSunLight(  
    renderEncoder: renderEncoder,  
    scene: scene,  
    params: params)  
renderEncoder.endEncoding()
```

You call the method and end the render pass encoding.

## Updating Renderer

You'll now add the new lighting pass to `Renderer` and pass in the necessary textures and render pass descriptor.

- Open `Renderer.swift`, and add a new property to `Renderer`:

```
var lightingRenderPass: LightingRenderPass
```

- Add this line before `super.init()`:

```
lightingRenderPass = LightingRenderPass(view: metalView)
```

You initialized the lighting render pass.

- Add this line to `mtkView(_:drawableSizeWillChange:)`:

```
lightingRenderPass.resize(view: view, size: size)
```

You currently don't resize any textures in `LightingRenderPass`. However, it's a good idea to call the `resize` method in case you add anything later.

- In `draw(scene:in:)`, locate if `options.renderChoice == .deferred`.

- At the end of the conditional closure, add:

```
lightingRenderPass.albedoTexture =  
gBufferRenderPass.albedoTexture  
lightingRenderPass.normalTexture =  
gBufferRenderPass.normalTexture  
lightingRenderPass.positionTexture =  
gBufferRenderPass.positionTexture  
lightingRenderPass.descriptor = descriptor  
lightingRenderPass.draw(  
    commandBuffer: commandBuffer,
```



```
scene: scene,
uniforms: uniforms,
params: params)
```

Here, you pass the textures to the lighting pass and set the render pass descriptor. You then process the lighting render pass.

You've set up everything on the CPU side. Now it's time to turn to the GPU.

## The Lighting Shader Functions

First, you'll create a vertex function that will position a quad. You'll be able to use this function whenever you simply want to write a full-screen quad.

- Open **Deferred.metal**, and add an array of six vertices for the quad:

```
constant float3 vertices[6] = {
    float3(-1, 1, 0),      // triangle 1
    float3(1, -1, 0),
    float3(-1, -1, 0),
    float3(-1, 1, 0),      // triangle 2
    float3(1, 1, 0),
    float3(1, -1, 0)
};
```

- Add the new vertex function:

```
vertex VertexOut vertex_quad(uint vertexID [[vertex_id]])
{
    VertexOut out {
        .position = float4(vertices[vertexID], 1)
    };
    return out;
}
```

For each of the six vertices, you return the position in the vertices array.

- Add the new fragment function:

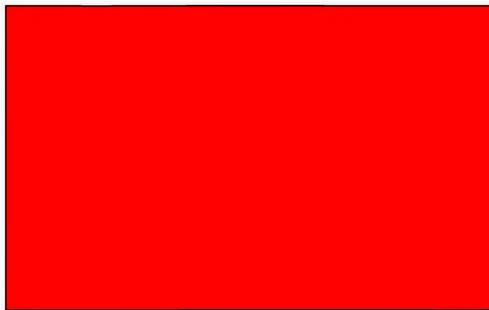
```
fragment float4 fragment_deferredSun(
    VertexOut in [[stage_in]],
    constant Params &params [[buffer(ParamsBuffer)]],
    constant Light *lights [[buffer(LightBuffer)]],
    texture2d<float> albedoTexture [[texture(BaseColor)]],
    texture2d<float> normalTexture [[texture(NormalTexture)]],
    texture2d<float> positionTexture [[texture(NormalTexture +
1)]])
```



```
{  
    return float4(1, 0, 0, 1);  
}
```

These are standard parameters for a fragment function. For now, you return the color red.

- Build and run the app, and you'll see a red screen, which is an excellent result as this is the color you currently return from `fragment_deferredSun`.



*Returning red from the fragment function*

You can now work out the lighting and make your render a little more exciting.

- Replace the contents of `fragment_deferredSun` with:

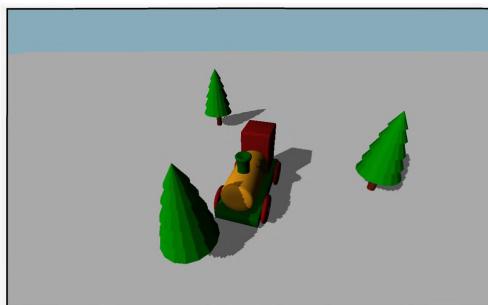
```
uint2 coord = uint2(in.position.xy);  
float4 albedo = albedoTexture.read(coord);  
float3 normal = normalTexture.read(coord).xyz;  
float3 position = positionTexture.read(coord).xyz;  
Material material {  
    .baseColor = albedo.xyz,  
    .specularColor = float3(0),  
    .shininess = 500  
};  
float3 color = phongLighting(  
    normal,  
    position,  
    params,  
    lights,  
    material);  
color *= albedo.a;  
return float4(color, 1);
```

Since the quad is the same size as the screen, `in.position` matches the screen position, so you can use it as coordinates for reading the textures. You then call `phongLighting` with the values you read from the textures. The material values should be captured in the previous G-buffer pass, but they're added here for brevity.



You stored the shadow in the albedo alpha channel in the G-buffer pass. So, after calculating the phong lighting for the sun and ambient lights, you simply multiply by the alpha channel to get the shadow.

- Build and run the app.



*Accumulating the directional light and shadows*

The result should be the same as the forward render pass, except for the point lights. However, you'll notice that the sky is a different color. The color changed because you calculate the sunlight for every fragment on the screen, even where there's no original model geometry. In the next chapter, you'll deal with this problem by using stencil masks.

This seems a lot of work for not much gain so far. But now comes the payoff!

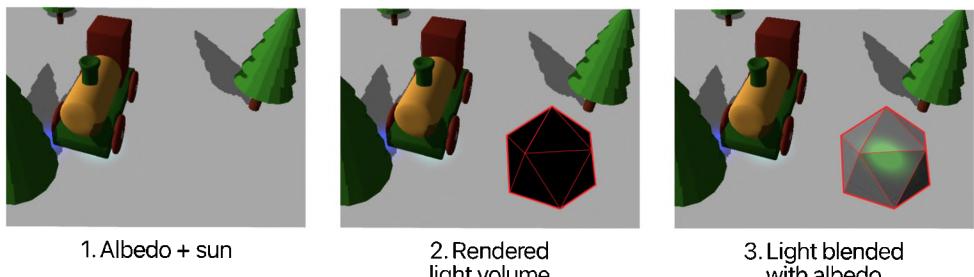
Instead of rendering just 20 point lights, you'll render as many as your device can take: in the order of thousands more than the forward renderer.

## Adding Point Lights

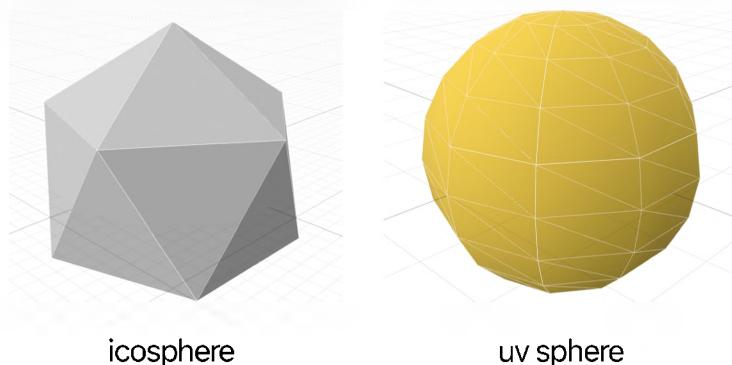
So far, you've drawn the plain albedo and shaded it with directional light. You need a second fragment function for calculating point lights.

In the original forward pass algorithm, you iterated through all the point lights for every fragment and performed the point light accumulation. Now you'll render a light volume in the shape of a sphere for every point light. Only the fragments you render for that light volume will require the point light accumulation.

The problem comes when one light volume is in front of another. The fragment function result will overwrite any previous result. You'll overcome this problem by accumulating the result into the final drawable by blending rather than overwriting the fragment.

*Blending the light volume*

Inside your starter project is a model named **icosphere.obj**. You'll render one of these for each point light.

*Icosphere and UV sphere*

The icosphere is a low-resolution sphere with only sixty vertices. Compare it to a UV sphere. The icosphere's faces are more regular and all have a similar area, whereas the UV sphere's faces are smaller at the two top and bottom poles.

This app assumes that all point lights have the same radius attenuation, which fits inside the icosphere. If a point light has a larger radius, the icosphere's straight edges could cut it off. You could also add more vertices to the icosphere, making it rounder, but that would make the rendering less efficient.

► Open **LightingRenderPass.swift**, and add a new property to **LightingRenderPass**:

```
var icosphere = Model(name: "icosphere.obj")
```

You initialize the sphere for later use.

Now you need a new pipeline state object with new shader functions to render the sphere.

- Open **Pipelines.swift**, and copy `createForwardPSO(colorPixelFormat:)` to a new method called `createPointLightPSO(colorPixelFormat:)`.
- Change the vertex function's name to "`vertex_pointLight`" and the fragment function's name to "`fragment_pointLight`".

Later, you'll need to add blending to the light accumulation. The pipeline state is how you tell the GPU that you require blending, so shortly. You'll add this to the pipeline state object.

- Open **LightingRenderPass.swift**, and add a new property for the pipeline state object:

```
var pointLightPSO: MTLRenderPipelineState
```

- Initialize the pipeline state in `init(view:)`:

```
pointLightPSO = PipelineStates.createPointLightPSO(  
    colorPixelFormat: view.colorPixelFormat)
```

- Create a new method to draw the point light volumes:

```
func drawPointLight(  
    renderEncoder: MTLRenderCommandEncoder,  
    scene: GameScene,  
    params: Params  
) {  
    renderEncoder.pushDebugGroup("Point lights")  
    renderEncoder.setRenderPipelineState(pointLightPSO)  
    renderEncoder.setVertexBuffer(  
        scene.lighting.pointBuffer,  
        offset: 0,  
        index: LightBuffer.index)  
    renderEncoder.setFragmentBuffer(  
        scene.lighting.pointBuffer,  
        offset: 0,  
        index: LightBuffer.index)  
}
```

The vertex function needs the light position to position each icosphere, while the fragment function needs the light attenuation and color.

- After the previous code, add:

```
guard let mesh = icosphere.meshes.first,
```

```
let submesh = mesh.submeshes.first else { return }
for (index, vertexBuffer) in mesh.vertexBuffers.enumerated() {
    renderEncoder.setVertexBuffer(
        vertexBuffer,
        offset: 0,
        index: index)
}
```

You set up the vertex buffers with the icosphere's mesh attributes.

## Instancing

If you had one thousand point lights, a draw call for each light volume would bring your system to a crawl. **Instancing** is a great way to tell the GPU to draw the same geometry a specific number of times. The GPU informs the vertex function which instance it's currently drawing so that you can extract information from arrays containing instance information.

In `SceneLighting`, you have an array of point lights with the position and color. Each of these point lights is an instance. You'll draw the icosphere mesh for each point light.

► After the previous code, add the draw call:

```
renderEncoder.drawIndexedPrimitives(
    type: .triangle,
    indexCount: submesh.indexCount,
    indexType: submesh.indexType,
    indexBuffer: submesh.indexBuffer,
    indexBufferOffset: submesh.indexBufferOffset,
    instanceCount: scene.lighting.pointLights.count)
renderEncoder.popDebugGroup()
```

Adding `instanceCount` to the draw call means that the GPU will repeat drawing the icosphere's vertex and submesh information for the specified number of instances. GPU hardware is optimized to do this.

► Call this new method from `draw(commandBuffer:scene:uniforms:params:)` before `renderEncoder.endEncoding()`:

```
drawPointLight(
    renderEncoder: renderEncoder,
    scene: scene,
    params: params)
```

## Creating the Point Light Shader Functions

- Open **Deferred.metal**, and add the new structures that the vertex function will need:

```
struct PointLightIn {
    float4 position [[attribute(Position)]];
};

struct PointLightOut {
    float4 position [[position]];
    uint instanceId [[flat]];
};
```

You're only interested in the position, so you only use the vertex descriptor's position attribute.

You send the position to the rasterizer, but you also send the instance ID so that the fragment function can extract the light details from the point lights array. You don't want any rasterizer interpolation, so you mark the instance ID with the attribute `[[flat]]`.

- Add the new vertex function:

```
vertex PointLightOut vertex_pointLight(
    PointLightIn in [[stage_in]],
    constant Uniforms &uniforms [[buffer(UniformsBuffer)]],
    constant Light *lights [[buffer(LightBuffer)]],
    // 1
    uint instanceId [[instance_id]])
{
    // 2
    float4 lightPosition = float4(lights[instanceId].position, 0);
    float4 position =
        uniforms.projectionMatrix * uniforms.viewMatrix
    // 3
    * (in.position + lightPosition);
    PointLightOut out {
        .position = position,
        .instanceId = instanceId
    };
    return out;
}
```

The points of interest are:

1. Use the attribute `[[instance_id]]` to detect the current instance.
2. Use the instance ID to index into the lights array.
3. Add the light's position to the vertex position. Since you're not dealing with scaling or rotation, you don't need to multiply by a model matrix.

► Add the new fragment function:

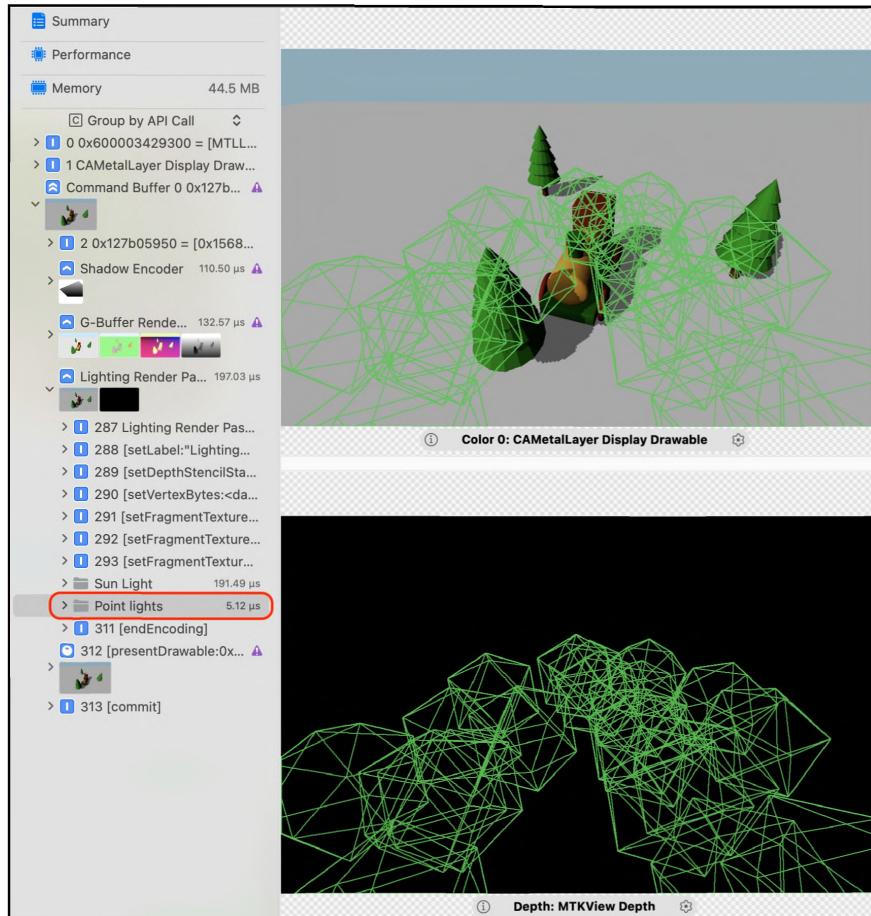
```
fragment float4 fragment_pointLight(  
    PointLightOut in [[stage_in]],  
    texture2d<float> normalTexture [[texture(NormalTexture)]],  
    texture2d<float> positionTexture  
        [[texture(NormalTexture + 1)]],  
    constant Light *lights [[buffer(LightBuffer)]])  
{  
    Light light = lights[in.instanceId];  
    uint2 coords = uint2(in.position.xy);  
    float3 normal = normalTexture.read(coords).xyz;  
    float3 position = positionTexture.read(coords).xyz;  
  
    Material material {  
        .baseColor = 1  
    };  
    float3 lighting =  
        calculatePoint(light, position, normal, material);  
    lighting *= 0.5;  
    return float4(lighting, 1);  
}
```

You extract the light from the lights array using the instance ID sent by the vertex function. Just as you did for the sun light, you read in the textures from the previous render pass and calculate the point lighting. You reduce the intensity by `0.5`, as blending will make the lights brighter.

Notice the base color is `1`. Rather than taking the color from the albedo, this time, you'll achieve the glowing point light with GPU blending.

► Build and run the app. Your render is as before. No glowing point lights here.

Capture the GPU workload and check out the attachments in the Point lights render encoder section:



*Point light volume drawing*

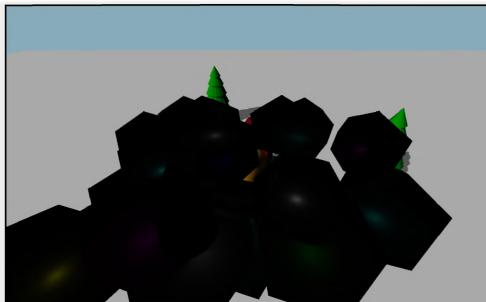
Theicospheres are drawing, but if you check the depth attachment with the magnifier, you'll see all the values are zero. There's a problem with the depth: theicospheres are not rendering in front of the quad.

► Open **LightingRenderPass.swift**, and add a new method:

```
static func buildDepthStencilState() -> MTLDepthStencilState? {
    let descriptor = MTLDepthStencilDescriptor()
    descriptor.isDepthWriteEnabled = false
    return Renderer.device.makeDepthStencilState(descriptor:
        descriptor)
}
```

This code overrides the default `RenderPass` protocol method. You disable depth writes because you always want the icospheres to render.

- Build and run the app.



*Rendering icospheres*

Your icosphere volumes now render in front of the quad.

## Blending

- Open `Pipelines.swift`. In `createPointLightPS0(colorPixelFormat:)`, add this code before return:

```
let attachment = pipelineDescriptor.colorAttachments[0]
attachment?.isBlendingEnabled = true
attachment?.rgbBlendOperation = .add
attachment?.alphaBlendOperation = .add
attachment?.sourceRGBBlendFactor = .one
attachment?.sourceAlphaBlendFactor = .one
attachment?.destinationRGBBlendFactor = .one
attachment?.destinationAlphaBlendFactor = .zero
attachment?.sourceRGBBlendFactor = .one
attachment?.sourceAlphaBlendFactor = .one
```

Here, you specify that you enabled blending. You shouldn't have this on by default because blending is an expensive operation. The other properties determine how to combine the source and destination fragments.

All these blending properties are at their defaults, except for `destinationRGBBlendFactor`. They're written out here in full to show what you can change. The important change is `destinationRGBBlendFactor` from zero to one, so blending will occur.

The icospheres will blend with the color already drawn in the background quad. Black will disappear, leaving only the light color.

- Build and run the app.



*A few point lights rendering*

Now it's time to crank up those point lights. Be careful when you switch to the forward renderer. If you have a lot of lights, your system will appear to hang while the forward render pass laboriously calculates the point light effect on each fragment.

- Open **SceneLighting.swift**. In `init()`, replace:

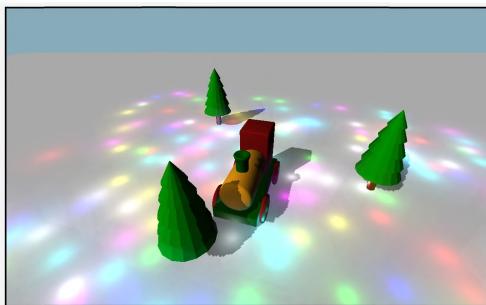
```
pointLights = Self.createPointLights(  
    count: 20,  
    min: [-3, 0.1, -3],  
    max: [3, 0.3, 3])
```

- With:

```
pointLights = Self.createPointLights(  
    count: 200,  
    min: [-6, 0.1, -6],  
    max: [6, 0.3, 6])
```

With this code, you create 200 point lights, specifying the minimum and maximum xyz values to constrain the lights to that area.

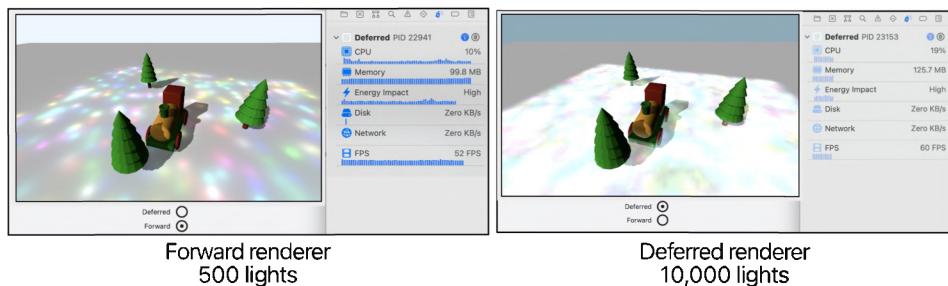
- Build and run the app.



*Two hundred point lights*

Your point lights blend beautifully and are comparable with the forward renderer. The only problem is the color of the sky, which you'll fix in the following chapter.

- On the **Debug navigator**, check your FPS for both Deferred and Forward.
- Continue gradually increasing count in the previous code until your forward pass FPS decreases below 60 FPS. Make a note of the number of lights for comparison.
- Increase count and check how many point lights your deferred render can manage before degrading below 60 FPS. Many lights are so bright that you may have to dial down the light color in `createPointLights(count:min:max:)` with `light.color *= 0.2`.



*Render algorithm comparison*

On my M1 Mac Mini, performance in a small window starts degrading on the forward renderer at about 400 lights, whereas the deferred renderer can cope with 10,000 lights. With the window maximized, forward rendering starts degrading at about 30 to 40 lights, whereas the deferred renderer manages more than 600. On an iPhone 12 Pro, the forward renderer degraded at 100 lights, whereas the deferred renderer could manage 1500 at 60 FPS.

This chapter has opened your eyes to two rendering techniques: forward and deferred. As the game designer, you get to choose your rendering method. Forward and deferred rendering are just two: Several other techniques can help you get the most out of your frame time.

There are also many ways of configuring your forward and deferred render passes. **references.markdown** in the **resources** folder for this chapter has a few links for further research.

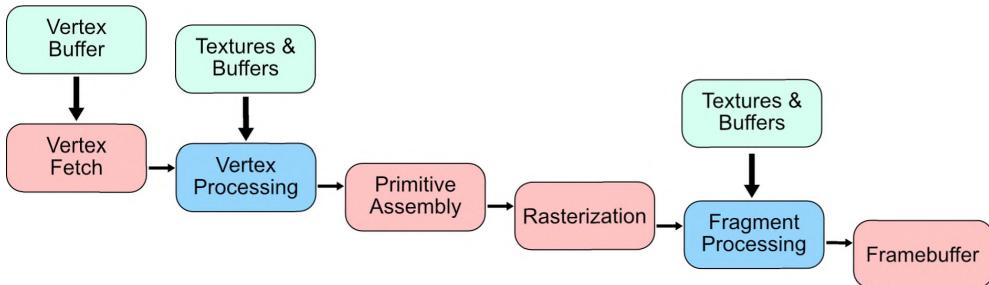
In the next chapter, you'll learn how to make your deferred render pass even faster by taking advantage of Apple's new Silicon.

## Key Points

- **Forward rendering** processes all lights for all fragments.
- **Deferred rendering** captures albedo, position and normals for later light calculation. For point lights, only the necessary fragments are rendered.
- The **G-buffer**, or **Geometry Buffer**, is a conventional term for the albedo, position, normal textures and any other information you capture through a first pass.
- An **icosphere** model provides a volume for rendering the shape of a point light.
- Using instancing, the GPU can efficiently render the same geometry many times.
- The pipeline state object specifies whether the result from the fragment function should be blended with the currently attached texture.

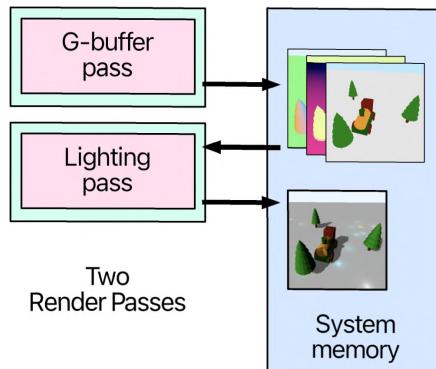
# Chapter 15: Tile-Based Deferred Rendering

Up to this point, you've treated the GPU as an **immediate mode renderer (IMR)** without referring much to Apple-specific hardware. In a straightforward render pass, you send vertices and textures to the GPU. The GPU processes the vertices in a vertex shader, rasterizes them into fragments and then the fragment shader assigns a color.



*Immediate mode pipeline*

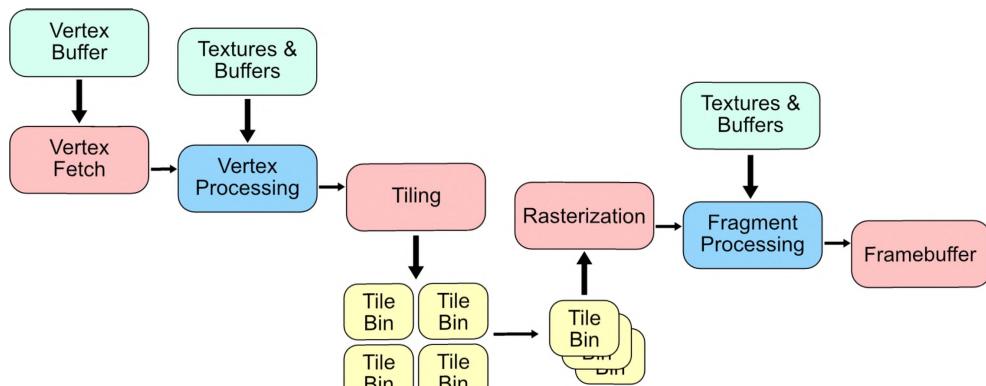
The GPU uses system memory to transfer resources between passes where you have multiple passes.



*Immediate mode using system memory*

Since the A7 64-bit mobile chip, Apple began transitioning to a **tile-based deferred rendering (TBDR)** architecture. With the arrival of Apple Silicon on Macs, this transition is complete.

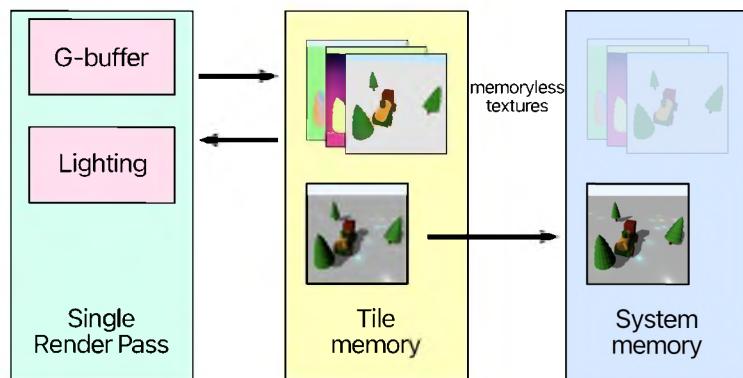
The TBDR GPU adds extra hardware to perform the primitive processing in a tiling stage. This process breaks up the screen into tiles and assigns the geometry from the vertex stage to a tile. It then forwards each tile to the rasterizer. Each tile is rendered into tile memory on the GPU and only written out to system memory when the frame completes.



*TBDR pipeline*

## Programmable Blending

Instead of writing the texture in one pass and reading it in the next pass, tile memory enables **programmable blending**. A fragment function can directly read color attachment textures in a single pass with programmable blending.



*Programmable blending with memoryless textures*

The G-buffer doesn't have to transfer the temporary textures to system memory anymore. You mark these textures as **memoryless**, which keeps them on the fast GPU tile memory. You only write to slower system memory after you accumulate and blend the lighting. This speeds up rendering because you use less bandwidth.

## Tiled Deferred Rendering

Confusingly, **tiled deferred rendering** can apply to the deferred rendering or shading technique as well as the name of an architecture. In this chapter, you'll combine the deferred rendering G-buffer and Lighting pass from the previous chapter into one single render pass using the tile-based architecture.

To complete this chapter, you need to run the code on a device with an Apple GPU. This device could be an Apple Silicon macOS device or any iOS device running the latest iOS 15. The iOS simulator and Intel Macs can't handle the code, but the starter project will run Forward Rendering in place of Tiled Deferred Rendering instead of crashing.

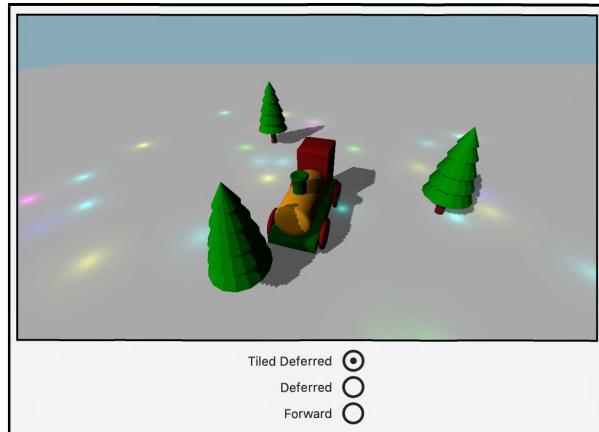
# The Starter Project

- In Xcode, open the starter project for this chapter.

This project is the same as the end of the previous chapter, except:

- In the **SwiftUI Views** group, there's a new option for `tiledDeferred` in **Options.swift**. Renderer will update `tiledSupported` depending on whether the device supports tiling.
- In the **Render Passes** group, the deferred rendering pipeline state creation methods in **Pipelines.swift** have an extra Boolean parameter of `tiled: .Later`, you'll assign a different fragment function depending on this parameter.
- A new file, **TiledDeferredRenderPass.swift**, combines `GBufferRenderPass` and `LightingRenderPass` into one long file. The code is substantially similar, with the two render passes combined into `draw(commandBuffer:scene:uniforms:params:)`. You'll convert this file from the immediate mode deferred rendering algorithm to tile-based deferred rendering.
- Renderer instantiates `TiledDeferredRenderPass` if the device supports tiling.

- Build and run the app on your TBDR device.



*The starter app*

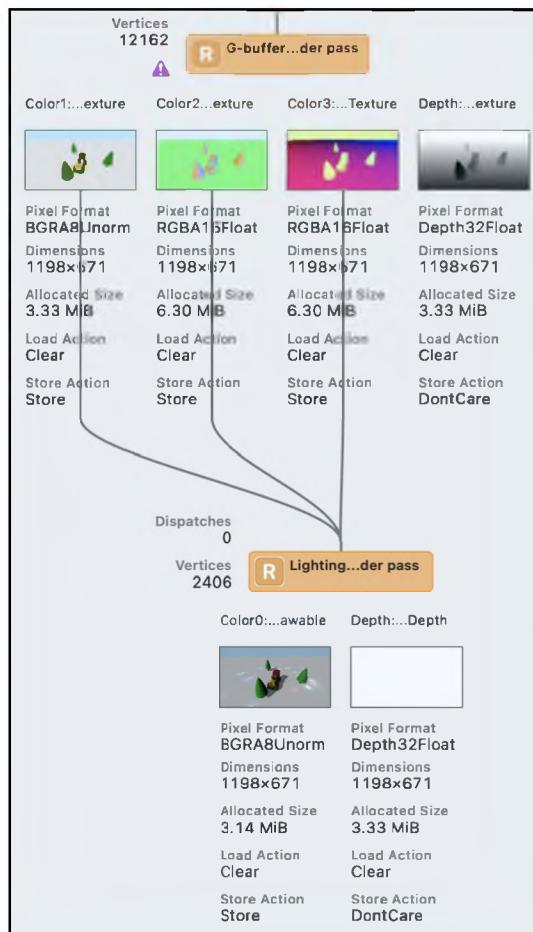
The render is the same as at the end of the previous chapter but with an added **Tiled Deferred** option under the Metal view.

**Note:** If you run the app on a non-TBDR device, the option will be marked **Tiled Deferred not Supported!**

Apple assigns **GPU families** to devices. The A7 is **iOS GPU family 1**. When using newer Metal features, use `device.supportsFamily(_ :)` to check whether the current device supports the capabilities you're requesting.

In `init(metalView:options:)`, `Renderer` checks the GPU family. If the device supports Apple family 3 GPUs, which Apple introduced with the A9 chip, it supports tile-based deferred rendering.

► Capture the GPU workload and refresh your memory on the render pass hierarchy:



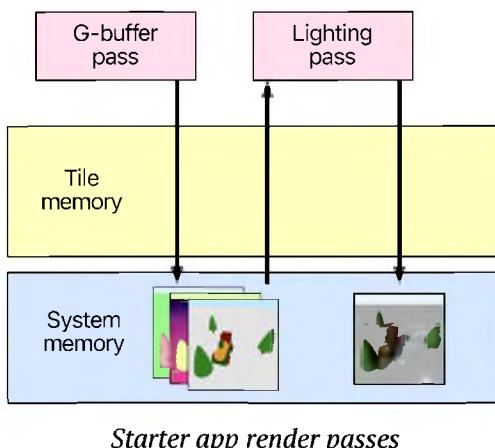
*GPU frame capture*



You have a G-buffer pass where you fill in the albedo, normal and position textures. You also have a Light accumulation pass, where you render a quad and calculate the lighting using the G-buffer textures.

In the **Render Passes** group, open **TiledDeferredRenderPass.swift** and examine the code. `draw(commandBuffer:scene:uniforms:params:)` contains both the G-buffer pass and the Lighting pass. There's a lot of code, but you should recognize it from the previous chapter.

► Currently, this is what happens during your render passes:



You write the G-buffer textures to system memory and then read them back from system memory.

These are the steps you'll take to move the G-buffer textures to tile memory:

1. Change the texture storage mode from private to `memoryless`.
2. Change the descriptor's color attachment store action for all the G-buffer textures to `dontCare`.
3. In the Lighting pass, stop sending the color attachment textures to the fragment function.
4. Create new fragment shaders for rendering the sun and point lights.
5. Combine the two render encoder passes with their descriptors into one.
6. Update the pipeline state objects to match the new render pass descriptor.

As you work through the chapter, you'll encounter common errors so you can learn how to fix them when you make them in the future.

## 1. Making the Textures Memoryless

- Open `TiledDeferredRenderPass.swift`. In `resize(view:size:)`, change the storage mode for all four textures from `storageMode: private` to:

```
storageMode: .memoryless
```

- Build and run the app.

You'll get an error: **Memoryless attachment content cannot be stored in memory**. You're still storing the attachment back to system memory. Time to fix that.

## 2. Changing the Store Action

- Stay in `TiledDeferredRenderPass.swift`. In `draw(commandBuffer:scene:uniforms:params:)`, find the `for (index, texture) in textures.enumerated()` loop and change `attachment?.storeAction = .store` to:

```
attachment?.storeAction = .dontCare
```

This line stops the textures from transferring to system memory.

- Build and run the app.

You'll get another error: **failed assertion `Set Fragment Buffers Validationtexture is Memoryless, and cannot be assigned`**. For the Lighting pass, you send the textures to the fragment shader as texture parameters. However, you can't do that with memoryless textures because they're already resident on the GPU. You'll fix that next.

## 3. Removing the Fragment Textures

- In `drawSunLight(renderEncoder:scene:params:)`, remove:

```
renderEncoder.setFragmentTexture(  
    albedoTexture,  
    index: BaseColor.index)  
renderEncoder.setFragmentTexture(  
    normalTexture,  
    index: NormalTexture.index)  
renderEncoder.setFragmentTexture(  
    positionTexture,
```



```
index: NormalTexture.index + 1)
```

- Build and run the app.

You'll probably get a black screen now because your deferred shader functions are expecting textures.

## 4. Creating the New Fragment Functions

- Still in **TiledDeferredRenderPass.swift**, in `init(view:)`, change the three pipeline state objects' `tiled: false` parameters to:

```
tiled: true
```

- Open **Pipelines.swift**. In `createSunLightPSO(colorPixelFormat:tiled:)` and `createPointLightPSO(colorPixelFormat:tiled:)`, check which fragment functions you need to create:

- `fragment_tiled_deferredSun`
- `fragment_tiled_pointLight`

You can still use the same vertex functions and G-buffer fragment function.

- In the **Shaders** group, open **Deferred.metal**.
- Copy `fragment_deferredSun` to a new function called `fragment_tiled_deferredSun`.
- In `fragment_tiled_deferredSun`, since you're not sending the fragment textures to the fragment function any more, remove the parameters:

```
texture2d<float> albedoTexture [[texture(BaseColor)]],  
texture2d<float> normalTexture [[texture(NormalTexture)]],  
texture2d<float> positionTexture [[texture(texture(NormalTexture  
+ 1))]]
```

- Add a new parameter:

```
GBufferOut gBuffer
```

`GBufferOut` is the structure that refers to the color attachment render target textures.



► Change:

```
uint2 coord = uint2(in.position.xy);
float4 albedo = albedoTexture.read(coord);
float3 normal = normalTexture.read(coord).xyz;
float3 position = positionTexture.read(coord).xyz;
```

► To:

```
float4 albedo = gBuffer.albedo;
float3 normal = gBuffer.normal.xyz;
float3 position = gBuffer.position.xyz;
```

Instead of swapping out to system memory, you now read the color attachment textures in the fast GPU tile memory. In addition to this speed optimization, you directly access the memory rather than reading in a texture.

Repeat this process for the point lights.

► Copy `fragment_pointLight` to a new function named `fragment_tiled_pointLight`

► Remove the parameters:

```
texture2d<float> normalTexture [[texture(NormalTexture)]],
texture2d<float> positionTexture [[texture(NormalTexture + 1)]],
```

► Add the parameter:

```
GBufferOut gBuffer
```

► Change:

```
uint2 coords = uint2(in.position.xy);
float3 normal = normalTexture.read(coords).xyz;
float3 position = positionTexture.read(coords).xyz;
```

► To:

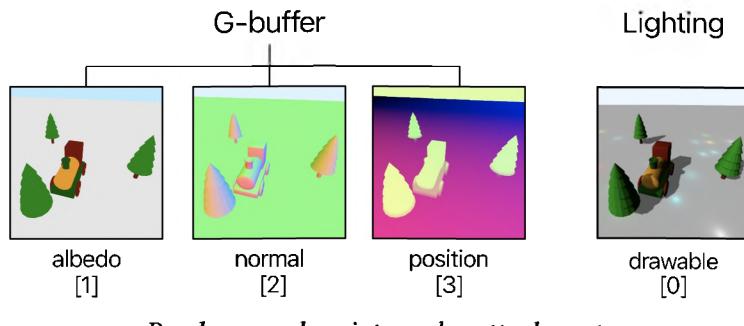
```
float3 normal = gBuffer.normal.xyz;
float3 position = gBuffer.position.xyz;
```

► Build and run the app.

When creating the sun light pipeline state, you now get the error: **Shaders reads from a color attachment whose pixel format is MTLPixelFormatInvalid.**



These are the attachments you set up in the two render passes in the previous chapter's Deferred Rendering:



*Render pass descriptor color attachments*

Currently, when writing the G-buffer in `fragment_gBuffer`, your render pass descriptor `colorAttachments[0]` is `nil`, and your pipeline state `colorAttachments[0]` pixel format is invalid.

However, when calculating directional light, you only set up `colorAttachments[0]`, which is done in **Pipelines.swift**, in `createSunLightPSO(colorPixelFormat:tiled:)`. When `fragment_tiled_deferredSun` reads from `gBuffer`, the other color attachment pixel formats are invalid.

Instead of using two render pass descriptors and render command encoders, you'll configure the view's current render pass descriptor to use all the color attachments. Then you'll set up the pipeline state configuration to match.

## 5. Combining the Two Render Passes

► Open **TiledDeferredRenderPass.swift**. In `draw(commandBuffer:scene:uniforms:params:)`, change `let descriptor = MTLRenderPassDescriptor()` to:

```
let descriptor = viewCurrentRenderPassDescriptor
```

You'll use the view's current render pass descriptor, passed in from `Renderer`, to configure your render command encoder.

- Still in `draw(commandBuffer:scene:uniforms:params:)`, remove:

```
renderEncoder.endEncoding()

// MARK: Lighting pass
// Set up Lighting descriptor
guard let renderEncoder =
    commandBuffer.makeRenderCommandEncoder(
        descriptor: viewCurrentRenderPassDescriptor) else {
    return
}
```

Here, you remove the second render command encoder.

## 6. Updating the Pipeline States

- Open **Pipelines.swift**. Add this code to `createSunLightPSO(colorPixelFormat:tiled:)` and `createPointLightPSO(colorPixelFormat:tiled:)` after setting `colorAttachments[0].pixelFormat`:

```
if tiled {
    pipelineDescriptor.setColorAttachmentPixelFormats()
}
```

This code sets the color pixel formats to match the render target textures.

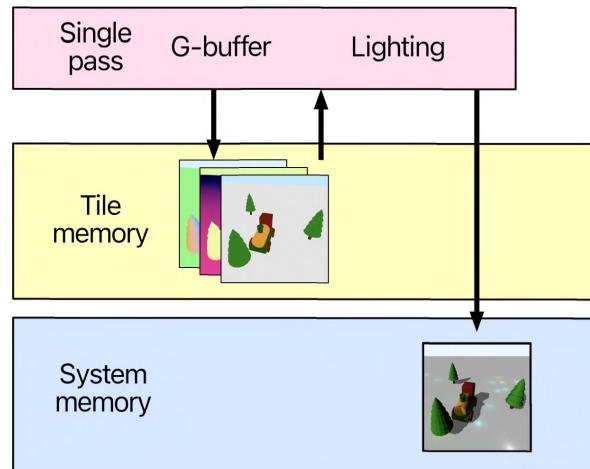
- In `createGBufferPSO(colorPixelFormat:tiled:)`, after setting `colorAttachments[0].pixelFormat`, add:

```
if tiled {
    pipelineDescriptor.colorAttachments[0].pixelFormat
        = colorPixelFormat
}
```

In the previous chapter, your G-buffer render pass descriptor had no texture in `colorAttachments[0]`. However, when you use the view's current render pass descriptor, `colorAttachment[0]` stores the view's current drawable texture, so you match that texture's pixel format.



Now you store the textures in tile memory and use a single render pass.



*A single render pass*

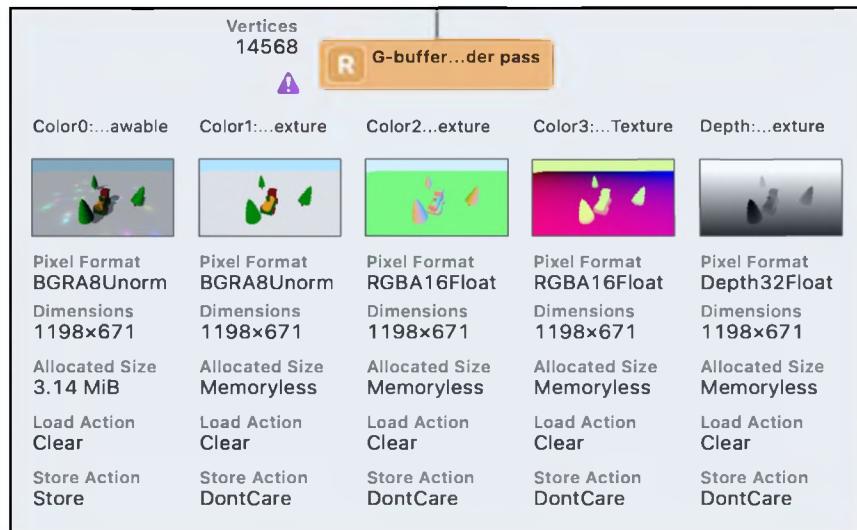
- Build and run the app.



*The final render*

Finally, you'll see the result you want. The render is the same whether you choose **Tiled Deferred** or **Deferred**.

- With the **Tiled Deferred** option selected, capture the GPU workload. You'll see that all your textures, aside from the shadow pass, process in the single render pass.



*The final frame capture*

Your four memoryless render target textures show up as **Memoryless** on the capture, proving they aren't taking up any system memory.

Now for the exciting part — to see how many lights you can run at 60 frames per second.

- Open **SceneLighting.swift**, and locate in `init()`:

```
pointLights = Self.createPointLights(
    count: 40,
    min: [-6, 0.1, -6],
    max: [6, 1, 6])
```

- While running your app and choosing the **Deferred** option, slowly raise the count of point lights until you're no longer getting 60 frames per second. Then see how many point lights you can get when choosing the **Tiled Deferred** option.

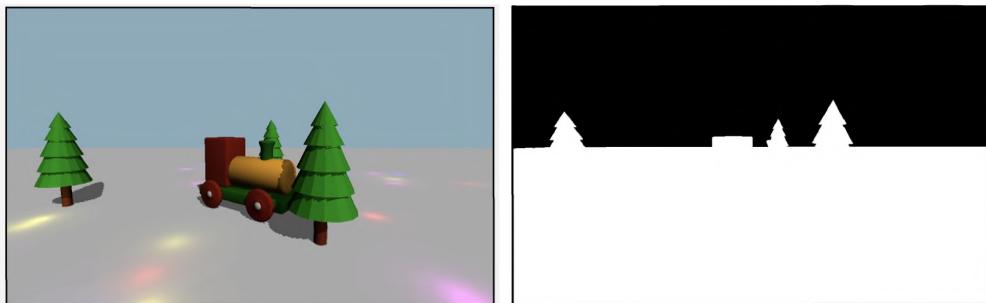
On **Tiled Deferred**, my M1 Mac mini runs 18,000 point lights at 60 FPS in a small window. On **Deferred**, it'll only achieve 38 FPS with the same number of lights. **Don't attempt Forward Rendering with this many lights!**

## Stencil Tests

The last step in completing your deferred rendering is to fix the sky. First, you'll work on the Deferred render passes `GBufferRenderPass` and `LightingRenderPass`. Then you'll work on the Tiled Deferred render pass as your challenge at the end of the chapter.

Currently, when you render the quad in the lighting render pass, you accumulate the directional lighting on all the quad's fragments. Wouldn't it be great to only process fragments where model geometry is rendered?

Fortunately, that's what stencil testing was designed to do. In the following image, the stencil texture is on the right. The black area should mask the image so that only the white area renders.



*Stencil testing*

As you already know, part of rasterization is performing a depth test to ensure the current fragment is in front of any fragments already rendered. The depth test isn't the only test the fragment has to pass. You can configure a stencil test.

Up to now, when you created the `MTLDepthStencilState`, you only configured the depth test. In the pipeline state objects, you set the depth pixel format to `depth32f float` with a matching depth texture.

A stencil texture consists of 8-bit values, from 0 to 255. You'll add this texture to the depth buffer so that the depth buffer will consist of both depth texture and stencil texture.

For a better understanding of the stencil buffer, examine the following image.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1
0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	1
0	0	0	1	2	1	1	1
0	0	1	2	2	1	1	1
0	0	0	0	0	0	0	0

A stencil texture

In this scenario, the buffer is initially cleared with zeros. When the pink triangle renders, the rasterizer increments the fragments the triangle covers. The second yellow triangle renders, and the rasterizer again increments the fragments that the triangle covers.

## Stencil Test Configuration

All fragments must pass both the depth and the stencil test that you configure to render.

As part of the configuration you set:

- The comparison function.
- The operation on pass or fail.
- A read and write mask.

Take a closer look at the comparison function.

### 1. The Comparison Function

When the rasterizer performs a stencil test, it compares a **reference value** with the value in the stencil texture using a comparison function. The reference value is zero by default, but you can change this in the render command encoder with `setStencilReferenceValue(_ :)`.

The comparison function is a mathematical comparison operator, such as `equal` or `lessEqual`. A comparison function of `always` will let the fragment pass the stencil test, whereas with a stencil comparison of `never`, the fragment will always fail.

For instance, if you want to use the stencil buffer to mask out the yellow triangle area in the previous example, you could set a reference value of 2 in the render command encoder and then set the comparison to `notEqual`. Only fragments that don't have their stencil buffer set to 2 will pass the stencil test.

## 2. The Stencil Operation

Next, you set the stencil operations to perform on the stencil buffer. There are three possible results to configure:

- Stencil test failure.
- Stencil test pass and depth failure.
- Stencil test pass and depth pass.

The default operation for each result is `keep`, which doesn't change the stencil buffer.

Other operations include:

- **incrementClamp**: The stencil buffer increments the stencil buffer fragment until the maximum of 255.
- **incrementWrap**: The stencil buffer increments the stencil buffer fragment and, if necessary, wraps around from 255 to 0.
- **decrementClamp** and **decrementWrap**: The same as increment, except the stencil buffer value decreases.
- **invert**: Performs a bitwise NOT operation, which inverts all of the bits.
- **replace**: Replaces the stencil buffer fragment with the reference value.

To get the stencil buffer to increase when a triangle renders in the previous example, you perform the `incrementClamp` operation when the fragment passes the depth test.

## 3. The Read and Write Mask

There's one more wrinkle. You can specify a read mask and a write mask. By default, these masks are 255 or 11111111 in binary. When you test a bit value against 1, the value doesn't change.

Now that you have the concept and principles under your belt, it's time to learn what all this means.

## Create the Stencil Texture

The stencil texture buffer is an extra 8-bit buffer attached to the depth texture buffer. You optionally configure it when you configure the depth buffer.

► Open **Pipelines.swift**. In `createGBufferPSO(colorPixelFormat:tiled:)`, after `pipelineDescriptor.depthAttachmentPixelFormat = .depth32Float`, add:

```
if !tiled {
    pipelineDescriptor.depthAttachmentPixelFormat
        = .depth32Float_stencil8
    pipelineDescriptor.stencilAttachmentPixelFormat
        = .depth32Float_stencil8
}
```

This code configures both the depth and stencil attachment to use one texture, including the 32-bit depth and the 8-bit stencil buffers.

► Open **GBufferRenderPass.swift**. In `resize(view:size:)`, change `depthTexture` to:

```
depthTexture = Self.makeText(
    size: size,
    pixelFormat: .depth32Float_stencil8,
    label: "Depth and Stencil Texture")
```

Here, you create the texture with the matching pixel format.

► In `draw(commandBuffer:scene:uniforms:params:)`, after configuring the descriptor's depth attachment, add:

```
descriptor?.stencilAttachment.texture = depthTexture
descriptor?.stencilAttachment.storeAction = .store
```

With this code, you tell the descriptor to use the depth texture as the stencil attachment and store the texture after use.

- Build and run the app, and choose the Deferred option.
- Capture the GPU workload and examine the command buffer.

Vertices 12162		R G-Bu...der Pass		
Color0...Texture	Color1...Texture	Color2...Texture	Depth...Texture	Stencil...Texture
Pixel Format BGRA8Unorm	Pixel Format RGBA16Float	Pixel Format RGBA16Float	Pixel Format Depth...encil8	Pixel Format Depth...encil8
Dimensions 1178x725	Dimensions 1178x725	Dimensions 1178x725	Dimensions 1178x725	Dimensions 1178x725
Allocated Size 3.62 MiB	Allocated Size 6.89 MiB	Allocated Size 6.89 MiB	Allocated Size 4.62 MiB	Allocated Size 4.62 MiB
Load Action Clear	Load Action Clear	Load Action Clear	Load Action Clear	Load Action Clear
Store Action Store	Store Action Store	Store Action Store	Store Action DontCare	Store Action Store

New stencil texture

Sure enough, you now have a stencil texture along with your other textures.

## Configure the Stencil Operation

- Open **GBufferRenderPass.swift**, and add this new method:

```
static func buildDepthStencilState() -> MTLDepthStencilState? {
    let descriptor = MTLDepthStencilDescriptor()
    descriptor.depthCompareFunction = .less
    descriptor.isDepthWriteEnabled = true
    return Renderer.device.makeDepthStencilState(
        descriptor: descriptor)
}
```

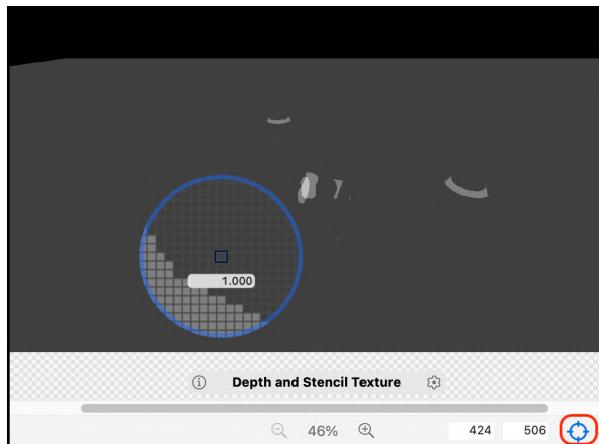
This is the same method to create a depth stencil state object in **RenderPass**, but you'll override it with your stencil configuration.

- Add the following code to `buildDepthStencilState()` before return:

```
let frontFaceStencil = MTLStencilDescriptor()
frontFaceStencil.compareFunction = .always
frontFaceStencil.failureOperation = .keep
frontFaceStencil.depthFailureOperation = .keep
frontFaceStencil.depthStencilPassOperation = .incrementClamp
descriptor.frontFaceStencil = frontFaceStencil
```

`frontFaceStencil` affects the stencil buffer only for models' faces facing the camera. The stencil test will always pass, and nothing happens if the stencil or depth tests fail. If the depth and stencil tests pass, the stencil buffer increases by 1.

- Build and run the app, and choose the Deferred option.  
► Capture the GPU workload and examine the stencil buffer with the magnifier:



*The ground is rendered in front of the trees and sometimes fails the depth test*

Most of the texture is mid-gray with a value of 1. On the trees, which are mostly 1, there are small patches of 2, which incidentally uncovers some inefficient overlapping geometry in the tree model.

It's important to realize that the geometry is processed in the order it's rendered. In `GameScene`, this is set up as:

```
models = [treefir1, treefir2, treefir3, train, ground]
```

The ground is the last to render. It fails the depth test when the fragment is behind a tree or the train and doesn't increment the stencil buffer.

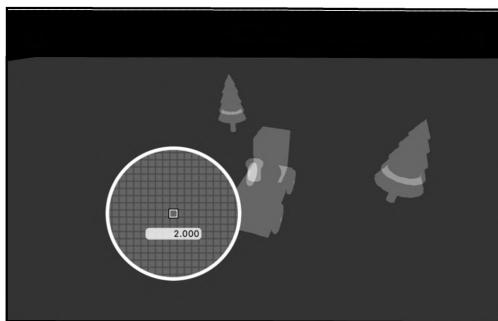
Compare this with a stencil test where the ground is the first to render.

- Open **GameScene.swift**. In `init()`, change the `models` assignment to:

```
models = [ground, treefir1, treefir2, treefir3, train]
```

This code renders the ground first.

- Build and run the app, and choose the Deferred option.
- Capture the GPU workload and compare the stencil texture.



*Ground renders first*

When the tree renders this time, the ground passing the depth test has already incremented the stencil buffer to 1, so the tree passes the depth test and increments the buffer to 2, then 3 when there is extra geometry.

You now have a stencil texture with zero where no geometry renders and non-zero where there is geometry.

All this aims to compute deferred lighting only in those areas with geometry. You can achieve this with your current stencil texture. Where the stencil buffer is zero, you can ignore the fragment in the light render pass.

To achieve this, you'll:

1. Pass in the depth/stencil texture from `GBufferRenderPass` to `LightingRenderPass`.
2. In addition to setting `LightingRenderPass`'s render pass descriptor's stencil attachment, you must assign the depth texture to the descriptor's depth attachment because you combine the stencil texture with depth.
3. `LightingRenderPass` uses two pipeline states: one for the sun and one for point lights. Both must have the depth and stencil pixel format of `depth32float_stencil`.

## 1. Passing in the Depth/Stencil Texture

- Open **LightingRenderPass.swift**, and add a new texture property to **LightingRenderPass**:

```
weak var stencilTexture: MTLTexture?
```

- Add this line to the top of **draw(commandBuffer:scene:uniforms:params:)**:

```
descriptor?.stencilAttachment.texture = stencilTexture
```

- Open **Renderer.swift**. In **draw(scene:in:)**, add this line where you assign the textures to **lightingRenderPass**:

```
lightingRenderPass.stencilTexture = gBufferRenderPass.depthTexture
```

## 2. Setting Up the Render Pass Descriptor

- Open **LightingRenderPass.swift**. At the top of **draw(commandBuffer:scene:uniforms:params:)**, add:

```
descriptor?.depthAttachment.texture = stencilTexture
descriptor?.stencilAttachment.loadAction = .load
descriptor?.depthAttachment.loadAction = .dontCare
```

You set the stencil attachment to load so that the **LightingRenderPass** can use the stencil texture for stencil testing. You don't need the depth texture, so you set a load action of **dontCare**.

## 3. Changing the Pipeline State Objects

- Open **Pipelines.swift**.

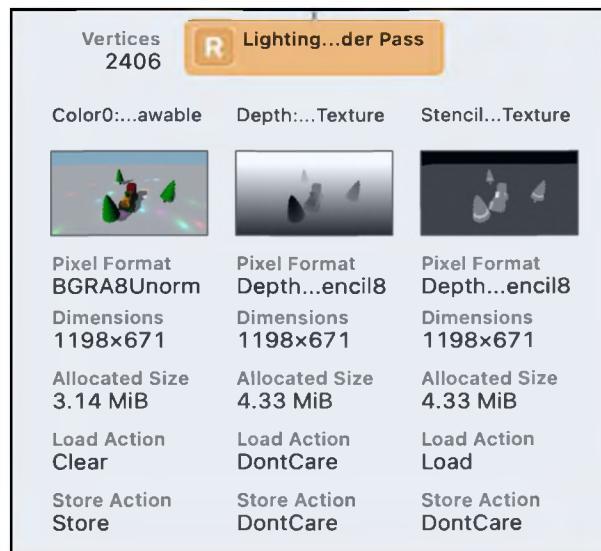
In both **createSunLightPSO(colorPixelFormat:)** and **createPointLightPSO(colorPixelFormat:)**, after **pipelineDescriptor.depthAttachmentPixelFormat = .depth32Float**, add:

```
if !tiled {
    pipelineDescriptor.depthAttachmentPixelFormat
        = .depth32Float_stencil8
    pipelineDescriptor.stencilAttachmentPixelFormat
        = .depth32Float_stencil8
}
```



This code configures the pipeline state to match the render pass descriptor's depth and stencil texture pixel format.

- Build and run the app, and choose the Deferred option.
- Capture the GPU workload and examine the frame so far.



*Stencil texture in frame capture*

`LightingRenderPass` correctly receives the stencil buffer from `GBufferRenderPass`.

## Masking the Sky

When you render the quad in `LightingRenderPass`, you want to bypass all fragments that are zero in the stencil buffer.

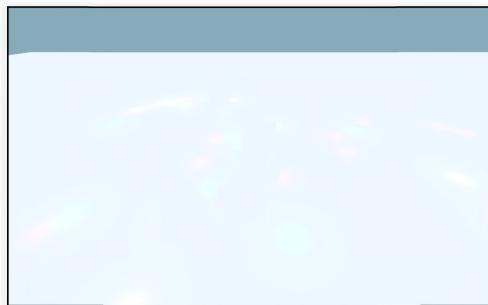
- Open `LightingRenderPass.swift`, and add this code to `buildDepthStencilState()` before return:

```
let frontFaceStencil = MTLStencilDescriptor()
frontFaceStencil.compareFunction = .equal
frontFaceStencil.stencilFailureOperation = .keep
frontFaceStencil.depthFailureOperation = .keep
frontFaceStencil.depthStencilPassOperation = .keep
descriptor.frontFaceStencil = frontFaceStencil
```

(Spoiler: Deliberate mistake. :])

You haven't changed the reference value in the render command encoder, so the reference value is zero. Here, you say that all stencil buffer fragments equal to zero will pass the stencil test. You don't need to change the stencil buffer, so all of the operations are keep.

- Build and run the app, and choose the Deferred option.



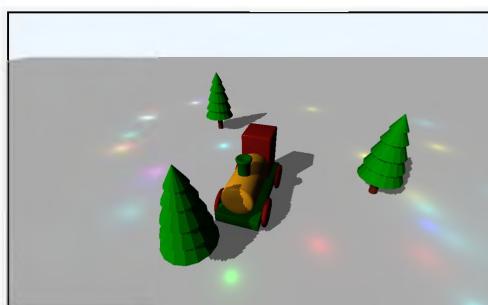
*A deliberate mistake*

In this render, all fragments that are zero render. That's the top part. The bottom section with the plane and trees doesn't render but shows the clear blue sky background. Of course, it should be the other way around.

- In `buildDepthStencilState()`, change the stencil compare function:

```
frontFaceStencil.stencilCompareFunction = .notEqual
```

- Build and run the app, then choose the Deferred option.



*Clear blue skies*

At last, the brooding, stormy sky is replaced by the Metal view's blue `MTLClearColor` that you set way back in `Renderer`'s initializer.

## Challenge

You fixed the sky for your Deferred Rendering pass. Your challenge is now to fix it in the Tiled Deferred render pass. Here's a hint: just follow the steps for the Deferred render pass. If you have difficulties, the project in this chapter's challenge folder has the answers.

## Key Points

- **Tile-based deferred rendering** takes advantage of Apple's special GPUs.
- Keeping data in **tile memory** rather than transferring to system memory is much more efficient and uses less power.
- Mark textures as **memoryless** to keep them in tile memory.
- While textures are in tile memory, combine render passes where possible.
- **Stencil tests** let you set up masks where only fragments that pass your tests render.
- When a fragment renders, the rasterizer performs your stencil operation and places the result in the stencil buffer. With this stencil buffer, you control which parts of your image renders.

## Where to Go From Here?

Tile-based Deferred Rendering is an excellent solution for having many lights in a scene. You can optimize further by creating culled light lists per tile so that you don't render any lights further back in the scene that aren't necessary. Apple's Modern Rendering with Metal 2019 video (<https://apple.co/3mfdtEY>) will help you understand how to do this. The video also points out when to use various rendering technologies.

# Chapter 16: GPU Compute Programming

**General Purpose GPU (GPGPU)** programming uses the many-core GPU architecture to speed up parallel computation. Data-parallel compute processing is useful when you have large chunks of data and need to perform the same operation on each chunk. Examples include machine learning, scientific simulations, ray tracing and image/video processing.

In this chapter, you'll perform some simple GPU programming and explore how to use the GPU in ways other than vertex rendering.



## The Starter Project

- Open Xcode and build and run this chapter's starter project. The scene contains a lonely warrior. The renderer is the forward renderer using your Phong shader.



*The starter project*

From this render, you might think that the warrior is left-handed. Depending on how you render him, he can be ambidextrous.

- Press 1 on your keyboard.

The view changes to the front view. However, the warrior faces towards positive z instead of toward the camera.



*Facing backwards*

The way the warrior renders is due to both math and file formats. In Chapter 6, “Coordinate Spaces”, you learned that this book uses a left-handed coordinate system. Blender exports the obj file for use in a right-handed coordinate system.

If you want a right-handed warrior, there are a few ways to solve this issue:

1. Rewrite all of your coordinate positioning.
2. In `vertex_main`, invert `position.z` when rendering the model.
3. On loading the model, invert `position.z`.

If all of your models are reversed, option #1 or #2 might be good. However, if you only need some models reversed, option #3 is the way to go. All you need is a fast parallel operation. Thankfully, one is available to you using the GPU.

**Note:** Ideally, you would convert the model as part of your model pipeline rather than in your final app. After flipping the vertices, you can write the model out to a new file.

## Winding Order and Culling

Inverting the z position will flip the winding order of vertices, so you may need to consider this. When Model I/O reads in the model, the vertices are in clockwise winding order.

- To demonstrate this, open `ForwardRenderPass.swift`.
- In `draw(commandBuffer:scene:uniforms:params:)`, add this code after `renderEncoder.setRenderPipelineState(pipelineState):`

```
renderEncoder.setFrontFacing(.counterClockwise)
renderEncoder.setCullMode(.back)
```

Here, you tell the GPU to expect vertices in counterclockwise order. The default is clockwise. You also tell the GPU to cull any faces that face away from the camera. As a general rule, you should cull back faces since they're usually hidden, and rendering them isn't necessary.

- Build and run the app.



*Rendering with incorrect winding order*

Because the winding order of the mesh is currently clockwise, the GPU is culling the wrong faces, and the model appears to be inside-out. Rotate the model to see this more clearly. Inverting the z coordinates will correct the winding order.

## Reversing the Model on the CPU

Before working out the parallel algorithm for the GPU, you'll first explore how to reverse the warrior on the CPU. You'll compare the performance with the GPU result. In the process, you'll learn how to access and change Swift data buffer contents with pointers.

- In the **Geometry** group, open **VertexDescriptor.swift**. Take a moment to refresh your memory about the layout in which Model I/O loads the model buffers in `defaultLayout`.

Five buffers are involved, but you're only interested in the first one, `VertexBuffer`. It consists of a `float3` for `Position` and a `float3` for `Normal`. You don't need to consider UVs because they're in the next layout.

- In the **Shaders** group, open **Common.h**, and add a new structure:

```
struct VertexLayout {  
    vector_float3 position;  
    vector_float3 normal;  
};
```

This structure matches the layout in the vertex descriptor for buffer 0. You'll use it to read the loaded mesh buffer.

- In the Game group, open **GameScene.swift**, and add a new method to **GameScene**:

```
mutating func convertMesh(_ model: Model) {
    let startTime = CFAbsoluteTimeGetCurrent()
    for mesh in model.meshes {
        // 1
        let vertexBuffer = mesh.vertexBuffers[VertexBuffer.index]
        let count =
            vertexBuffer.length / MemoryLayout<VertexLayout>.stride
        // 2
        var pointer = vertexBuffer
            .contents()
            .bindMemory(to: VertexLayout.self, capacity: count)
        // 3
        for _ in 0..<count {
            // 4
            pointer.pointee.position.z = -pointer.pointee.position.z
            // 5
            pointer = pointer.advanced(by: 1)
        }
        // 6
        print("CPU Time:", CFAbsoluteTimeGetCurrent() - startTime)
    }
}
```

Here's a code breakdown:

1. First, you find the number of vertices in the vertex buffer. You calculate the number of vertices in the model by dividing the buffer length by the size of the vertex attribute layout. The result should match the number of vertices in the file. There are 6862 for the warrior.
2. `vertexBuffer.contents()` returns a `MTLBuffer`. You **bind** the buffer contents to `pointer`, making `pointer` an `UnsafeMutablePointer<VertexLayout>`.
3. You then iterate through each vertex.
4. The `pointee` is an instance of `VertexLayout`, and you invert the `z` position.
5. You then advance the pointer to the next vertex instance and continue.
6. Finally, you print out the time taken to do the operation.

**Note:** If you're unfamiliar with Swift pointers, read our article [Unsafe Swift: Using Pointers and Interacting With C](#) (<https://bit.ly/3tncI1b>).

- Add this code to the end of `init()` to call the new method:

```
convertMesh(warrior)
```

- Build and run the app.



*A right-handed warrior*

The warrior is now right-handed. On my M1 Mac Mini, the time taken was `0.00179`. That's pretty fast, but the warrior is a small model with only six thousand vertices.

Look for operations you could possibly do in parallel and process with a GPU kernel. Within the `for` loop, you perform the same operation on every vertex **independently**, so it's a good candidate for GPU compute. **Independently** is the critical word, as GPU threads perform operations independently from each other.

## Compute Processing

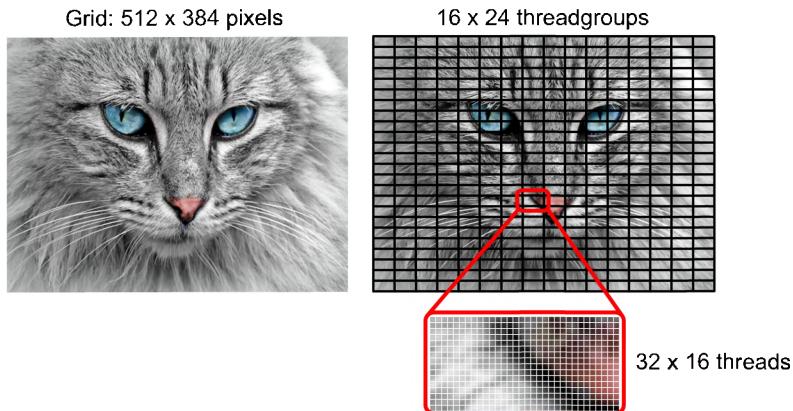
In many ways, compute processing is similar to the render pipeline. You set up a command queue and a command buffer. In place of the render command encoder, compute uses a **compute command encoder**. Instead of using vertex or fragment functions in a compute pass, you use a **kernel function**. **Threads** are the input to the kernel function, and the kernel function operates on each thread.

## Threads and Threadgroups

To determine how many times you want the kernel function to run, you need to know the size of the array, texture or volume you want to process. This size is the **grid** and consists of **threads** organized into **threadgroups**.

The grid is defined in three dimensions: width, height and depth. But often, especially when you're processing images, you'll only work with a 1D or 2D grid. Every point in the grid runs one instance of the kernel function, each on a separate thread.

► Look at the following example image:



*Threads and threadgroups*

The image is  $512 \times 384$  pixels. You need to tell the GPU the number of threads per grid and the number of threads per threadgroup.

**Threads per grid:** In this example, the grid is two dimensions, and the number of threads per grid is the image size of 512 by 384.

**Threads per threadgroup:** Specific to the device, the pipeline state's `threadExecutionWidth` suggests the best width for performance, and `maxTotalThreadsPerThreadgroup` specifies the maximum number of threads in a threadgroup. On a device with 512 as the maximum number of threads, and a thread execution width of 32, the optimal 2d threadgroup size would have a width of 32 and a height of  $512 / 32 = 16$ . So the threads per threadgroup will be 32 by 16.

In this case, the compute dispatch code looks something like this:

```
let threadsPerGrid = MTLSize(width: 512, height: 384, depth: 1)
let width = pipelineState.threadExecutionWidth
let threadsPerThreadgroup = MTLSize(
    width: width,
    height: pipelineState.maxTotalThreadsPerThreadgroup / width,
    depth: 1)
computeEncoder.dispatchThreads(
    threadsPerGrid,
    threadsPerThreadgroup: threadsPerThreadgroup)
```

You specify the threads per grid and let the pipeline state work out the optimal threads per threadgroup.

## Non-uniform Threadgroups

The threads and threadgroups work out evenly across the grid in the previous image example. However, if the grid size isn't a multiple of the threadgroup size, Metal provides **non-uniform threadgroups**.



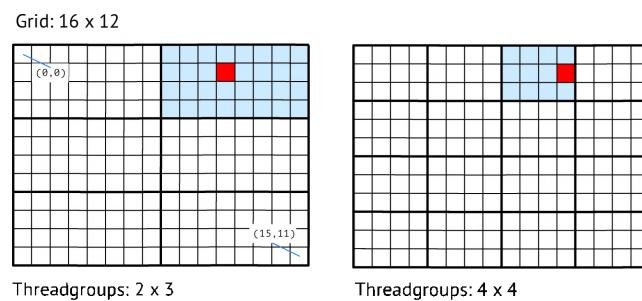
*Non-uniform threadgroups*

Non-uniform threadgroups are only a feature of the Apple GPU family 4 and onwards. The feature set was introduced with A11 devices running iOS 11. A11 chips first appeared in iPhone 8.

## Threadgroups per Grid

You can choose how you split up the grid. Threadgroups have the advantage of executing a group of threads together and also sharing a small chunk of memory. It's common to organize threads into threadgroups to work on smaller parts of the problem independently from other threadgroups.

In the following image, a 16 by 12 grid is split first into  $2 \times 3$  threadgroups and then into  $4 \times 4$  thread groups.



*Threadgroups in a 2D grid*



In the kernel function, you can locate each pixel in the grid. The red pixel in both grids is located at (11, 1).

You can also uniquely identify each thread within the threadgroup. The blue threadgroup on the left is located at (1, 0) and on the right at (2, 0). The red pixels in both grids are threads located within their own threadgroup at (3, 1).

You have control over the number of threadgroups. However, you need to add an extra threadgroup to the size of the grid to make sure at least one threadgroup executes.

Using the cat image example, you could choose to set up the threadgroups in the compute dispatch like this:

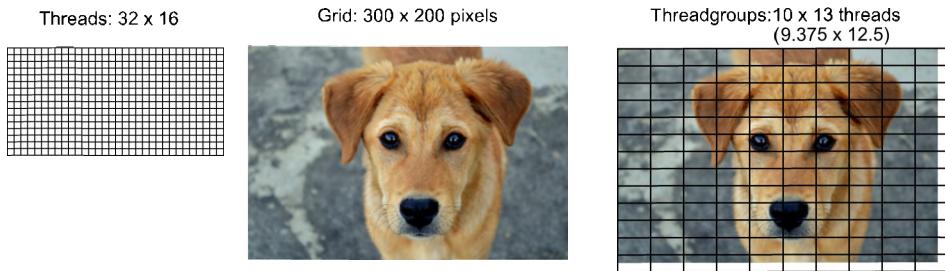
```
let width = 32
let height = 16
let threadsPerThreadgroup = MTLSize(
    width: width, height: height, depth: 1)
let gridWidth = 512
let gridHeight = 384
let threadGroupCount = MTLSize(
    width: (gridWidth + width - 1) / width,
    height: (gridHeight + height - 1) / height,
    depth: 1)
computeEncoder.dispatchThreadgroups(
    threadGroupCount,
    threadsPerThreadgroup: threadsPerThreadgroup)
```

You specify the threads per threadgroup. In this case, the threadgroup will consist of 32 threads wide, 16 threads high and 1 thread deep.

If the size of your data does not match the size of the grid, you may have to perform boundary checks in the kernel function.



In the following example, with a threadgroup size of 32 by 16 threads, the number of threadgroups necessary to process the image would be 10 by 13. You'd have to check that the threadgroup isn't using threads that are off the edge of the image.



*Underutilized threads*

The threads that are off the edge are underutilized. That is, they're threads that you dispatched, but there was no work for them to do.

## Reversing the Warrior Using GPU Compute Processing

The previous example was a two-dimensional image, but you can create grids in one, two or three dimensions. The warrior problem acts on an array in a buffer and will require a one-dimensional grid.

- In the **Geometry** group, open **Model.swift**, and add a new method to **Model**:

```
func convertMesh() {
    // 1
    guard let commandBuffer =
        Renderer.commandQueue.makeCommandBuffer(),
        let computeEncoder =
        commandBuffer.makeComputeCommandEncoder()
    else { return }
    // 2
    let startTime = CFAbsoluteTimeGetCurrent()
    // 3
    let pipelineState: MTLComputePipelineState
    do {
        // 4
        guard let kernelFunction =
            Renderer.library.makeFunction(name: "convert_mesh") else {
            fatalError("Failed to create kernel function")
        }
```

```
// 5
pipelineState = try
    Renderer.device.makeComputePipelineState(
        function: kernelFunction)
} catch {
    fatalError(error.localizedDescription)
}
computeEncoder.setComputePipelineState(pipelineState)
}
```

Going through the code:

1. You create the compute command encoder the same way you created the render command encoder.
2. You add a start time to see how long the conversion takes to execute.
3. For compute processing, you use a compute pipeline state. This requires fewer state changes on the GPU, so you don't need a descriptor.
4. Soon, you'll create the kernel function `convert_mesh`.
5. Finally, you create the pipeline state using the kernel function. You then set the GPU pipeline state in the compute encoder.

► Continue by adding the following code to the bottom of `convertMesh()`:

```
for mesh in meshes {
    let vertexBuffer = mesh.vertexBuffers[VertexBuffer.index]
    computeEncoder.setBuffer(vertexBuffer, offset: 0, index: 0)
    let vertexCount = vertexBuffer.length /
        MemoryLayout<VertexLayout>.stride
}
```

Just as you did in the earlier CPU version, you calculate the vertex count, which is the number of threads required.

## Setting up Threadgroups

► At the bottom and within the `for` loop closure, continue with:

```
let threadsPerGroup = MTLSize(
    width: pipelineState.threadExecutionWidth,
    height: 1,
    depth: 1)
let threadsPerGrid = MTLSize(width: vertexCount, height: 1,
    depth: 1)
computeEncoder.dispatchThreads(
```

```
    threadsPerGrid,  
    threadsPerThreadgroup: threadsPerGroup)  
computeEncoder.endEncoding()
```

You set up the grid and threadgroup the same way as the initial image example. Since your model's vertices are a one-dimensional array, you only set up `width`. Then, you extract the device-dependent thread execution width from the pipeline state to get the number of threads in a thread group. The grid size is the number of vertices in the model.

The dispatch call is the compute command encoder's equivalent to the render command encoder's draw call. The GPU will execute the kernel function specified in the pipeline state when you commit the command buffer.

## Performing Code After Completing GPU Execution

The command buffer can execute a closure after its GPU operations have finished.

- Outside the `for` loop, add this code at the end of `convert_mesh()`:

```
commandBuffer.addCompletedHandler { _ in  
    print(  
        "GPU conversion time:",  
        CFAbsoluteTimeGetCurrent() - startTime)  
}  
commandBuffer.commit()
```

You supply a closure that calculates the amount of time the procedure takes and print it out. You then commit the command buffer to the GPU.

## The Kernel Function

That completes the Swift setup. You simply specify the kernel function to the pipeline state and create an encoder using that pipeline state. With that, it's only necessary to give the thread information to the encoder. The rest of the action takes place inside the kernel function.

- In the **Shaders** group, create a new Metal file named **ConvertMesh.metal**, and add:

```
#import "Common.h"

kernel void convert_mesh(
    device VertexLayout *vertices [[buffer(0)]],
    uint id [[thread_position_in_grid]])
{
    vertices[id].position.z = -vertices[id].position.z;
}
```

A kernel function can't have a return value. Using the `thread_position_in_grid` attribute, you pass in the vertex buffer and identify the thread ID using the `thread_position_in_grid` attribute. You then invert the vertex's z position.

This function will execute for every vertex in the model.

- Open **GameScene.swift**. In `init()`, replace `convertMesh(warrior)` with:

```
warrior.convertMesh()
```

- Build and run the app. Press the 1 key for the front view of the model.



*A right-handed warrior*

The console prints out the time of GPU processing. You've had your first experience with data-parallel processing, and the warrior is now right-handed and faces toward the camera.

Compare the time with the CPU conversion. On my M1 Mac Mini, the GPU conversion time is `0.000692`. Always check the comparative times, as setting up a GPU pipeline is a time cost. It may take less time to perform the operation on the CPU on small operations.

## Atomic Functions

Kernel functions perform operations on individual threads. However, you may want to perform an operation that requires information from other threads. For example, you might want to find out the total number of vertices your kernel worked on.

Your kernel function operates on each thread independently, and these threads update each vertex position simultaneously. If you send the kernel function a variable to store the total in a buffer, the function can increment the total, but other threads will be doing the same thing simultaneously. Therefore you won't get the correct total.

An **atomic operation** works in shared memory and is visible to other threads.

- Open **Model.swift**. In `convertMesh()`, add the following code before `for mesh in meshes:`

```
let totalBuffer = Renderer.device.makeBuffer(  
    length: MemoryLayout<Int>.stride,  
    options: [])  
let vertexTotal = totalBuffer?.contents().bindMemory(to:  
    Int.self, capacity: 1)  
vertexTotal?.pointee = 0  
computeEncoder.setBuffer(totalBuffer, offset: 0, index: 1)
```

Here, you create a buffer to hold the total number of vertices. You bind the buffer to a pointer and set the contents to zero. You then send the buffer to the GPU.

- Still in `convertMesh()`, add this code to the command buffer's completion handler:

```
print("Total Vertices:", vertexTotal?.pointee ?? -1)
```

You print out the contents of the buffer, which contains the total number of vertices.

- Open **ConvertMesh.metal** and add this code to `convert_mesh`'s parameters:

```
device int &vertexTotal [[buffer(1)]],
```

- Then, add this code to the end of the function:

```
vertexTotal++;
```

You add one to `vertexTotal` each time the function executes.

- Build and run the app. Check the debug console output.

```
loaded texture: Warrior-color
loaded texture: Warrior-normal
loaded texture: Warrior-roughness
GPU conversion time: 0.0007570981979370117
Total Vertices: 3
```

Each time you run the app, you'll get a different incorrect result depending on how the GPU scheduled the threads. The threads add one to the buffer at precisely the same time before the buffer can store the values from the other threads.

- Still in `ConvertMesh.metal`, change the `vertexTotal` parameter to:

```
device atomic_int &vertexTotal [[buffer(1)]],
```

Instead of an `int`, you define an `atomic_int`, telling the GPU that this will work in shared memory.

- Replace `vertexTotal++` with:

```
atomic_fetch_add_explicit(&vertexTotal, 1,
    memory_order_relaxed);
```

Since you can't do simple operations on the atomic variable anymore, you call the built-in function that takes in `vertexTotal` as the first parameter and the amount to add as the second parameter.

- Build and run the app.

Now the correct number of vertices prints out in the debug console.

```
loaded texture: Warrior-color
loaded texture: Warrior-normal
loaded texture: Warrior-roughness
GPU conversion time: 0.0007050037384033203
Total Vertices: 6862
```

There are various atomic functions, and you can find out more about them in Chapter 6.15.2, “Atomic Functions” in Apple’s Metal Shading Language Specification (<https://apple.co/3zLTwve>)

With this simple introduction to compute shaders, you’re ready for the next few chapters, where you’ll create stunning particle effects and even control some strange creatures called boids.

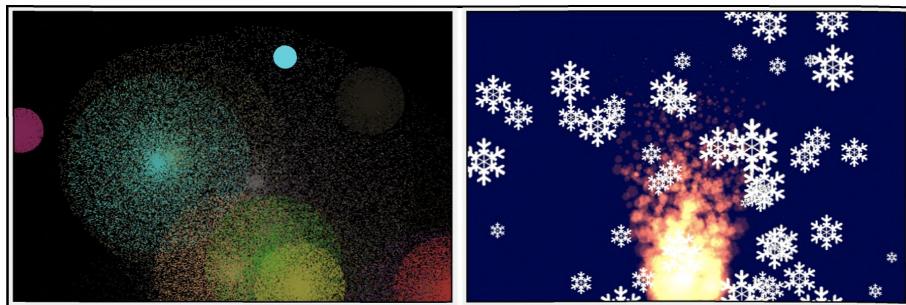
## Key Points

- **GPU compute**, or general purpose GPU programming, helps you perform data operations in parallel without using the more specialized rendering pipeline.
- You can move any task that operates on multiple items independently to the GPU. Later, you'll see that you can even move the repetitive task of rendering a scene to a compute shader.
- GPU memory is good at simple parallel operations, and with Apple Silicon, you can keep chained operations in tile memory instead of moving them back to system memory.
- Compute processing uses a compute pipeline with a kernel function.
- The kernel function operates on a grid of threads organized into **threadgroups**. This grid can be 1D, 2D or 3D.
- **Atomic functions** allow inter-thread operations.

# Chapter 17: Particle Systems

One of the many ways to create art and present science in code is by making use of particles. A **particle** is a tiny graphical object that carries basic information about itself such as color, position, life, speed and direction of movement.

Nothing explains a visual effect better than an image showing what you'll be able to achieve at the end of this chapter:



*Particle systems*

Particle systems are widely used in:

- Video games and animation: hair, cloth, fur.
- Modeling of natural phenomena: fire, smoke, water, snow.
- Scientific simulations: galaxy collisions, cellular mitosis, fluid turbulence.

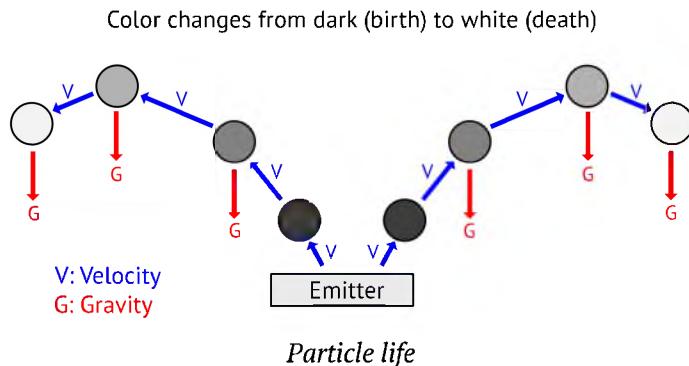
**Note:** William Reeves is credited as being the “father” of particle systems. While at Lucasfilm, Reeves created the *Genesis Effect* in 1982 while working on the movie *Star Trek II: The Wrath of Khan*. Later, he joined Pixar Animation Studios where he’s still creating amazing animations using particles.

In a moment, you’ll get your feet wet trying out one such practical application: fireworks. But first, what exactly is a particle?

## Particle

Newtonian dynamics describe the relationship between any small body – a particle – and the forces acting upon it, as well as its motion in response to those forces. Newton’s three laws of motion define the relationship between them. The first two laws define motion as a result of either inertia or force interference upon the particle’s current state of motion (stationary or moving). You’ll be working with them in this chapter. The third law, however, defines motion as a reaction of two or more particles interacting with each other. You’ll work with this law in Chapter 18, “Particle Behavior”.

A fourth law, if you wish, is the law of life. It’s not one of the Newtonian motion laws, but it does indeed apply to particles. Particles are born. They move and interact with the environment, and then they die.



You need a particle system to create fireworks. But first, you need to define a particle that has – at a minimum – a position, direction, speed, color and life. What makes a particle system cohesive, though, are emitters.

## Emitter

An **emitter** is nothing more than a particle generator — in other words, a source of particles. You can make your particle system more exciting by having several emitters shooting out particles from different positions.

Fireworks are sequential explosions that occur a few seconds apart from each other, so you will create several emitters.

## The Starter Project

- In Xcode, open the starter project for this chapter, then build and run it.



*The starter project*

The project contains an empty scene with a dark blue clear screen, rendering with the forward renderer that you're already familiar with.

- The **Particles** group contains a few files that you'll use later in the chapter.
- In the **Textures** group, **Textures.xcassets** contains two sprite textures that you'll attach to particle systems.
- **Renderer** is already set up with skeleton structures to render fireworks and particles.
- **PipelineStates** contains a factory method to create a particle render pipeline state.
- The **Fireworks** group is where you'll create your first particle system.

## Creating a Particle and Emitter

- In the **Shaders** group, open **Common.h**, and add a new structure:

```
struct Particle {  
    vector_float2 position;  
    float direction;  
    float speed;  
    vector_float4 color;  
    float life;  
};
```

This structure holds the properties for each particle.

- In the **Fireworks** group, create a new Swift file called **FireworksEmitter.swift**, and replace the code with this:

```
import MetalKit  
  
struct FireworksEmitter {  
    let particleBuffer: MTLBuffer  
  
    init(  
        particleCount: Int,  
        size: CGSize,  
        life: Float  
    ) {  
        let bufferSize =  
            MemoryLayout<Particle>.stride * particleCount  
        particleBuffer =  
            Renderer.device.makeBuffer(length: bufferSize)!  
    }  
}
```

This code takes in the particle count and the particle's lifetime. You then create the buffer that will hold all the particle data.

You'll randomize the initial position and color once for all particles, and then randomize each particle's speed and direction based on the size of the display area.

- Add the following code to `init(particleCount:size:life:)`:

```
let width = Float(size.width)  
let height = Float(size.height)  
let position = float2(  
    Float.random(in: 0...width),  
    Float.random(in: 0...height))  
let color = float4(  
    Float.random(in: 0...life) / life,
```



```
Float.random(in: 0...life) / life,  
Float.random(in: 0...life) / life,  
1)
```

You set up initial properties for the emitter. Each firework emitter will have a random position where all the particles will start. All particles will have the same random color.

- Continue adding the following code after the code you just added:

```
var pointer =  
    particleBuffer.contents().bindMemory(  
        to: Particle.self,  
        capacity: particleCount)  
for _ in 0..    let direction =  
        2 * Float.pi * Float.random(in: 0...width) / width  
    let speed = 3 * Float.random(in: 0...width) / width  
    pointer.pointee.position = position  
    pointer.pointee.direction = direction  
    pointer.pointee.speed = speed  
    pointer.pointee.color = color  
    pointer.pointee.life = life  
    pointer = pointer.advanced(by: 1)  
}
```

Here, you loop through the buffer using a pointer to access each particle object and set their properties.

You've now created lots of particles, each with distinct direction and speed. All of them have the same original position, color and life.

- In the **Fireworks** group, open **Fireworks.swift**.

This file contains two methods that you'll fill out shortly. Renderer calls these on each frame.

- Add these properties to **Fireworks**:

```
let particleCount = 10000  
let maxEmitters = 8  
var emitters: [FireworksEmitter] = []  
let life: Float = 256  
var timer: Float = 0
```

You'll set up an array of emitters, each with 10,000 particles.

- Add this code to `update(size:)`:

```
timer += 1
if timer >= 50 {
    timer = 0
    if emitters.count > maxEmitters {
        emitters.removeFirst()
    }
    let emitter = FireworksEmitter(
        particleCount: particleCount,
        size: size,
        life: life)
    emitters.append(emitter)
}
```

You reset a timer variable every time it reaches a threshold (50 in this case). At that point, you add a new emitter and then remove the oldest one.

Build and run the app to verify everything is working. Note, however, you'll see the same solid dark blue color as you did in the beginning as you're not doing any drawing yet.

The shader function is where you'll update the particles' life and position. Each of the particles is updated independently, and so will work well with the granular control of GPU threads in compute encoding!

## The Compute Pipeline State Object

- Open `Pipelines.swift` and add this to `PipelineStates`:

```
static func createComputePSO(function: String)
    -> MTLComputePipelineState {
    guard let kernel = Renderer.library.makeFunction(name:
function)
    else { fatalError("Unable to create \(function) PSO") }
    let pipelineState: MTLComputePipelineState
    do {
        pipelineState =
            try Renderer.device.makeComputePipelineState(function:
kernel)
        } catch {
            fatalError(error.localizedDescription)
        }
        return pipelineState
    }
```

As you learned in the previous chapter, for a compute pipeline state, you don't need a pipeline state descriptor. You simply create the pipeline state directly from the kernel function. You'll be able to use this method to create compute pipeline state objects using only the name of the kernel.

## The Fireworks Pass

- Open **Fireworks.swift**, and add the pipeline states and initializer:

```
let clearScreenPSO: MTLComputePipelineState
let fireworksPSO: MTLComputePipelineState

init() {
    clearScreenPSO =
        PipelineStates.createComputePSO(function: "clearScreen")
    fireworksPSO =
        PipelineStates.createComputePSO(function: "fireworks")
}
```

When you display your particles on the screen, you'll write them to the drawable texture. Because you're not going through the render pipeline, you will need to clear the texture to a night sky black. You'll do this in a simple compute shader.

**Note:** In this particular application, because you're using the drawable texture, you could set the initial metal view's clear color to black instead of running a clear screen kernel function. But clearing the drawable texture will give you practice in using a 2D grid, and the skill to clear other textures too.

After clearing the screen, you'll then be able to calculate the fireworks particles.

You will require a different kernel function for each process, requiring two different pipeline state objects.



## Clearing the Screen

The clear screen kernel function will run on every pixel in the drawable texture. The texture has a width and height, and is therefore a two dimensional grid.

- Still in **Fireworks.swift**, add this to `draw(commandBuffer:view:)`:

```
// 1
guard let computeEncoder =
    commandBuffer.makeComputeCommandEncoder(),
    let drawable = view.currentDrawable
    else { return }
computeEncoder.setComputePipelineState(clearScreenPS0)
computeEncoder.setTexture(drawable.texture, index: 0)
// 2
var threadsPerGrid = MTLSize(
    width: Int(view.drawableSize.width),
    height: Int(view.drawableSize.height),
    depth: 1)
// 3
let width = clearScreenPS0.threadExecutionWidth
var threadsPerThreadgroup = MTLSize(
    width: width,
    height: clearScreenPS0.maxTotalThreadsPerThreadgroup / width,
    depth: 1)
// 4
computeEncoder.dispatchThreads(
    threadsPerGrid,
    threadsPerThreadgroup: threadsPerThreadgroup)
computeEncoder.endEncoding()
```

Going through the code:

1. You create the compute command encoder and set its pipeline state. You also send the drawable texture to the GPU at index 0 for writing.
2. The grid size is the drawable's width by the drawable's height.
3. Using the pipeline state thread values, you calculate the number of threads per threadgroup.
4. You dispatch these threads to do the work in parallel on the GPU.

- In the **Shaders** group, create a new Metal file named **Fireworks.metal**, and add the kernel functions you set up in your pipeline states:

```
#import "Common.h"

kernel void clearScreen(
```

```
texture2d<half, access::write> output [[texture(0)]],  
uint2 id [[thread_position_in_grid]]  
{  
    output.write(half4(0.0, 0.0, 0.0, 1.0), id);  
}  
  
kernel void fireworks()  
{ }
```

`clearScreen` takes, as arguments, the drawable texture you sent from the CPU and the 2D thread index. You specify in the parameter that you will need write access. Then, you write the color black into the drawable texture for each thread/pixel.

The kernel function's `id` parameter uses the `[[thread_position_in_grid]]` attribute qualifier which uniquely locates a thread within the compute grid and enables it to work distinctly from the others. In this case, it identifies each pixel in the texture.

You'll return and fill out `fireworks` shortly.

► Open `Renderer.swift`, and add this at the end of `init(metalView:options:)`:

```
metalView.framebufferOnly = false
```

Because you're writing to the view's drawable texture in `clearScreen`, you need to change the underlying render target usage by enabling writing to the frame buffer.

► Take a look at `draw(scene:in:)`.

`Renderer` is already set up to call `fireworks(update:size:)` and `fireworks.draw(commandBuffer:view:)`.

► Build and run the app.

`clearScreen` writes the color to the view's drawable texture, so you'll finally see the view color turns from blue to a night sky ready to display your fireworks.



*Drawable cleared to black*

## Dispatching the Particle Buffer

Now that you've cleared the screen, you'll set up a new encoder to dispatch the particle buffer to the GPU.

► Open `Fireworks.swift`, and add this to the end of `draw(commandBuffer:view:)`:

```
// 1
guard let particleEncoder =
    commandBuffer.makeComputeCommandEncoder()
    else { return }
particleEncoder.setComputePipelineState(fireworksPSO)
particleEncoder.setTexture(drawable.texture, index: 0)
// 2
threadsPerGrid = MTLSize(width: particleCount, height: 1, depth:
1)
for emitter in emitters {
    // 3
    let particleBuffer = emitter.particleBuffer
    particleEncoder.setBuffer(particleBuffer, offset: 0, index: 0)
    threadsPerThreadgroup = MTLSize(
        width: fireworksPSO.threadExecutionWidth,
        height: 1,
        depth: 1)
    particleEncoder.dispatchThreads(
        threadsPerGrid,
        threadsPerThreadgroup: threadsPerThreadgroup)
}
particleEncoder.endEncoding()
```

This code is very similar to the previous compute encoder setup:

1. You create a second command encoder and set the particle pipeline state and drawable texture to it.
2. You change the dimensionality from 2D to 1D and set the number of threads per grid to equal the number of particles.
3. You dispatch threads for each emitter in the array.

Since your `threadsPerGrid` is now 1D, you'll need to match `[[thread_position_in_grid]]` in the shader kernel function with a `uint` parameter. Threads will not be dispatched for each pixel anymore but rather for each particle, so `[[thread_position_in_grid]]` in this case will only affect a particular pixel in the drawable texture if there is a particle positioned at that pixel.

All right, time for some physics chatter!

## Particle Dynamics

Particle dynamics makes heavy use of Newton's laws of motion. Particles are considered to be small objects approximated as point masses. Since volume is not something that characterizes particles, scaling or rotational motion will not be considered. Particles will, however, make use of translation motion so they'll always need to have a position.

Besides a position, particles might also have a direction and speed of movement (velocity), forces that influence them (e.g., gravity), a mass, a color and an age. Since the particle footprint is so small in memory, modern GPUs can generate 4+ million particles, and they can follow the laws of motion at 60 fps.

For now, you're going to ignore gravity, so its value will be  $0$ . Time in this example won't change so it will have a value of  $1$ . As a consequence, velocity will always be the same. You can also assume the particle mass is always  $1$ , for convenience.

To calculate the position, you use this formula:

$$x_2 = x_1 + v_1 t + \frac{1}{2} a t^2$$

Where  $x_2$  is the new position,  $x_1$  is the old position,  $v_1$  is the old velocity,  $t$  is time, and  $a$  is acceleration.

This is the formula to calculate the new velocity from the old one:

$$v_2 = v_1 + a t$$

However, since the acceleration is  $0$  in the fireworks example, the velocity will always have the same value.

As you might remember from Physics class, the formula for velocity is:

```
velocity = speed * direction
```

Plugging all this information into the first formula above gives you the final equation to use in the kernel.

Again, as for the second formula, since acceleration is  $0$ , the last term cancels out:

```
newPosition = oldPosition * velocity
```

Finally, you're creating exploding fireworks, so your firework particles will move in a circle that keeps growing away from the initial emitter origin, so you need to know the equation of a circle:

$$\begin{aligned}x &= a + r \cos t \\y &= b + r \sin t\end{aligned}$$

Using the angle that the particle direction makes with the axes, you can re-write the velocity equation from the parametric form of the circle equation using the trigonometric functions *sine* and *cosine* as follows:

```
xVelocity = speed * cos(direction)
yVelocity = speed * sin(direction)
```

Great! Why don't you write all this down in code now?

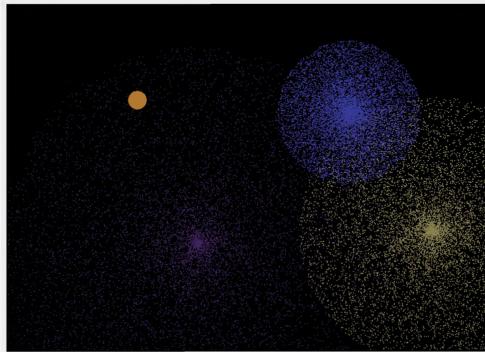
## Implementing Particle Physics

- Open `Fireworks.metal` and replace `fireworks` with:

```
kernel void fireworks(
    texture2d<half, access::write> output [[texture(0)]],
    // 1
    device Particle *particles [[buffer(0)]],
    uint id [[thread_position_in_grid]]) {
    // 2
    float xVelocity = particles[id].speed
        * cos(particles[id].direction);
    float yVelocity = particles[id].speed
        * sin(particles[id].direction) + 3.0;
    particles[id].position.x += xVelocity;
    particles[id].position.y += yVelocity;
    // 3
    particles[id].life -= 1.0;
    half4 color;
    color = half4(particles[id].color) * particles[id].life /
255.0;
    // 4
    color.a = 1.0;
    uint2 position = uint2(particles[id].position);
    output.write(color, position);
    output.write(color, position + uint2(0, 1));
    output.write(color, position - uint2(0, 1));
    output.write(color, position + uint2(1, 0));
    output.write(color, position - uint2(1, 0));
}
```

Going through this code:

1. Get the particle buffer from the CPU and use a 1D index to match the grid dimensions you dispatched earlier.
  2. Compute the velocity and update the position for the current particle according to the laws of motion and using the circle equation as explained above.
  3. Update the life variable and compute a new color after each update. The color will fade as the value held by the life variable gets smaller and smaller.
  4. Write the updated color at the current particle position, as well as its neighboring particles to the left, right, top and bottom to create the look and feel of a thicker particle.
- Build and run the app, and finally you get to enjoy some fireworks!



*Fireworks!*

You can improve the realism of particle effects in at least a couple of ways. One of them is to attach a sprite or a texture to each particle in a render pass. Instead of a dull point, you'll then be able to see a textured point which looks way more lively.

You can practice this technique in the next particle endeavor: a snowing simulation.

## Particle Systems

The fireworks particle system was tailor-made for fireworks. However, particle systems can be very complex with many different options for particle movement, colors and sizes. In the **Particles** group, the `Emitter` class in the starter project is a simple example of a generic particle system where you can create many different types of particles using a particle descriptor.

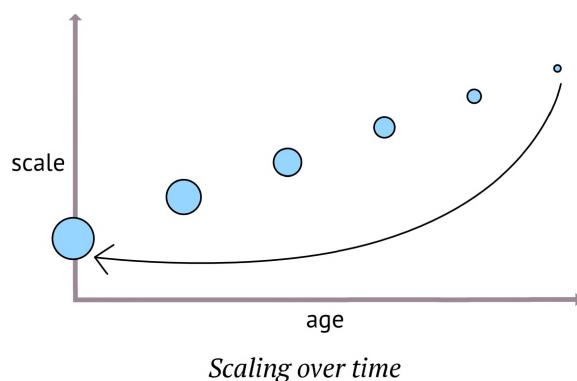
For example, you're going to create snow falling, but also a fire blazing upwards. These particle systems will have different speeds, textures and directions.

► In the **Particles** group, open **ParticleEffects.swift**, and examine the code.

To create a particle system, you first create a descriptor which describes all the characteristics of each individual particle.

Many of the properties in **ParticleDescriptor** are **ClosedRanges**. For example, as well as position, there is a **positionXRange** and **positionYRange**. This allows you to specify a starting position but also allows randomness within limits. If you specify a position of [10, 0], and a **positionXRange** of 0...180, then each particle will be within the range of 10 to 190.

Each particle has a **startScale** and an **endScale**. By setting the **startScale** to 1 and the **endScale** to 0, you can make the particle get smaller over its lifespan.



► Open **Emitter.swift**.

When you create the emitter, as well as the particle descriptor, you also specify:

- **texture**: You'll render this texture for each particle using the particle coordinates.
- **particleCount**: The total number of particles.
- **birthRate**: How many particles should generate at one time.
- **birthDelay**: Delays particle generation. This allows you to slowly release particles (like a gentle snow flurry) or send them out more quickly (like a blazing fire).
- **blending**: Some particle effects require blending, such as you did for your point lights in Chapter 14, "Deferred Rendering".

Emitter creates a buffer the size of all the particles. `emit()` processes each new particle and creates it with the particle settings you set up in the particle descriptor.

More complex particle systems would maintain a **live** buffer and a **dead** buffer. As particles die, they move from **live** to **dead**, and as the system requires new particles, it recovers them from **dead**. However, in this more simple system, a particle never dies. As soon as a particle's age reaches its life-span, it's reborn with the values it started with.

## Resetting the Scene

To add this new, more generic particle system, remove your fireworks simulation from Renderer.

- Open `Renderer.swift` and remove:

```
var fireworks: Fireworks
```

- In `init(metalView:options:)`, remove:

```
fireworks = Fireworks()
```

- In `draw(scene:in:)`, remove:

```
// Render Fireworks with compute shaders
fireworks.update(size: view.drawableSize)
fireworks.draw(commandBuffer: commandBuffer, view: view)
```

- Build and run the app, and your screen will show the metal view's initial clear color.



*Reset project*

Your game may require many different particle effects, so you'll add them to the game scene.

- Open **GameScene.swift**, and add a new property to **GameScene**:

```
var particleEffects: [Emitter] = []
```

Each particle effect will be an emitter, which you'll add to this array.

## Updating the Particle Structure

With a more complex particle system, you need to store more particle properties.

- Open **Common.h**, and add this to the end of **Particle**:

```
float age;
float size;
float scale;
float startScale;
float endScale;
vector_float2 startPosition;
```

You'll now be able to decay the size of the particle over time.

- Open **Emitter.swift** and remove the **Particle** structure. **Emitter** will now use the structure in **Common.h**.

## Rendering a Particle System

You'll attach a texture to snow particles to improve the realism of your rendering. To render textured particles, as well as having a compute kernel to update the particles, you'll also have to perform a render pass.

- In the **Render Passes** group, open **ParticlesRenderPass.swift**.

This is a skeleton render pass with the minimum requirements to conform to **RenderPass**. It's already set up to render in **Renderer**.

`draw(commandBuffer:scene:uniforms:params:)` first calls `update(commandBuffer:scene:)`. This is where you'll first create the compute pipeline to update the particles. You'll then build the pipeline for rendering the particles in `render(commandBuffer:scene:)`.

- In ParticlesRenderPass, create the necessary pipeline state properties and replace init(view:):

```
let computePSO: MTLComputePipelineState
let renderPSO: MTLRenderPipelineState
let blendingPSO: MTLRenderPipelineState

init(view: MTKView) {
    computePSO = PipelineStates.createComputePSO(
        function: "computeParticles")
    renderPSO = PipelineStates.createParticleRenderPSO(
        pixelFormat: view.colorPixelFormat)
    blendingPSO = PipelineStates.createParticleRenderPSO(
        pixelFormat: view.colorPixelFormat,
        enableBlending: true)
}
```

computeParticles is the kernel function where you'll update the particles every frame.

The two render pipeline states will use the vertex function vertex\_particle to position a point on the screen, and fragment\_particle to draw the sprite texture into the point.

- Add this code to update(commandBuffer:scene:):

```
// 1
guard let computeEncoder =
    commandBuffer.makeComputeCommandEncoder() else { return }
computeEncoder.label = label
computeEncoder.setComputePipelineState(computePSO)
// 2
let threadsPerGroup = MTLSize(
    width: computePSO.threadExecutionWidth, height: 1, depth: 1)
// 3
for emitter in scene.particleEffects {
    emitter.emit()
    if emitter.currentParticles <= 0 { continue }
    // 4
    let threadsPerGrid = MTLSize(
        width: emitter.particleCount, height: 1, depth: 1)
    computeEncoder.setBuffer(
        emitter.particleBuffer,
        offset: 0,
        index: 0)
    computeEncoder.dispatchThreads(
        threadsPerGrid,
        threadsPerThreadgroup: threadsPerGroup)
}
computeEncoder.endEncoding()
```



Going through the code:

1. You create the compute command encoder and set the pipeline state.
2. You use the pipeline state's default thread execution width.
3. You process each emitter in the scene.
4. The 1D grid is the number of particles in the emitter.

**Note:** The previous code, which may create non-uniform threadgroup sizes, will only work on macOS, and iOS devices included in Apple 4 and later GPUs. In addition, Simulator does not support non-uniform threadgroup sizes.

► Add this code to `render(commandBuffer:scene:)`:

```
guard let descriptor = descriptor,
       let renderEncoder =
           commandBuffer.makeRenderCommandEncoder(descriptor:
descriptor)
else { return }
renderEncoder.label = label
var size: float2 = [Float(size.width), Float(size.height)]
renderEncoder.setVertexBytes(
    &size,
    length: MemoryLayout<float2>.stride,
    index: 0)
```

`render(commandBuffer:scene:)` will contain familiar rendering code. Renderer supplies `ParticlesRenderPass` with the view's current render pass descriptor before calling the `draw` method. You then create the render encoder using this descriptor.

`size` contains the view's drawable size which the vertex shader will require. Renderer calls `resize(view:size:)` to update `size` when the view size changes.

► Continue by adding the following:

```
// 1
for emitter in scene.particleEffects {
    if emitter.currentParticles <= 0 { continue }
    renderEncoder.setRenderPipelineState(
        emitter.blending ? blendingPSO : renderPSO)
// 2
    renderEncoder.setVertexBuffer(
        emitter.particleBuffer,
```



```

        offset: 0,
        index: 1)
    renderEncoder.setVertexBytes(
        &emitter.position,
        length: MemoryLayout<float2>.stride,
        index: 2)
    renderEncoder.setFragmentTexture(
        emitter.particleTexture,
        index: 0)
    // 3
    renderEncoder.drawPrimitives(
        type: .point,
        vertexStart: 0,
        vertexCount: 1,
        instanceCount: emitter.currentParticles)
}
renderEncoder.endEncoding()

```

Going through the code:

1. For each emitter in the scene, you set the pipeline state object depending upon whether the emitter's particles require blending.
2. You send the emitter particle buffer and the emitter's position to the vertex shader. You also set the texture that all the particles will use.
3. You draw a point primitive for each particle. Each particle is an instance, so you set `instanceCount` to the number of live particles.

## The Vertex and Fragment Functions

All right, time to configure the shader functions.

- In the **Shaders** group, create a new Metal file named **Particles.metal**, and add the following code:

```

#import "Common.h"

// 1
kernel void computeParticles(
    device Particle *particles [[buffer(0)]],
    uint id [[thread_position_in_grid]])
{
    // 2
    float xVelocity = particles[id].speed
        * cos(particles[id].direction);
    float yVelocity = particles[id].speed
        * sin(particles[id].direction);
    particles[id].position.x += xVelocity;
}

```

```

particles[id].position.y += yVelocity;
// 3
particles[id].age += 1.0;
float age = particles[id].age / particles[id].life;
particles[id].scale = mix(particles[id].startScale,
                           particles[id].endScale, age);
// 4
if (particles[id].age > particles[id].life) {
    particles[id].position = particles[id].startPosition;
    particles[id].age = 0;
    particles[id].scale = particles[id].startScale;
}
}

```

This code:

1. Takes in the particle buffer as a shader function argument and sets a 1D index to iterate over this buffer.
2. Calculates the velocity for each particle and then updates its position by adding the velocity to it.
3. Updates the particle's age and scales the particle depending on its age.
4. If the particle's age has reached its total lifespan, reset the particle to its original properties.

► Add a new structure for the vertex and fragment shaders:

```

struct VertexOut {
    float4 position [[position]];
    float point_size [[point_size]];
    float4 color;
};

```

► Add the vertex function to process the particle position and color:

```

// 1
vertex VertexOut vertex_particle(
    constant float2 &size [[buffer(0)]],
    const device Particle *particles [[buffer(1)]],
    constant float2 &emitterPosition [[buffer(2)]],
    uint instance [[instance_id]])
{
    // 2
    float2 position = particles[instance].position
        + emitterPosition;
    VertexOut out {
        // 3
        .position =

```

```
    float4(position.xy / size * 2.0 - 1.0, 0, 1),
// 4
    .point_size = particles[instance].size
        * particles[instance].scale,
    .color = particles[instance].color
};
return out;
}
```

Going through this code:

1. Get the drawable size from the CPU as well as the updated particle buffer and the emitter's position, and use a 1D index to iterate over all particles.
2. Offset the particle position by the emitter's position.
3. Map the particle positions from a [0, 1] range to a [-1, 1] range so that the middle of the screen is now the origin (0, 0).
4. Set the particle's point size and color.

► Add the fragment shader:

```
// 1
fragment float4 fragment_particle(
    VertexOut in [[stage_in]],
    texture2d<float> particleTexture [[texture(0)]],
    float2 point [[point_coord]])
{
// 2
    constexpr sampler default_sampler;
    float4 color = particleTexture.sample(default_sampler, point);
// 3
    if (color.a < 0.5) {
        discard_fragment();
    }
// 4
    color = float4(color.xyz, 0.5);
    color *= in.color;
    return color;
}
```



Going through this code:

1. Get the processed particle fragments via `[[stage_in]]` and the snowflake texture from the CPU. The `[[point_coord]]` attribute is generated by the rasterizer and is a 2D coordinate that indicates where the current fragment is located within a point primitive in a `[0, 1]` range.
2. Create a sampler and use it to sample from the given texture at the current fragment position.
3. Apply alpha testing so you don't render the fragment at low alpha values.
4. Return the texture color combined with the particle color.

## Configuring Particle Effects

The particle computing and rendering structure is complete. All you have to do now is configure an emitter for snow particles.

► Open `ParticleEffects.swift`.

You'll create a factory method for each particle effect. In it, you'll set up a particle descriptor and return an `Emitter`.

A fire emitter is already set up in `ParticleEffects`, giving you a preview of what your snow emitter is going to look like.

► Add this to `ParticleEffects`:

```
static func createSnow(size: CGSize) -> Emitter {
    // 1
    var descriptor = ParticleDescriptor()
    descriptor.positionXRange = 0...Float(size.width)
    descriptor.direction = -.pi / 2
    descriptor.speedRange = 2...6
    descriptor.pointSizeRange = 80 * 0.5...80
    descriptor.startScale = 0
    descriptor.startScaleRange = 0.2...1.0
    // 2
    descriptor.life = 500
    descriptor.color = [1, 1, 1, 1]
    // 3
    return Emitter(
        descriptor,
        texture: "snowflake",
        particleCount: 100,
        birthRate: 1,
```



```
        birthDelay: 20)  
    }
```

Here's what's happening:

1. The descriptor describes how to initialize each particle. You set up ranges for position, speed and scale. Particles will appear at the top of the screen in random positions.
2. A particle has an age and a life-span. A snowflake particle will remain alive for 500 frames and then recycle. You want the snowflake to travel from the top of the screen all the way down to the bottom of the screen. `life` has to be long enough for this to happen. If you give your snowflake a short life, it will disappear while still on screen.
3. You tell the emitter how many particles in total should be in the system. `birthRate` and `birthDelay` control how fast the particles emit. With these parameters, you'll emit one snowflake every twenty frames until there are 100 snowflakes in total. If you want a blizzard rather than a few flakes, then you can set the birthrate higher and the delay between each emission less.

Particle parameters are really fun to experiment with. Once you have your snowflakes falling, change any of these parameters to see what the effect is.

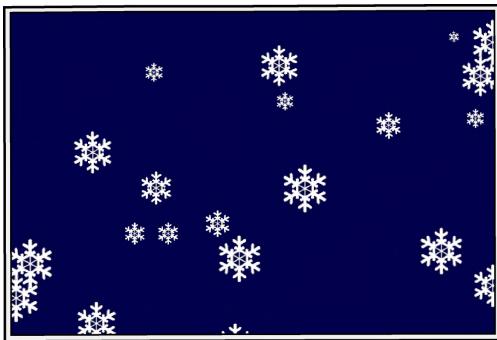
► Open `GameScene.swift`, and add this to `update(size: CGSize)`:

```
let snow = ParticleEffects.createSnow(size: size)  
snow.position = [0, Float(size.height) + 100]  
particleEffects = [snow]
```

You set up the emitter off the top of the screen, so that particles don't suddenly pop in. You then add it to the scene's list of particles. Whenever the view resizes, you will reset the particle effects to fit in the new size.



- Build and run the app, and enjoy the relaxing snow with variable snowflake speeds and sizes:



A snow particle system

Go back and experiment with some of the particle settings. With `particleCount` of 800, `birthDelay` of 2 and `speedRange` of 4...8, you start off with a gentle snowfall that gradually turns into a veritable blizzard.

## Fire

Brrr. That snow is so cold, you need a fire.

- Open `GameScene.swift`, and replace `particleEffects = [snow]` with:

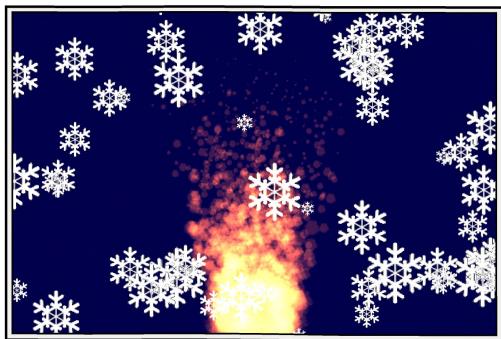
```
let fire = ParticleEffects.createFire(size: size)
fire.position = [0, 0]
particleEffects = [snow, fire]
```

This positions the emitter just off the bottom of the screen.

- Open **ParticleEffects.swift** and examine the `createFire(size:)` settings, and see if you can work out what the particle system will look like.

You're loading more particles than for snow, and a different texture. The birth rate is higher, and there's no delay. The direction is upwards, with a slight variation in range. The particle scales down over its life. The color is fiery orange with blending enabled.

- Build and run to see this new particle system in action.



*Fire and snow*

## Key Points

- Particle emitters emit particles. These particles carry information about themselves, such as position, velocity and color.
- Particle attributes can vary over time. A particle may have a life and decay after a certain amount of time.
- As each particle in a particle system has the same attributes, the GPU is a good fit for updating them in parallel.
- Particle systems, depending on given attributes, can simulate physics or fluid systems, or even hair and grass systems.

## Where to Go From Here?

You've only just begun playing with particles. There are many more particle characteristics you could include in your particle system:

- Color over life.
- Gravity.
- Acceleration.
- Instead of scaling linearly over time, how about scaling slowly then faster?

If you want more ideas, review the links in this chapter's **references.markdown**.

There are some things you haven't yet looked at, like collisions or reaction to acting forces. You have also not read anything about intelligent agents and their behaviors. You'll learn more about all this next, in Chapter 18, "Particle Behavior".

# Chapter 18: Particle Behavior

As you learned in the previous chapter, particles have been at the foundation of computer animation for years. In computer graphics literature, three major animation paradigms are well defined and have rapidly evolved in the last two decades:

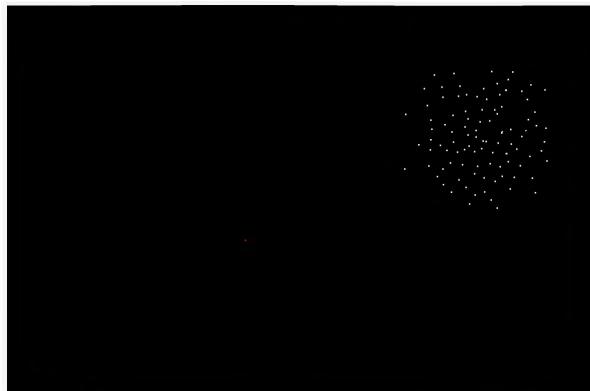
- **Keyframe animation:** Starting parameters are defined as initial frames, and then an interpolation procedure is used to fill the remaining values for in-between frames. You'll cover this topic in Chapter 23, "Animation".
- **Physically based animation:** Starting values are defined as animation parameters, such as a particle's initial position and velocity, but intermediate values are not specified externally. This topic was covered in Chapter 17, "Particle Systems".
- **Behavioral animation:** Starting values are defined as animation parameters. In addition, a cognitive process model describes and influences the way intermediate values are later determined.

In this chapter, you'll focus on the last paradigm as you work through:

- Velocity and bounds checking.
- Swarming behavior.
- Behavioral animation.
- Behavioral rules.



By the end of the chapter, you'll build and control a swarm exhibiting basic behaviors you might see in nature.



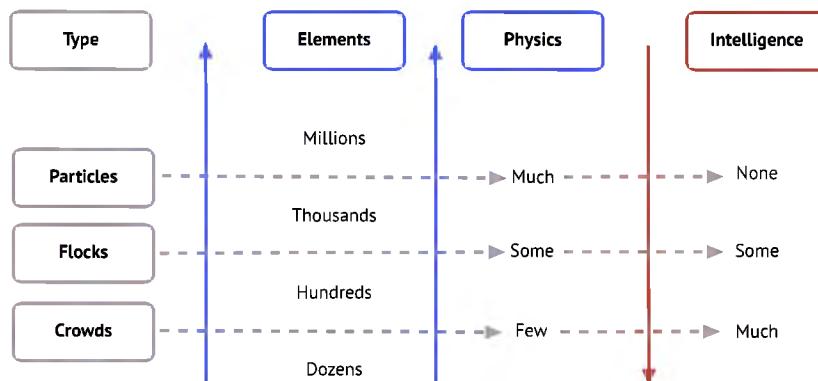
## Behavioral Animation

You can broadly split behavioral animation into two major categories:

- **Cognitive behavior:** This is the foundation of *artificial life* which differs from artificial intelligence in that AI objects do not exhibit behaviors or have their own preferences. It can range from a simple cause-and-effect based system to more complex systems, known as *agents*, that have a psychological profile influenced by the surrounding environment.
- **Aggregate behavior:** Think of this as the overall outcome of a group of agents. This behavior is based on the individual rules of each agent and can influence the behavior of neighbors.

In this chapter, you'll keep your focus on aggregate behavior.

There's a strict correlation between the various types of aggregate behavior entities and their characteristics. In the following table, notice how the presence of a physics system or intelligence varies between entity types.



- **Particles** are the largest aggregate entities and are mostly governed by the laws of physics, but they lack intelligence.
- **Flocks** are an entity that's well-balanced between size, physics and intelligence.
- **Crowds** are smaller entities that are rarely driven by physics rules and are highly intelligent.

Working with crowd animation is both a challenging and rewarding experience. However, the purpose of this chapter is to describe and implement a flocking-like system, or to be more precise, a swarm of insects.

## Swarming Behavior

Swarms are gatherings of insects or other small-sized beings. The swarming behavior of insects can be modeled in a similar fashion as the flocking behavior of birds, the herding behavior of animals or the shoaling behavior of fish.

You know from the previous chapter that particle systems are fuzzy objects whose dynamics are mostly governed by the laws of physics. There are no interactions between particles, and usually, they are unaware of their neighboring particles. In contrast, swarming behavior uses the concept of neighboring quite heavily.

The swarming behavior follows a set of basic movement rules developed in 1986 by **Craig Reynolds** in an artificial flocking simulation program known as **Boids**. Since this chapter is heavily based on his work, the term **boid** will be used throughout the chapter instead of **particle**.

Initially, this basic set only included three rules: cohesion, separation and alignment. Later, more rules were added to extend the set to include a new type of agent; one that has *autonomous behavior* and is characterized by the fact that it has more intelligence than the rest of the swarm. This led to defining new models such as *follow-the-leader* and *predator-prey*.

Time to transform all of this knowledge into a swarm of quality code.

## The Starter Project

- In Xcode, open the starter project for this chapter. There are only a few files in this project.
  - In the **Flocking** group, **Emitter.swift** creates a buffer containing the particles. But each particle has only a position attribute.
  - Renderer calls a **FlockingPass** on every frame and supplies the view's current drawable texture as the GPU texture to update.
  - **FlockingPass** first clears the texture using the **clearScreen** compute shader from the previous project. It then dispatches threads for the number of particles. The dispatch code in **FlockingPass.draw(in:commandBuffer:)** contains an example of both macOS code and iOS code where non-uniform threads are not supported.
  - In the **Shaders** group, **Flocking.metal** has two kernel functions. One clears the drawable texture, and the other writes a pixel, representing a boid, to the given texture.

- Build and run the project, and you'll see this:



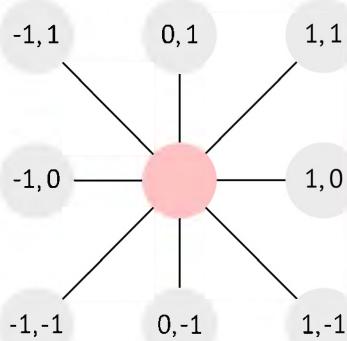
*The starter app*

There's a problem: a visibility issue. In its current state, the boids are barely distinguishable despite being white on a black background.

There's a neat trick you can apply in cases like this when you don't want to use a texture for boids (like you used in the previous chapter). In fact, scientific simulations and computational fluid dynamics projects very rarely use textures, if ever.

You can't use the `[ [point_size]]` attribute here because you're not rendering in the traditional sense. Instead, you're writing pixels in a kernel function directly to the drawable's texture.

The trick is to "paint" the surrounding neighbors of each boid, which makes the current boid seem larger than it really is.



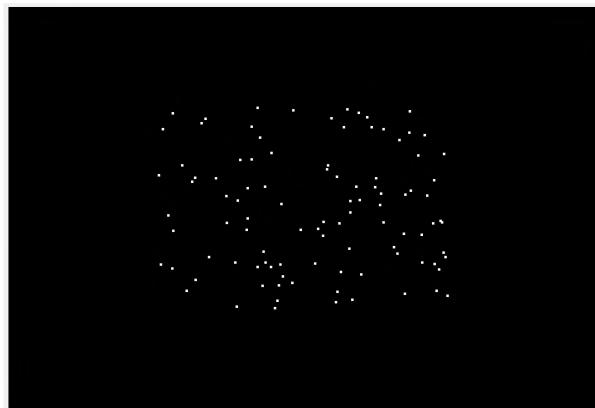
*Painting the pixels around the boid*

- In **Flocking.metal**, add this code at the end of the boids kernel function:

```
output.write(color, location + uint2(-1, 1));
output.write(color, location + uint2( 0, 1));
output.write(color, location + uint2( 1, 1));
output.write(color, location + uint2(-1, 0));
output.write(color, location + uint2( 1, 0));
output.write(color, location + uint2(-1, -1));
output.write(color, location + uint2( 0, -1));
output.write(color, location + uint2( 1, -1));
```

This code modifies the neighboring pixels around all sides of the boid which causes the boid to appear larger.

- Build and run the app, and you'll see that the boids are more distinguishable now.

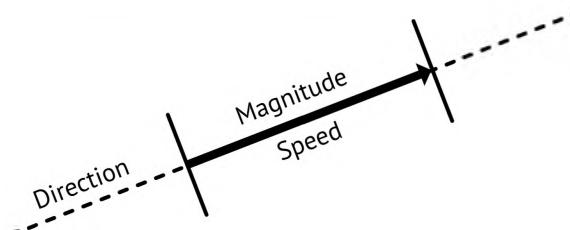


*Larger boids*

That's a good start, but how do you get them to move around? For that, you need to look into velocity.

# Velocity

Velocity is a vector made up of two other vectors: direction and speed. The speed is the magnitude or length of the vector, and the direction is given by the linear equation of the line on which the vector lies.



*Properties of a vector*

- In the **Flocking** group, open **Emitter.swift**, add a new member to the end of the **Particle** structure:

```
var velocity: float2
```

- In **init(particleCount:size:)**, inside the particle loop, add this before the last line where you advance the pointer:

```
let velocity: float2 = [
    Float.random(in: -5...5),
    Float.random(in: -5...5)
]
pointer.pointee.velocity = velocity
```

This gives the particle (boid) a random direction and speed that ranges between -5 and 5.

- In the **Shaders** group, open **Flocking.metal**, and add velocity as a new member of the **Boid** structure:

```
float2 velocity;
```

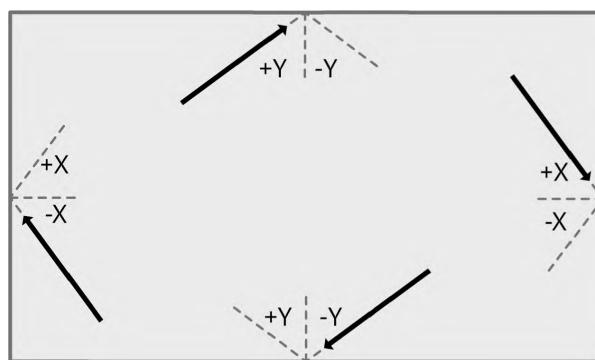
- In **boids**, add this code after the line where you define **position**:

```
float2 velocity = boid.velocity;
position += velocity;
boid.position = position;
boid.velocity = velocity;
boids[id] = boid;
```

This code gets the current velocity, updates the current position with the velocity, and then updates the boid data before storing the new values.

Build and run the app, and you'll see that the boids are now moving everywhere on the screen and... uh, wait! It looks like they're disappearing from the screen too. What happened?

Although you set the velocity to random values, you still need a way to force the boids to stay on the screen. Essentially, you need a way to make the boids bounce back when they hit any of the edges.



*Reflect and bounce at the edges*

For this function to work, you need to add checks for X and Y to make sure the boids stay in the rectangle defined by the origin and the size of the window, in other words, the width and height of your scene.

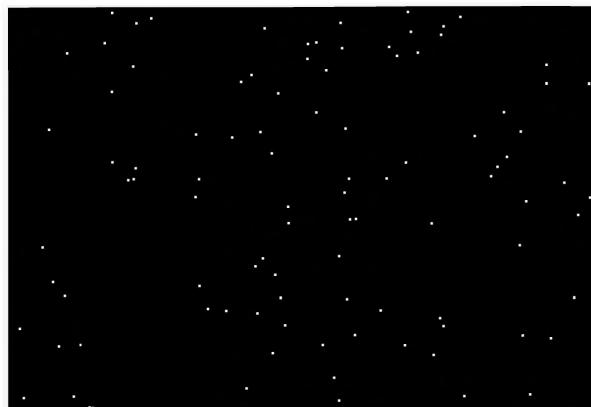
► In `boids`, add this code after `float2 velocity = boid.velocity;`:

```
if (position.x < 0 || position.x > output.get_width()) {
    velocity.x *= -1;
}

if (position.y < 0 || position.y > output.get_height()) {
    velocity.y *= -1;
}
```

Here, you check whether a boid coordinate gets outside the screen. If it does, you change the velocity sign, which changes the direction of the moving boid.

- Build and run the app, and you'll see that the boids are now bouncing back when hitting an edge.



*Bouncing boids*

Currently, the boids only obey the laws of physics. They'll travel to random locations with random velocities, and they'll stay on the window screen because of a few strict physical rules you're imposing on them.

The next stage is to make the boids behave as if they are able to think for themselves.

## Behavioral Rules

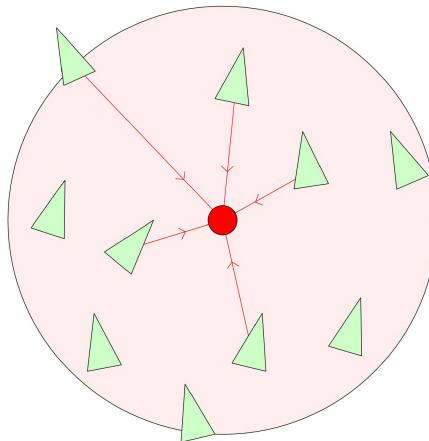
There's a basic set of steering rules that swarms and flocks can adhere to, and it includes:

- Cohesion
- Separation
- Alignment
- Escaping
- Dampening

You'll learn about each of these rules as you implement them in your project.

## Cohesion

**Cohesion** is a steering behavior that causes the boids to stay together as a group. To determine how cohesion works, you need to find the average position of the group, known as the *center of gravity*. Each neighboring boid will then apply a steering force in the direction of this center and converge near the center.



Cohesion

► In **Flocking.metal**, at the top of the file, add three global constants:

```
constant float average = 100;
constant float attenuation = 0.1;
constant float cohesionWeight = 2.0;
```

With these constants, you defined:

- **average**: A value that represents a smaller group of the swarm that stays cohesive.
- **attenuation**: A toning down factor that lets you relax the cohesion rule.
- **cohesionWeight**: The contribution made to the final cumulative behavior.

► Create a new function for cohesion before boids:

```
float2 cohesion(
    uint index,
    device Boid* boids,
    uint particleCount)
{
    // 1
    Boid thisBoid = boids[index];
    float2 position = float2(0);
```

```
// 2
for (uint i = 0; i < particleCount; i++) {
    Boid boid = boids[i];
    if (i != index) {
        position += boid.position;
    }
}
// 3
position /= (particleCount - 1);
position = (position - thisBoid.position) / average;
return position;
}
```

Going through the code:

1. Isolate the current boid at the given index from the rest of the group. Define and initialize `position`.
2. Loop through all of the boids in the swarm, and accumulate each boid's position to the `position` variable.
3. Get an average position value for the entire swarm, and calculate another averaged position based on the current boid position and the fixed value `average` that preserves average locality.

► In `boids`, add this code immediately before `position += velocity`:

```
float2 cohesionVector =
    cohesion(id, boids, particleCount) * attenuation;

// velocity accumulation
velocity += cohesionVector * cohesionWeight;
```

Here, you determine the cohesion vector for the current boid and then attenuate its force. You'll build upon the velocity accumulation line as you go ahead with new behavioral rules. For now, you give cohesion a weight of 2 and add it to the total velocity.



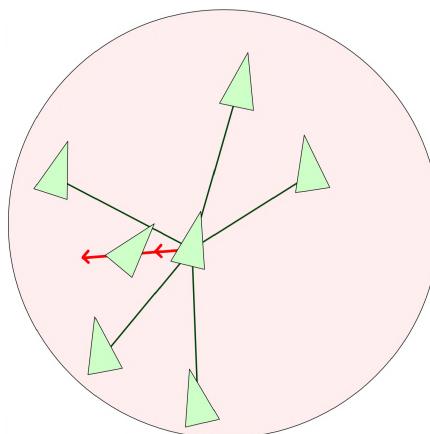
- Build and run the app. Notice how the boids are initially trying to get away — following their random directions. Moments later, they're pulled back toward the center of the flock.



*Converging boids*

## Separation

**Separation** is another steering behavior that allows a boid to stay a certain distance from nearby neighbors. This is accomplished by applying a repulsion force to the current boid when the set threshold for proximity is reached.



*Separation*

- Add two more global constants:

```
constant float limit = 20;  
constant float separationWeight = 1.0;
```

Here's what they're for:

- **limit**: A value that represents the proximity threshold that triggers the repulsion force.
  - **separationWeight**: The contribution made by the separation rule to the final cumulative behavior.
- Then, add the new separation function before **boids**:

```
float2 separation(
    uint index,
    device Boid* boids,
    uint particleCount)
{
    // 1
    Boid thisBoid = boids[index];
    float2 position = float2(0);
    // 2
    for (uint i = 0; i < particleCount; i++) {
        Boid boid = boids[i];
        if (i != index) {
            if (abs(distance(boid.position, thisBoid.position))
                < limit) {
                position =
                    position - (boid.position - thisBoid.position);
            }
        }
    }
    return position;
}
```

Going through the code:

1. Isolate the current boid at the given index from the rest of the group. Define and initialize **position**.
2. Loop through all of the boids in the swarm; if this is a boid other than the isolated one, check the distance between the current and isolated boids. If the distance is smaller than the proximity threshold, update the position to keep the isolated boid within a safe distance.

► In **boids**, before the **// velocity accumulation** comment, add this:

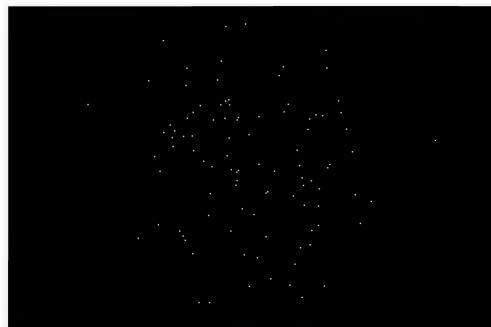
```
float2 separationVector = separation(id, boids, particleCount)
    * attenuation;
```



- Then, update the velocity accumulation to include the separation contribution by adding this code immediately after the last:

```
velocity += cohesionVector * cohesionWeight  
+ separationVector * separationWeight;
```

- Build and run the project. Notice that now there's a counter-effect of pushing back from cohesion as a result of the separation contribution.

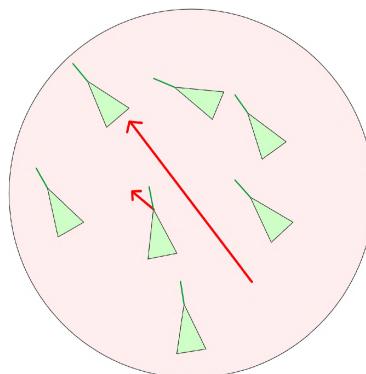


*Boid separation*

## Alignment

**Alignment** is the last of the three steering behaviors Reynolds used for his flocking simulation. The main idea is to calculate an average of the velocities for a limited number of neighbors. The resulting average is often referred to as the *desired velocity*.

With alignment, a steering force gets applied to the current boid's velocity to make it align with the group.



*Alignment*

- To get this working, add two global constants:

```
constant float neighbors = 8;
constant float alignmentWeight = 3.0;
```

With these constants, you define:

- **neighbors**: A value that represents the size of the local group that determines the “desired velocity”.
- **alignmentWeight**: The contribution made by the alignment rule to the final cumulative behavior.

- Then, add the new alignment function before `boids`:

```
float2 alignment(
    uint index,
    device Boid* boids,
    uint particleCount)
{
    // 1
    Boid thisBoid = boids[index];
    float2 velocity = float2(0);
    // 2
    for (uint i = 0; i < particleCount; i++) {
        Boid boid = boids[i];
        if (i != index) {
            velocity += boid.velocity;
        }
    }
    // 3
    velocity /= (particleCount - 1);
    velocity = (velocity - thisBoid.velocity) / neighbors;
    return velocity;
}
```

Going through the code:

1. Isolate the current boid at the given index from the rest of the group. Define and initialize `velocity`.
2. Loop through all of the boids in the swarm, and accumulate each boid’s velocity to the `velocity` variable.
3. Get an average velocity value for the entire swarm, and then calculate another averaged velocity based on the current boid velocity and the size of the local group, `neighbors`, which preserves locality.

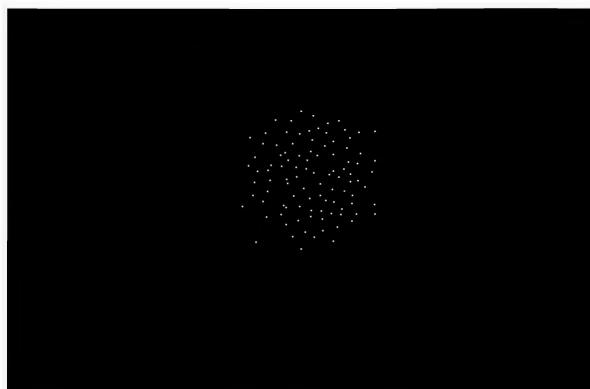
- In `boids`, before the `// velocity accumulation` comment, add this code:

```
float2 alignmentVector = alignment(id, boids, particleCount)
    * attenuation;
```

- Then, add this to update the velocity accumulation to include the alignment contribution:

```
velocity += cohesionVector * cohesionWeight
    + separationVector * separationWeight
    + alignmentVector * alignmentWeight;
```

- Build and run the app. The flock is homogeneous now because the alignment contribution brings balance to the previous two opposed contributions.

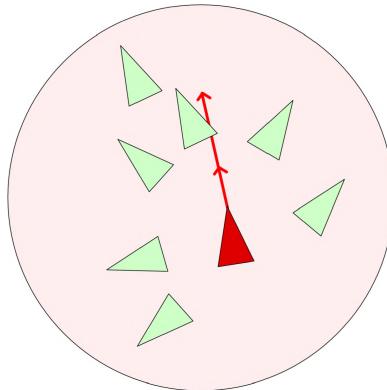


*Boids aligning*

## Escaping

**Escaping** is a new type of steering behavior that introduces an agent with autonomous behavior and slightly more intelligence — the *predator*.

In the predator-prey behavior, the predator tries to approach the closest prey on one side, while on the other side, the neighboring boids try to escape.



*Escaping*

- ▶ Like before, add new global constants to indicate the weight of the escaping force and the speed of reaction to the predator:

```
constant float escapingWeight = 0.01;
constant float predatorWeight = 10.0;
```

- ▶ Then, add the new escaping function before boids:

```
float2 escaping(Boid predator, Boid boid) {
    return -predatorWeight * (predator.position - boid.position)
        / average;
}
```

You return the averaged position of neighboring boids relative to the predator position. The final result is then adjusted and negated because the escaping direction is the opposite of where the predator is located.

- ▶ At the top of boids, replace:

```
Boid boid = boids[id];
```

- ▶ With the following code:

```
Boid predator = boids[0];
Boid boid;
if (id != 0) {
    boid = boids[id];
}
```

Here, you isolate the first boid in the buffer and label it as the predator. For the rest of the boids, you create a new boid object.

- Toward the end of `boids`, after defining `color`, add this:

```
if (id == 0) {
    color = half4(1.0, 0.0, 0.0, 1.0);
    location = uint2(boids[0].position);
}
```

The predator will stand out by coloring it red. You also save its current position.

- Before the line where you define `location`, add this:

```
// 1
if (predator.position.x < 0
    || predator.position.x > output.get_width()) {
    predator.velocity.x *= -1;
}
if (predator.position.y < 0
    || predator.position.y > output.get_height()) {
    predator.velocity.y *= -1;
}
// 2
predator.position += predator.velocity / 2.0;
boids[0] = predator;
```

With this code, you:

1. Check for collisions with the edges of the screen, and change the velocity when that happens.
2. Update the predator position with the current velocity, attenuated to half value to slow it down. Finally, save the predator position and velocity to preserve them for later use.

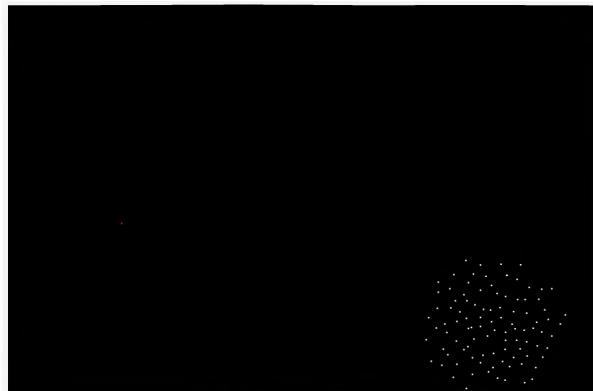
- Before the `// velocity accumulation` comment, add this:

```
float2 escapingVector = escaping(predator, boid) * attenuation;
```

- Then, add this to update the velocity accumulation to include the escaping contribution:

```
velocity += cohesionVector * cohesionWeight
+ separationVector * separationWeight
+ alignmentVector * alignmentWeight
+ escapingVector * escapingWeight;
```

- Build and run the app. Notice that some of the boids are steering away from the group and avoiding the predator.



*Escaping boids*

## Dampening

**Dampening** is the last steering behavior you'll looking at in this chapter. Its purpose is to dampen the effect of the escaping behavior, because at some point, the predator will stop its pursuit.

- Add one more global constant to represent the weight for the dampening:

```
constant float dampeningWeight = 1.0;
```

- Then, add the new dampening function before `boids`:

```
float2 dampening(Boid boid) {
    // 1
    float2 velocity = float2(0);
    // 2
    if (abs(boid.velocity.x) > limit) {
        velocity.x += boid.velocity.x / abs(boid.velocity.x)
                      * attenuation;
    }
    if (abs(boid.velocity.y) > limit) {
        velocity.y = boid.velocity.y / abs(boid.velocity.y)
                      * attenuation;
    }
    return velocity;
}
```

With this code, you:

1. Define and initialize the `velocity` variable.
2. Check if the velocity gets larger than the separation threshold. If it does, attenuate the velocity in the same direction.

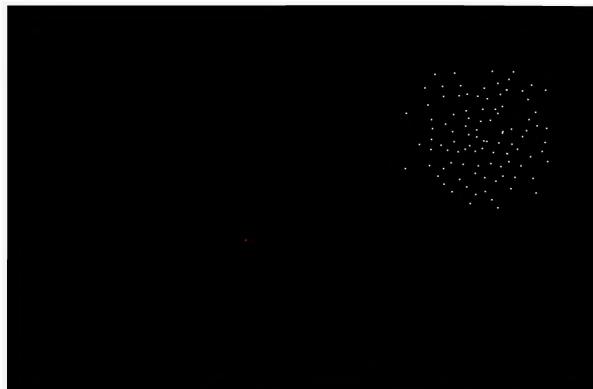
► In `boids`, before the `// velocity accumulation` comment, add this:

```
float2 dampeningVector = dampening(boid) * attenuation;
```

► Then, add this to update the velocity accumulation to include the dampening contribution:

```
velocity += cohesionVector * cohesionWeight  
+ separationVector * separationWeight  
+ alignmentVector * alignmentWeight  
+ escapingVector * escapingWeight  
+ dampeningVector * dampeningWeight;
```

► Build and run the app. Notice the boids are staying together with the group again after the predator breaks pursuit.



*Dampening keeps the boids together*

## Key Points

- You can give particles behavioral animation by causing them to react with other particles
- Swarming behavior has been widely researched. The Boids simulation describes basic movement rules.
- The behavioral rules for boids include cohesion, separation and alignment.
- Adding a predator to the particle mass requires an escaping algorithm.

## Where to Go From Here?

In this chapter, you learned how to construct basic behaviors and apply them to a small flock. Continue developing your project by adding a colorful background and textures for the boids. Or make it a 3D flocking app by adding projection to the scene. When you’re done, add the flock animation to your engine. Whatever you do, the sky is the limit.

This chapter barely scratched the surface of what is widely known as **behavioral animation**. Be sure to review the **references.markdown** file in the chapter directory for links to more resources about this wonderful topic.



# Section III: Advanced Metal

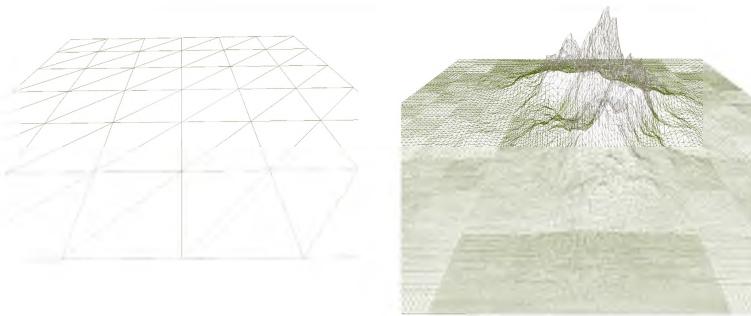
In this section, you'll learn many advanced features of Metal and explore realistic rendering techniques. You'll animate characters, and also manage rendering your scenes on the GPU.



# Chapter 19: Tessellation & Terrains

So far, you've used normal map trickery in the fragment function to show the fine details of your low poly models. To achieve a similar level of detail *without using normal maps* requires a change of model geometry by adding more vertices. The problem with adding more vertices is that when you send them to the GPU, it chokes up the pipeline. A hardware tessellator in the GPU can create vertices on the fly, adding a greater level of detail and thereby using fewer resources.

In this chapter, you'll create a detailed terrain using a small number of points. You'll send a flat ground plane with a grayscale texture describing the height, and the tessellator will create as many vertices as needed. The vertex function will then read the texture and **displace** (move) these new vertices vertically.



Control points

Tessellated vertices  
with displacement

## *Tessellation concept*

In this example, on the left side are the control points. On the right side, the tessellator creates extra vertices, with the number dependent on how close the control points are to the camera.



## Tessellation

For tessellation, instead of sending vertices to the GPU, you send patches. These patches are made up of control points — a minimum of three for a triangle patch, or four for a quad patch. The tessellator can convert each quad patch into a certain number of triangles: up to 4,096 triangles on a recent iMac and 256 triangles on an iPhone that's capable of tessellation.

**Note:** Tessellation is available on all Macs since 2012 and on iOS 10 GPU Family 3 and up. This includes the iPhone 6s and newer devices. However, Tessellation is not available on the iOS simulator.

With tessellation, you can:

- Send less data to the GPU. Because the GPU doesn't store tessellated vertices in graphics memory, it's more efficient on resources.
- Make low poly objects look less low poly by curving patches.
- Displace vertices for fine detail instead of using normal maps to fake it.
- Decide on the level of detail based on the distance from the camera. The closer an object is to the camera, the more vertices it contains.

## The Starter Project

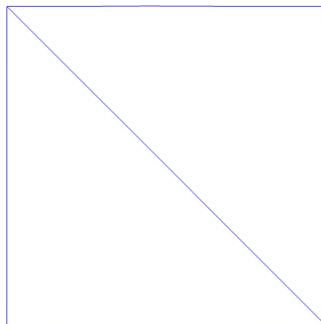
So that you can more easily understand the difference between rendering patches and rendering vertices, the starter project is a simplified renderer. All the rendering code is in **Renderer.swift**, with the pipeline state setup in **Pipelines.swift**.

**Quad.swift** contains the vertices and vertex buffer for the quad, and a method to generate control points.

- Open and run the starter project for this chapter.



The code in this project is the minimum needed for a simple render of six vertices to create a quad.

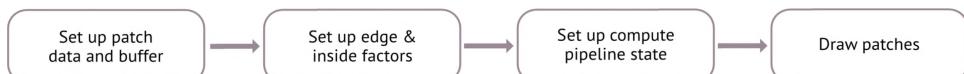


### *The starter app*

Your task in this chapter is to convert this quad to a terrain made up of patch quads with many vertices.

Before creating a tessellated terrain, you'll tessellate a single four-point patch. Instead of sending six vertices to the GPU, you'll send the positions of the four corners of the patch. You'll give the GPU **edge factors** and **inside factors** which tell the tessellator how many vertices to create. You'll render in wireframe line mode so you can see the vertices added by the tessellator, but you can change this with the **Wireframe** toggle in the app.

To convert the quad, you'll do the following on the CPU side:



*CPU pipeline*

On the GPU side, you'll set up a **tessellation kernel** that processes the edge and inside factors. You'll also set up a **post-tessellation vertex** shader that handles the vertices generated by the hardware tessellator.

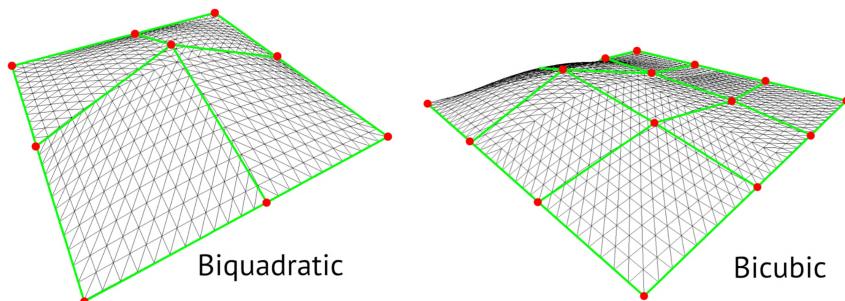


*GPU pipeline*

## Tessellation Patches

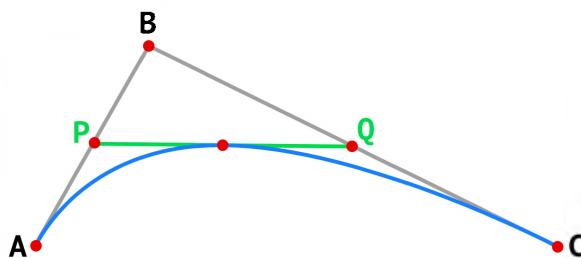
A patch consists of a certain number of control points, generally:

- **bilinear**: Four control points, one at each corner
- **biquadratic**: Nine control points
- **bicubic**: Sixteen control points



*Tessellated patches*

The control points make up a cage which is made up of spline curves. A spline is a parametric curve made up of control points. There are various algorithms to interpolate these control points, but here, A, B and C are the control points. As point P travels from A to B, point Q travels from B to C. The half way point between P and Q describes the blue curve.



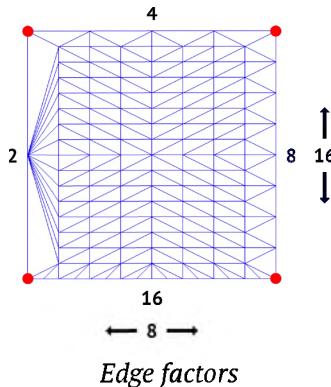
*A bezier curve*

To create the curved patch surface, the vertex function interpolates vertices to follow this parametric curve.

**Note:** Because the mathematics of curved surfaces is quite involved, you'll work with only four control points per patch in this chapter.

## Tessellation Factors

For each patch, you need to specify inside edge factors and outside edge factors. The four-point patch in the following image shows different edge factors for each edge — specified as [2, 4, 8, 16] — and two different inside factors — specified as [8, 16], for horizontal and vertical respectively.



The edge factors specify how many segments an edge will be split into. An edge factor of 2 has two segments along the edge. For the inside factors, look at the horizontal and vertical center lines. In this example, the horizontal center has eight segments, and the vertical center has sixteen.

Although only four control points (shown in red) went to the GPU, the hardware tessellator created a lot more vertices. However, creating more vertices on a flat plane doesn't make the render any more interesting. Later, you'll find out how to move these vertices around in the vertex function to make a bumpy terrain. But first, you'll discover how to tessellate a single patch.

► In `Renderer.swift`, in `Renderer`, add the following code:

```
let patches = (horizontal: 1, vertical: 1)
var patchCount: Int {
    patches.horizontal * patches.vertical
}
```

You create a constant for the number of patches you're going to create, in this case, one. `patchCount` is a convenience property that returns the total number of patches.

► Next, add this:

```
var edgeFactors: [Float] = [4]
var insideFactors: [Float] = [4]
```

Here, you set up the edge and inside factors as `Float` array properties. These variables indicate four segments along each edge, and four in the middle.

You can specify different factors for different edges by adding them to the array. For each patch, the GPU processes these edge factors and places the amount to tessellate each into a buffer.

- Create a property to provide a buffer of the correct length:

```
lazy var tessellationFactorsBuffer: MTLBuffer? = {  
    // 1  
    let count = patchCount * (4 + 2)  
    // 2  
    let size = count * MemoryLayout<Float>.size / 2  
    return Renderer.device.makeBuffer(  
        length: size,  
        options: .storageModePrivate)  
}()
```

1. `count` is the number of patches multiplied by the four edge factors and two inside factors.
2. Here you calculate the size of the buffer. In the tessellation kernel, you'll fill the buffer with a special type consisting of half-floats.

Now it's time to set up the patch data.

## Setting Up the Patch Data

Instead of an array of six vertices, you'll create a four-point patch with control points at the corners.

Currently, in `Quad.swift`, `Quad` holds a `vertexBuffer` property that contains the vertices. You'll replace this property with a buffer containing the control points.

- In `Renderer`, add the following property:

```
var controlPointsBuffer: MTLBuffer?
```

- At the end of `init(metalView:options:)`, fill the buffer with control points:

```
let controlPoints = Quad.createControlPoints(  
    patches: patches,  
    size: (2, 2))  
controlPointsBuffer =  
    Renderer.device.makeBuffer(  
        bytes: controlPoints,
```



```
length: MemoryLayout<float3>.stride * controlPoints.count)
```

**Quad.swift** contains a method, `createControlPoints(patches:size:)`. This method takes in the number of patches, and the unit size of the total number of patches. It then returns an array of xyz control points. Here, you create a patch with one corner at [-1, 0, 1], and the diagonal at [1, 0, -1]. This is a flat horizontal plane, but Renderer's `modelMatrix` rotates the patch by 90° so you can see the patch vertices.

## Set Up the Render Pipeline State

You can configure the tessellator by changing the pipeline state properties. Until now, you've processed only vertices with the vertex descriptor. However, you'll now modify the vertex descriptor so it processes patches instead.

- Open **Pipelines.swift**, and in `createRenderPSO(colorPixelFormat:)`, where you set up `vertexDescriptor`, add this:

```
vertexDescriptor.layouts[0].stepFunction = .perPatchControlPoint
```

With the old setup, you were using a default `stepFunction` of `.perVertex`. With that setup, the vertex function fetches new attribute data every time a new *vertex* is processed.

Now that you've moved on to processing patches, you need to fetch new attribute data for every *control point*.

## The Tessellation Kernel

To calculate the number of edge and inside factors, you'll set up a compute pipeline state object that points to the tessellation kernel shader function.

- Open **Renderer.swift**, and add a new property to `Renderer`:

```
var tessellationPipelineState: MTLComputePipelineState
```

- In `init(metalView:options:)`, before `super.init()`, add this:

```
tessellationPipelineState =
    PipelineStates.createComputePSO(function: "tessellation_main")
```

Here, you instantiate the pipeline state for the compute pipeline.

## Compute Pass

You now have a compute pipeline state and an MTLBuffer containing the patch data. You also created an empty buffer which the tessellation kernel will fill with the edge and inside factors. Next, you need to create the compute command encoder to dispatch the tessellation kernel.

- In `tessellation(commandBuffer:)`, add the following:

```
guard let computeEncoder =  
    commandBuffer.makeComputeCommandEncoder() else { return }  
computeEncoder.setComputePipelineState(  
    tessellationPipelineState)  
computeEncoder.setBytes(  
    &edgeFactors,  
    length: MemoryLayout<Float>.size * edgeFactors.count,  
    index: 0)  
computeEncoder.setBytes(  
    &insideFactors,  
    length: MemoryLayout<Float>.size * insideFactors.count,  
    index: 1)  
computeEncoder.setBuffer(  
    tessellationFactorsBuffer,  
    offset: 0,  
    index: 2)
```

`draw(in:)` calls `tessellation(commandBuffer:)` before any rendering. You create a compute command encoder and bind the edge and inside factors to the compute function (the tessellation kernel). If you have multiple patches, the compute function will operate in parallel on each patch, on different threads.

- To tell the GPU how many threads you need, continue by adding this after the previous code:

```
let width = min(  
    patchCount,  
    tessellationPipelineState.threadExecutionWidth)  
let gridSize =  
    MTLSize(width: patchCount, height: 1, depth: 1)  
let threadsPerThreadgroup =  
    MTLSize(width: width, height: 1, depth: 1)  
computeEncoder.dispatchThreadgroups(  
    gridSize,  
    threadsPerThreadgroup: threadsPerThreadgroup)  
computeEncoder.endEncoding()
```

The compute grid is one dimensional, with a thread for each patch.



Before changing the render encoder so it'll draw patches instead of vertices, you'll need to create the tessellation kernel.

## The Tessellation Kernel Function

- Create a new Metal file named **Tessellation.metal**, and add this:

```
#import "Common.h"

kernel void
tessellation_main(
    constant float *edge_factors [[buffer(0)]],
    constant float *inside_factors [[buffer(1)]],
    device MTLQuadTessellationFactorsHalf
        *factors [[buffer(2)]],
    uint pid [[thread_position_in_grid]])
```

`kernel` specifies the type of shader. The function operates on all threads (i.e., all patches) and receives the three things you sent over: the edge factors, inside factors and the empty tessellation factors buffer that you're going to fill in this function.

The fourth parameter is the patch ID with its thread position in the grid. The tessellation factors buffer consists of an array of edge and inside factors for each patch, and `pid` gives you the patch index into this array.

- Inside the kernel function, add the following:

```
factors[pid].edgeTessellationFactor[0] = edge_factors[0];
factors[pid].edgeTessellationFactor[1] = edge_factors[0];
factors[pid].edgeTessellationFactor[2] = edge_factors[0];
factors[pid].edgeTessellationFactor[3] = edge_factors[0];

factors[pid].insideTessellationFactor[0] = inside_factors[0];
factors[pid].insideTessellationFactor[1] = inside_factors[0];
```

This code fills in the tessellation factors buffer with the edge factors that you sent over. The edge and inside factors array you sent over only had one value each, so you put this value into all factors.

Filling out a buffer with values is a trivial thing for a kernel to do, and you could do this on the CPU. However, as you get more patches and more complexity on how to tessellate these patches, you'll understand why sending the data to the GPU for parallel processing is a useful step.

After the compute pass is done, the render pass takes over.



## The Render Pass

Before doing the render, you need to tell the render encoder about the tessellation factors buffer that you updated during the compute pass.

- Open **Renderer.swift**, and in `render(commandBuffer:view:)`, locate the `// draw` comment. Just after that comment, add this:

```
renderEncoder.setTessellationFactorBuffer(  
    tessellationFactorsBuffer,  
    offset: 0,  
    instanceStride: 0)
```

The post-tessellation vertex function reads from this buffer that you set up during the kernel function.

Instead of drawing triangles from the vertex buffer, you'll draw the patch using patch control points from the control points buffer.

- Replace:

```
renderEncoder.setVertexBuffer(  
    quad.vertexBuffer,  
    offset: 0,  
    index: 0)
```

With:

```
renderEncoder.setVertexBuffer(  
    controlPointsBuffer,  
    offset: 0,  
    index: 0)
```

- Replace the `drawPrimitives` command with this:

```
renderEncoder.drawPatches(  
    numberofPatchControlPoints: 4,  
    patchStart: 0,  
    patchCount: patchCount,  
    patchIndexBuffer: nil,  
    patchIndexBufferOffset: 0,  
    instanceCount: 1,  
    baseInstance: 0)
```

The render command encoder tells the GPU that it's going to draw one patch with four control points.

## The Post-Tessellation Vertex Function

- Open `Shaders.metal`.

The GPU calls the vertex function after the tessellator has done its job of creating the vertices. The function will operate on each one of these new vertices. In the vertex function, you'll tell each vertex what its position in the rendered quad should be.

- Rename `VertexIn` to `ControlPoint`.

The definition of `position` remains the same. Because you used a vertex descriptor to describe the incoming control point data, you can use the `[[stage_in]]` attribute.

The vertex function will check the vertex descriptor from the current pipeline state, find that the data is in buffer 0 and use the vertex descriptor layout to read in the data.

- Replace `vertex_main` with:

```
// 1
[[patch(quad, 4)]]
// 2
vertex VertexOut
    vertex_main(
// 3
        patch_control_point<ControlPoint> control_points
    [[stage_in]],
// 4
        constant Uniforms &uniforms [[buffer(BufferIndexUniforms)]],
// 5
        float2 patch_coord [[position_in_patch]])
{}
```

This is the post-tessellation vertex function where you return the correct position of the vertex for the rasterizer. Going through the code:

1. The function qualifier tells the vertex function that the vertices are coming from tessellated patches. It describes the type of patch, triangle or quad, and the number of control points, in this case, four.
2. The function is still a vertex shader function as before.
3. `patch_control_point` is part of the Metal Standard Library and provides the per-patch control point data.

4. Uniforms contains the model-view-projection matrix you passed in.
5. The tessellator provides a uv coordinate between 0 and 1 for the tessellated patch so that the vertex function can calculate its correct rendered position.

To visualize how this works, you can temporarily return the UV coordinates as the position and color.

► Add the following code to `vertex_main`:

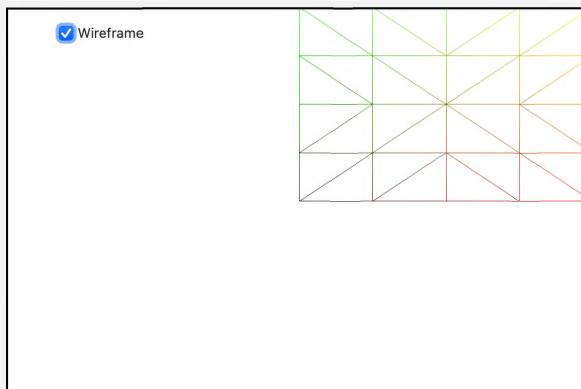
```
float u = patch_coord.x;
float v = patch_coord.y;

VertexOut out;
out.position = float4(u, v, 0, 1);
out.color = float4(u, v, 0, 1);
return out;
```

Here, you give the vertex a position as interpolated by the tessellator from the four patch positions, and a color of the same value for visualization.

► Build and run the app.

See how the patch is tessellated with vertices between 0 and 1? (Normalized Device Coordinates (NDC) are between -1 and 1 which is why all the coordinates are at the top right.)



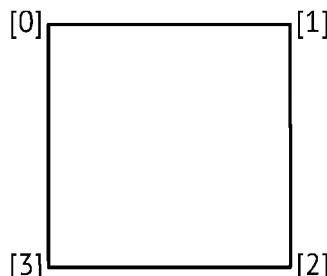
*Basic tessellation*

To have your vertex positions depend on the patch's actual position rather than between 0 and 1, you need to interpolate the patch's control points depending on the UV values.

- In `vertex_main`, after assigning the `u` and `v` values, add this:

```
float2 top = mix(  
    control_points[0].position.xz,  
    control_points[1].position.xz,  
    u);  
float2 bottom = mix(  
    control_points[3].position.xz,  
    control_points[2].position.xz,  
    u);
```

You interpolate values horizontally along the top of the patch and the bottom of the patch. Notice the index ordering: the patch indices are 0 to 3 clockwise.



*Control point winding order*

You can change this by setting the pipeline descriptor property `tessellationOutputWindingOrder` to `.counterClockwise`.

- Change the following code:

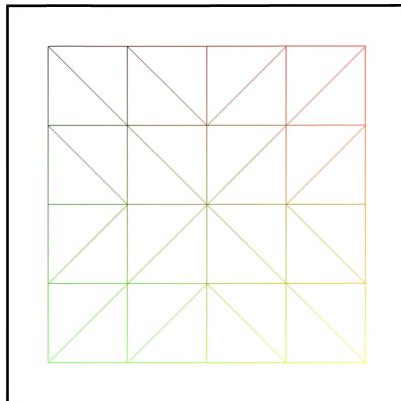
```
out.position = float4(u, v, 0, 1);
```

- To:

```
float2 interpolated = mix(top, bottom, v);  
float4 position = float4(  
    interpolated.x, 0.0,  
    interpolated.y, 1.0);  
out.position = uniforms.mvp * position;
```

You interpolate the vertical value between the top and bottom values and multiply it by the model-view-projection matrix to position the vertex in the scene. Currently, you're leaving  $y$  at  $0.0$  to keep the patch two-dimensional.

- Build and run the app to see your tessellated patch:



*A tessellated patch*

**Note:** Experiment with changing the edge and inside factors until you're comfortable with how the tessellator subdivides. For example, change the edge factors array to [2, 4, 8, 16], and change the kernel function so that the appropriate array value goes into each edge.

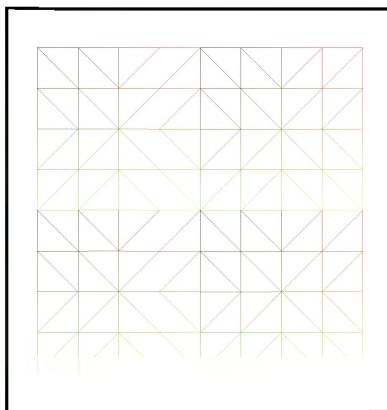
## Multiple Patches

Now that you know how to tessellate one patch, you can tile the patches and choose edge factors that depend on dynamic factors, such as distance.

- Open **Renderer.swift**, and change the patches initialization to:

```
let patches = (horizontal: 2, vertical: 2)
```

- Build and run the app to see the four patches joined together:



*Four tessellated patches*

In the vertex function, you can identify which patch you're currently processing using the `[[patch_id]]` attribute.

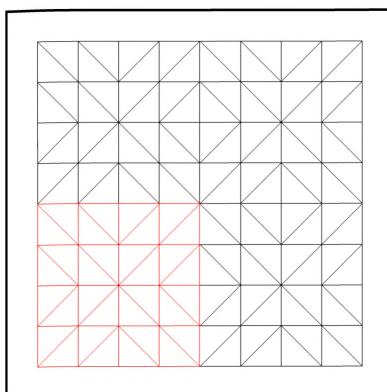
- Open **Shaders.metal**, and add this parameter to `vertex_main`:

```
uint patchID [[patch_id]]
```

- Change the assignment of `out.color` to:

```
out.color = float4(0);
if (patchID == 0) {
    out.color = float4(1, 0, 0, 1);
}
```

- Build and run the app.



*Colored by patch id*

Notice how the GPU colors the bottom left patch red. This is the first patch in the control points array.

## Tessellation By Distance

In this section, you're going to create a terrain with patches that are tessellated according to the distance from the camera. When you're close to a mountain, you need to see more detail; when you're farther away, less. Having the ability to dial in the level of detail is where tessellation comes into its own. By setting the level of detail, you save on how many vertices the GPU has to process in any given situation.

- Open **Common.h**, and add the following code:

```
typedef struct {
    vector_float2 size;
    float height;
    uint maxTessellation;
} Terrain;
```

You set up a new structure to describe the size and maximum tessellation of the terrain. You'll use `height` for scaling vertices on the y-axis later.

- Open **Renderer.swift**, and add a new constant to `Renderer`:

```
static let maxTessellation = 16
```

This value is the maximum amount you can tessellate per patch. On iOS devices, currently the maximum amount is 16, but on new Macs, the maximum is 64.

- Add a new property:

```
var terrain = Terrain(
    size: [2, 2],
    height: 1,
    maxTessellation: UInt32(Renderer.maxTessellation))
```

You describe the terrain with four patches and a maximum height of one unit.

- Locate where you set up `controlPoints` in `init(metalView:)`, and change it to:

```
let controlPoints = Quad.createControlPoints(  
    patches: patches,  
    size: (width: terrain.size.x, height: terrain.size.y))
```

Because your terrains are going to be much larger, you'll use the `terrain` constant to create the control points.

To calculate edge factors that are dependent on the distance from the camera, you will send the camera position, model matrix and control points to the kernel.

- In `tessellation(commandBuffer:)`, before you set the width of the compute threads with `let width = min(patchCount...,` add this:

```
var cameraPosition = float4(camera.position, 0)  
computeEncoder.setBytes(  
    &cameraPosition,  
    length: MemoryLayout<float4>.stride,  
    index: 3)  
var matrix = modelMatrix  
computeEncoder.setBytes(  
    &matrix,  
    length: MemoryLayout<float4x4>.stride,  
    index: 4)  
computeEncoder.setBuffer(  
    controlPointsBuffer,  
    offset: 0,  
    index: 5)  
computeEncoder.setBytes(  
    &terrain,  
    length: MemoryLayout<Terrain>.stride,  
    index: 6)
```

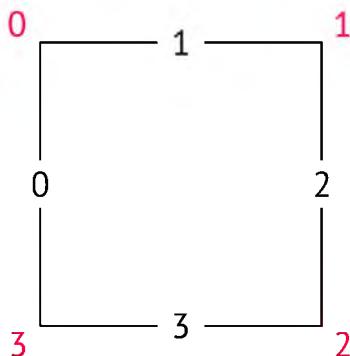
You send the camera position, along with the model matrix, the control points buffer and the terrain information to the tessellation kernel.

- Open `Tessellation.metal`, and add these parameters to `tessellation_main`:

```
constant float4 &camera_position [[buffer(3)]],  
constant float4x4 &modelMatrix [[buffer(4)]],  
constant float3* control_points [[buffer(5)]],  
constant Terrain &terrain [[buffer(6)]],
```

With these constants, you can compute the distance of the edges from the camera.

You'll set the edge and inside tessellation factors differently for each patch edge instead of sending a constant 4 for all of the edges. The further the patch edge is from the camera, the lower the tessellation on that edge. These are the edge and control point orders for each patch:



#### Edges and control points

To calculate the tessellation of an edge, you need to know the transformed mid-point of two control points. To calculate edge 2, for example, you get the midpoint of points 1 and 2 and find out the distance of that point from the camera. Where two patches join, it's imperative to keep the tessellation level for the joined edges the same, otherwise you get cracks. By calculating the distance of the mid-point, you end up with the same result for the overlapping edges.

► In `Tessellation.metal`, create a new function *before* `tessellation_main`:

```
float calc_distance(
    float3 pointA, float3 pointB,
    float3 camera_position,
    float4x4 modelMatrix)
{
    float3 positionA = (modelMatrix * float4(pointA, 1)).xyz;
    float3 positionB = (modelMatrix * float4(pointB, 1)).xyz;
    float3 midpoint = (positionA + positionB) * 0.5;

    float camera_distance = distance(camera_position, midpoint);
    return camera_distance;
}
```

This function takes in two points: The camera position and the model matrix. The function then finds the mid-point between the two points and calculates the distance from the camera.

- Remove all of the code from `tessellation_main`.
- Add the following line of code to `tessellation_main` to calculate the correct index into the tessellation factors array:

```
uint index = pid * 4;
```

`4` is the number of control points per patch, and `pid` is the patch ID. To index into the control points array for each patch, you skip over four control points at a time.

- Add this line to keep a running total of tessellation factors:

```
float totalTessellation = 0;
```

- Add a `for` loop for each of the edges:

```
for (int i = 0; i < 4; i++) {  
    int pointAIndex = i;  
    int pointBIndex = i + 1;  
    if (pointAIndex == 3) {  
        pointBIndex = 0;  
    }  
    int edgeIndex = pointBIndex;  
}
```

You cycle around four corners: 0, 1, 2, 3. On the first iteration, you calculate edge 1 from the mid-point of points 0 and 1. On the fourth iteration, you use points 3 and 0 to calculate edge 0.

- At the end of the `for` loop, call the distance calculation function:

```
float cameraDistance =  
    calc_distance(  
        control_points[pointAIndex + index],  
        control_points[pointBIndex + index],  
        camera_position.xyz,  
        modelMatrix);
```

- Then, still inside the `for` loop, set the tessellation factor for the current edge:

```
float tessellation =  
    max(4.0, terrain.maxTessellation / cameraDistance);  
factors[pid].edgeTessellationFactor[edgeIndex] = tessellation;  
totalTessellation += tessellation;
```

You set a minimum edge factor of 4. The maximum depends upon the camera distance and the maximum tessellation amount you specified for the terrain.

- After the `for` loop, add this:

```
factors[pid].insideTessellationFactor[0] =  
    totalTessellation * 0.25;  
factors[pid].insideTessellationFactor[1] =  
    totalTessellation * 0.25;
```

You set the two inside tessellation factors to be an average of the total tessellation for the patch. You've now finished creating the compute kernel which calculates the edge factors based on distance from the camera.

Lastly, you'll revise some of the default render pipeline state tessellation parameters.

- Open `Pipelines.swift`, and in `createRenderPSO(colorPixelFormat:)`, add this before the `return`:

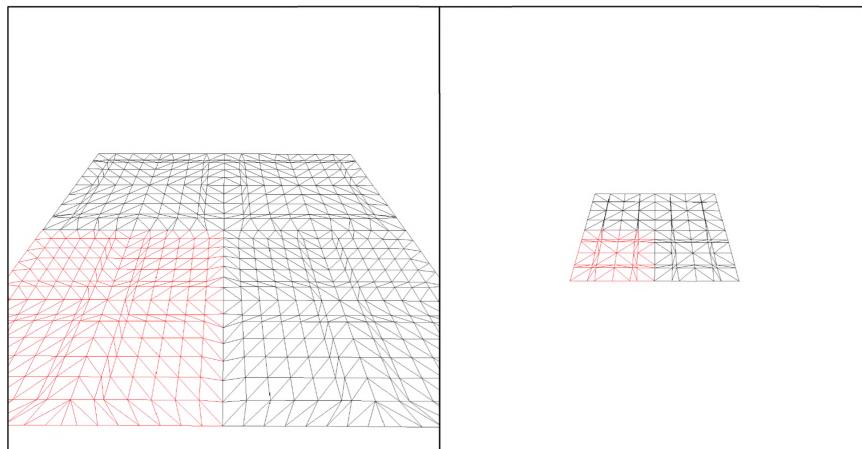
```
// 1  
pipelineDescriptor.tessellationFactorStepFunction = .perPatch  
// 2  
pipelineDescriptor.maxTessellationFactor =  
    Renderer.maxTessellation  
// 3  
pipelineDescriptor.tessellationPartitionMode = .fractionalEven
```

1. The step function was previously set to a default `.constant`, which sets the same edge factors on all patches. By setting this to `.perPatch`, the vertex function uses each patch's edge and inside factors information in the tessellation factors array.
2. You set the maximum number of segments per patch for the tessellator.
3. The partition mode describes how these segments are split up. The default is `.pow2`, which rounds up to the nearest power of two. Using `.fractionalEven`, the tessellator rounds up to the nearest even integer, so it allows for much more variation of tessellation.

- Build and run the app, and rotate and zoom your patches.



As you reposition the patches, the tessellator recalculates their distance from the camera and tessellates accordingly. Tessellating is a neat superpower!



*Tessellation by distance*

Check where the patches join. The triangles of each side of the patch should connect.

Now that you've mastered tessellation, you'll be able to add detail to your terrain.

## Displacement

You've used textures for various purposes in earlier chapters. Now you'll use a height map to change the height of each vertex. Height maps are grayscale images where you can use the texel value for the Y vertex position, with white being high and black being low. There are several height maps in **Textures.xcassets** you can experiment with.

► Open **Renderer.swift**, and create a property to hold the height map:

```
let heightMap: MTLTexture!
```

► In **init(metalView:)**, before calling **super.init()**, initialize **heightMap**:

```
do {
    heightMap = try TextureController.loadTexture(filename:
    "mountain")
} catch {
    fatalError(error.localizedDescription)
}
```

Here, you load the height map texture from the asset catalog.

- In `render(commandBuffer:view:)`, add the following code before the draw call `renderEncoder.drawPatches(...)`:

```
renderEncoder.setVertexTexture(heightMap, index: 0)
renderEncoder.setVertexBytes(
    &terrain,
    length: MemoryLayout<Terrain>.stride,
    index: 6)
```

You're already familiar with sending textures to the fragment shader, which makes the texture available to the vertex shader in the same way. You also send the terrain setup details.

- Open `Shaders.metal`, and add the following to `vertex_main`'s parameters:

```
texture2d<float> heightMap [[texture(0)]],
constant Terrain &terrain [[buffer(6)]],
```

You read in the texture and terrain information.

You're currently only using the `x` and `z` position coordinates for the patch and leaving the `y` coordinate as zero. You'll now map the `y` coordinate to the height indicated in the texture.

Just as you used `u` and `v` fragment values to read the appropriate texel in the fragment function, you use the `x` and `z` position coordinates to read the texel from the height map in the vertex function.

- After setting `position`, but *before* multiplying by the model-view-projection matrix, add this:

```
// 1
float2 xy = (position.xz + terrain.size / 2.0) / terrain.size;
// 2
constexpr sampler sample;
float4 color = heightMap.sample(sample, xy);
out.color = float4(color.r);
// 3
float height = (color.r * 2 - 1) * terrain.height;
position.y = height;
```

Going through the code:

1. You convert the patch control point values to be between 0 and 1 to be able to sample the height map. You include the terrain size because, although your patch control points are currently between -1 and 1, soon you'll be making a larger terrain.
2. Create a default sampler and read the texture as you have done previously in the fragment function. The texture is a grayscale texture, so you only use the .r value.
3. color is between 0 and 1, so for the height, shift the value to be between -1 and 1, and multiply it by your terrain height scale setting. This is currently set to 1.

► Next, remove the following code from the end of the vertex function, because you're now using the color of the height map.

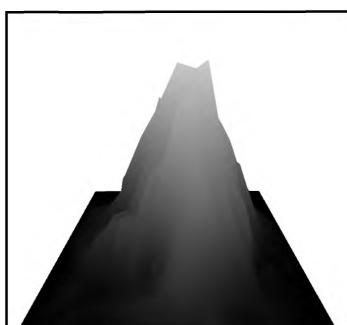
```
out.color = float4(0);
if (patchID == 0) {
    out.color = float4(1, 0, 0, 1);
}
```

► Open `Renderer.swift`, and change the rotation property initialization in `modelMatrix` to:

```
let rotation = float3(Float(-20).degreesToRadians, 0, 0)
```

You'll now look at your tessellated plane from the side, rather than from the top.

► Build and run the app to see the height map displacing the vertices. Notice how the white vertices are high and the black ones are low:



*Height map displacement*

This render doesn't yet have much detail, but that's about to change.

► In **Renderer.swift**, change the `maxTessellation` constant to:

```
static let maxTessellation: Int = {  
    #if os(macOS)  
    return 64  
    #else  
    return 16  
    #endif  
}()
```

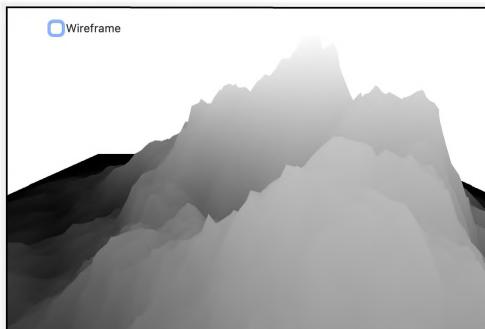
These are the maximum values for each OS. Because the maximum tessellation on iOS is so low, you may want to increase the number of patches rendered on iOS.

► Change patches and terrain to:

```
let patches = (horizontal: 6, vertical: 6)  
var terrain = Terrain(  
    size: [8, 8],  
    height: 1,  
    maxTessellation: UInt32(Renderer.maxTessellation))
```

This time, you're creating thirty-six patches over sixteen units.

► Build and run the app to see your patch height-mapped into a magnificent mountain. Don't forget to click off the wireframe option to see your mountain render in its full glory.



*A tessellated mountain*

Now it's time to render your mountain with different colors and textures depending on height.

## Shading By Height

In the previous section, you sampled the height map in the vertex function, and the colors are interpolated when sent to the fragment function. For maximum color detail, you need to sample from textures per fragment, not per vertex.

For that to work, you'll set up three textures: snow, cliff and grass. You'll send these textures to the fragment function and test the height there.

- Open **Renderer.swift**, and add three new texture properties to **Renderer**:

```
let cliffTexture: MTLTexture?  
let snowTexture: MTLTexture?  
let grassTexture: MTLTexture?
```

- In **init(metalView:)**, in the do closure where you create the height map, add this:

```
cliffTexture =  
    try TextureController.loadTexture(filename: "cliff-color")  
snowTexture =  
    try TextureController.loadTexture(filename: "snow-color")  
grassTexture =  
    try TextureController.loadTexture(filename: "grass-color")
```

These textures are in the asset catalog.

- To send the textures to the fragment function, in **render(commandBuffer:view:)**, add the following code before the **renderEncoder** draw call:

```
renderEncoder.setFragmentTexture(cliffTexture, index: 1)  
renderEncoder.setFragmentTexture(snowTexture, index: 2)  
renderEncoder.setFragmentTexture(grassTexture, index: 3)
```

- Open **Shaders.metal**, and add two new properties to **VertexOut**:

```
float height;  
float2 uv;
```

- At the end of **vertex\_main**, before the **return**, set the value of these two properties:

```
out.uv = xy;  
out.height = height;
```



You send the height value from the vertex function to the fragment function so that you can assign fragments the correct texture for that height.

- Add the textures to the `fragment_main` parameters:

```
texture2d<float> cliffTexture [[texture(1)]],  
texture2d<float> snowTexture [[texture(2)]],  
texture2d<float> grassTexture [[texture(3)]]
```

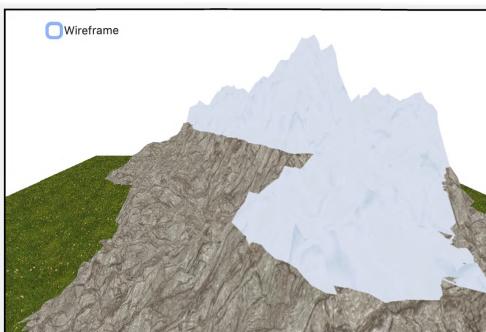
- Replace the contents of `fragment_main` with:

```
constexpr sampler sample(filter::linear, address::repeat);  
float tiling = 16.0;  
float4 color;  
if (in.height < -0.5) {  
    color = grassTexture.sample(sample, in.uv * tiling);  
} else if (in.height < 0.3) {  
    color = cliffTexture.sample(sample, in.uv * tiling);  
} else {  
    color = snowTexture.sample(sample, in.uv * tiling);  
}  
return color;
```

You create a tileable texture sampler and read in the appropriate texture for the height. Height is between  $-1$  and  $1$ , as set in `vertex_main`. You then tile the texture by  $16$  — an arbitrary value based on what looks best here.

- Build and run the app. Click the wireframe toggle to see your textured mountain.

You have grass at low altitudes and snowy peaks at high altitudes.



*A textured mountain*

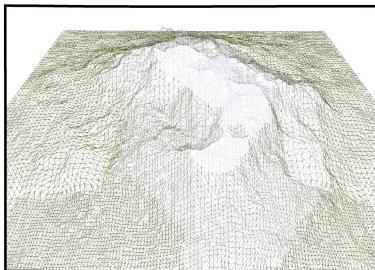
As you zoom and rotate, notice how the mountain seems to ripple. This is the tessellation level of detail being over-sensitive. One way of dialing this down is to change the render pass's tessellation partition mode.

- Open **Pipelines.swift**, and in `createRenderPSO(colorPixelFormat:)`, change the `pipelineDescriptor.tessellationPartitionMode` assignment to:

```
pipelineDescriptor.tessellationPartitionMode = .pow2
```

- Build and run the app.

As the tessellator rounds up the edge factors to a power of two, there's a larger difference in tessellation between the patches now, but the change in tessellation won't occur so frequently, and the ripple disappears.



*Rounding edge factors to a power of two*

## Shading By Slope

The snow line in your previous render is unrealistic. By checking the slope of the mountain, you can show the snow texture in flatter areas, and show the cliff texture where the slope is steep.

An easy way to calculate slope is to run a **Sobel filter** on the height map. A Sobel filter is an algorithm that looks at the gradients between neighboring pixels in an image. It's useful for edge detection in computer vision and image processing, but in this case, you can use the gradient to determine the slope between neighboring pixels.

## Metal Performance Shaders

The Metal Performance Shaders framework contains many useful, highly optimized shaders for image processing, matrix multiplication, machine learning and raytracing. You'll read more about them in Chapter 30, "Metal Performance Shaders."

The shader you'll use here is `MPSImageSobel`, which takes a source image texture and outputs the filtered image into a new grayscale texture. The whiter the pixel, the steeper the slope.

**Note:** In the challenge for this chapter, you'll use the Sobel-filtered image and apply the three textures to your mountain depending on slope.

- Open `Renderer.swift`, and import the Metal Performance Shaders framework:

```
import MetalPerformanceShaders
```

- Create a new method in `Renderer` to process the height map:

```
static func heightToSlope(source: MTLTexture) -> MTLTexture {  
}
```

Next, you'll send the height map to this method and return a new texture. To create the new texture, you first need to create a texture descriptor where you can assign the size, pixel format and tell the GPU how you will use the texture.

- Add this to `heightToSlope(source:)`:

```
let descriptor =  
    MTLTextureDescriptor.texture2DDescriptor(  
        pixelFormat: sourcePixelFormat,  
        width: source.width,  
        height: source.height,  
        mipmapped: false)  
descriptor.usage = [.shaderWrite, .shaderRead]
```

You create a descriptor for textures that you want to both read and write. You'll write to the texture in the MPS shader and read it in the fragment shader.

- Continue adding to the method:

```
guard let destination =  
    Renderer.device.makeTexture(descriptor: descriptor),  
    let commandBuffer = Renderer.commandQueue.makeCommandBuffer()  
else {  
    fatalError("Error creating Sobel texture")  
}
```

This creates the texture and the command buffer for the MPS shader.

- Now, add this:

```
let shader = MPSImageSobel(device: Renderer.device)  
shader.encode(  
    commandBuffer: commandBuffer,  
    sourceTexture: source,
```



```
destinationTexture: destination)
commandBuffer.commit()
return destination
```

You run the MPS shader and return the texture. That's all there is to running a Metal Performance Shader on a texture.

**Note:** The height maps in the asset catalog have a pixel format of **8 Bit Normalized - R**, or **R8Unorm**. Using the default pixel format of **RGBA8Unorm** with **MPSImageSobel** crashes. In any case, for grayscale texture maps that only use one channel, using **R8Unorm** as a pixel format is more efficient.

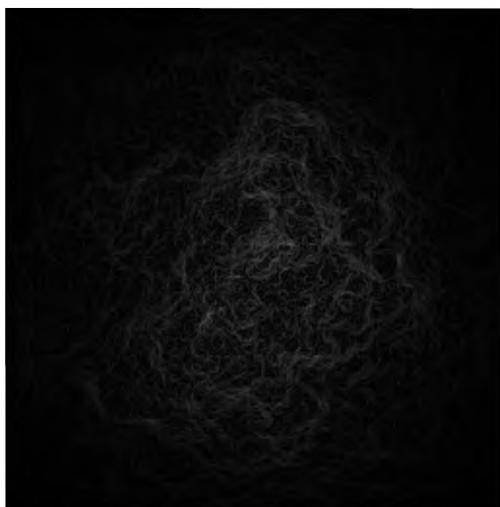
- To hold the terrain slope in a texture, add a new property to `Renderer`:

```
let terrainSlope: MTLTexture
```

- In `init(metalView:)`, before calling `super.init()`, initialize the texture:

```
terrainSlope = Renderer.heightToSlope(source: heightMap)
```

The texture when created will look like this:



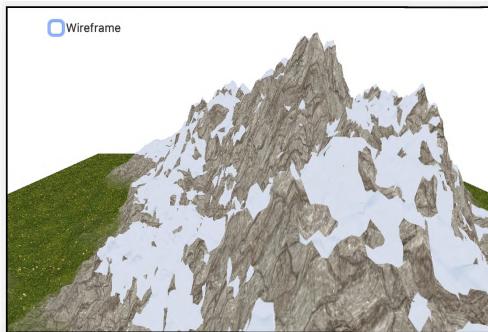
*The Sobel filter*

In the challenge, once you send this texture to the vertex shader, you'll be able to see it using the Capture GPU Frame icon. The white parts are the steep slopes.

## Challenge

Your challenge for this chapter is to use the slope texture from the Sobel filter to place snow on the mountain on the parts that aren't steep. Because you don't need pixel perfect accuracy, you can read the slope image in the vertex function and send that value to the fragment function. This is more efficient as there will be fewer texture reads in the vertex function than in the fragment function.

If everything goes well, you'll render an image like this:



*Shading by slope*

Notice how the grass blends into the mountain. This is done using the `mix()` function.

Currently, you have three zones — the heights where you render the three different textures. The challenge project has four zones:

- **grass:**  $< -0.6$  in height
- **grass blended with mountain:**  $-0.6$  to  $-0.4$
- **mountain:**  $-0.4$  to  $-0.2$
- **mountain with snow on the flat parts:**  $> -0.2$

See if you can get your mountain to look like the challenge project in the projects directory for this chapter.

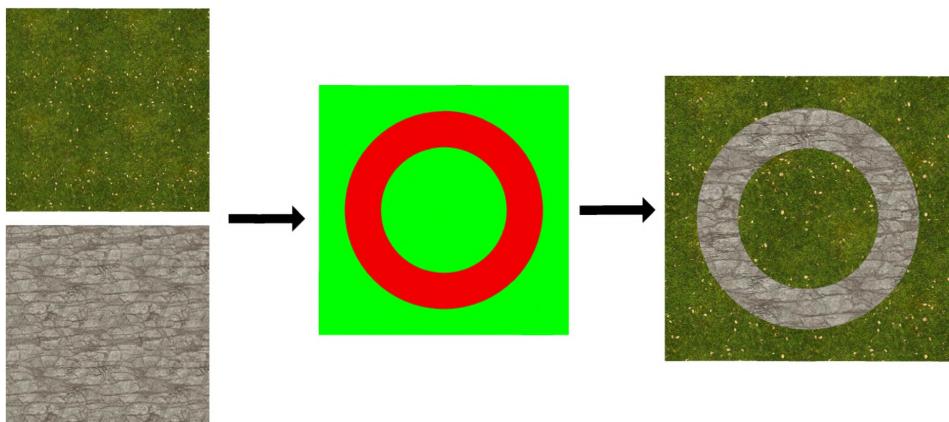
## Key Points

- Tessellation utilizes a tessellator chip on the GPU to create extra vertices.
- You send patches to the GPU rather than vertices. The tessellator then breaks down these patches to smaller triangles.
- A patch can be either a triangle or a quad.
- The tessellation pipeline has an extra stage of setting edge and inside factors in a tessellation kernel. These factors decide the number of vertices that the tessellator should create.
- The vertex shader handles the vertices created by the tessellator.
- Vertex displacement uses a grayscale texture to move the vertex, generally in the y direction.
- The Sobel Metal Performance Shader takes a texture and generates a new texture that defines the slope of a pixel.

## Where to Go From Here?

With very steep displacement, there can be lots of texture stretching between vertices. There are various algorithms to overcome this, and you can find one in Apple's excellent sample code: Dynamic Terrain with Argument Buffers at [https://developer.apple.com/documentation/metal/fundamental\\_components/gpu\\_resources/dynamic\\_terrain\\_with\\_argument\\_buffers](https://developer.apple.com/documentation/metal/fundamental_components/gpu_resources/dynamic_terrain_with_argument_buffers). This is a complex project that showcases argument buffers, but the dynamic terrain portion is interesting.

There's another way to do blending. Instead of using `mix()`, the way you did in the challenge, you can use a texture map to define the different regions. This is known as **texture splatting**. You create a **splat map** with the red, blue and green channels describing up to three textures and where to use them.



*A splat map*

With all of the techniques for reading and using textures that you've learned so far, texture splatting shouldn't be too difficult for you to implement.

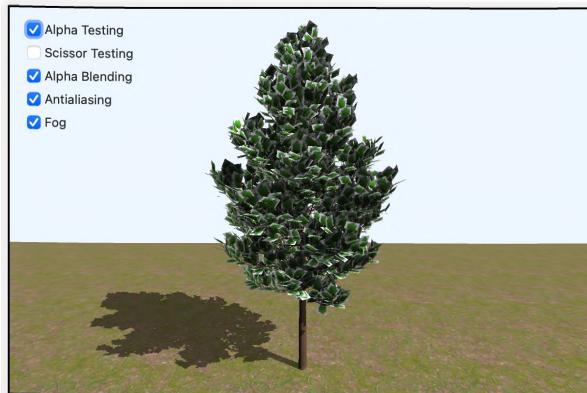
# Chapter 20: Fragment Post-Processing

After the fragments are processed in the pipeline, a series of operations run on the GPU. These operations are sometimes referred to as **Per-sample Processing** ([https://www.khronos.org/opengl/wiki/Per-Sample\\_Processing](https://www.khronos.org/opengl/wiki/Per-Sample_Processing)) and include: alpha testing, depth testing, stencil testing, scissor testing, blending and anti-aliasing. You've already encountered a few of these operations in earlier chapters, such as depth testing and stencil testing. Now it's time to revisit those concepts while also learning about the others.



# The Starter App

- In Xcode, open the starter app for this chapter, and build and run the app.



*The starter app*

The standard forward renderer renders the scene using the PBR shader. This scene has a tree and ground plane, along with an extra window model that you'll add later in this chapter. You can use the options at the top-left of the screen to toggle the post-processing effects. Those effects aren't active yet, but they will be soon!

Submesh and Model now accepts an optional texture to use as an opacity map. Later in this chapter, you'll update the PBR shader function to take into account a model's opacity. If you need help adding textures to your renderer, review Chapter 11, “Maps & Materials”.

## Using Booleans in a C Header File

In `Renderer.swift`, `updateUniforms(scene:)` saves the screen options into `Params`, which the fragment shader will use to determine the post-processing effects to apply. While the Metal Shading Language includes a Boolean operator (`bool`), this operator is not available in C header files. In the `Shaders` group included with this starter project, is `stdbool.h`. This file defines a `bool`, which `Common.h` imports. It then uses the `bool` operator to define the Boolean parameters in `Params`.

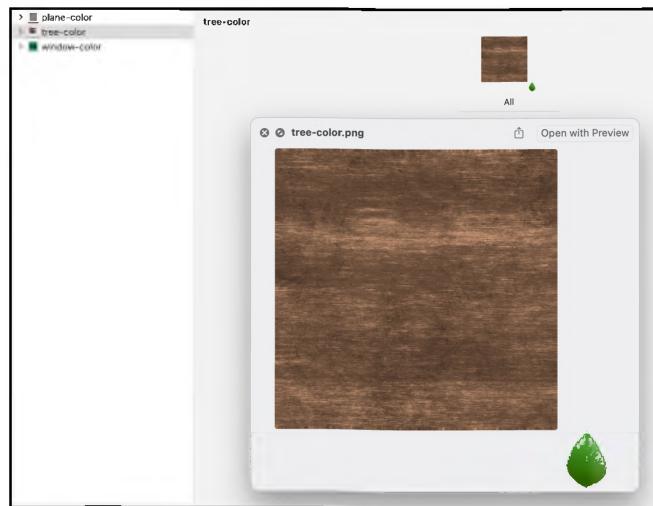
# Alpha Testing

Move closer to the tree using the scroll wheel or the two-finger gesture on your trackpad, and you'll notice the leaves look a little odd.



*Opaque edge around leaves*

- Open **Textures.xcassets**, and select **tree-color**. Preview the texture by pressing the space bar.



*Tree-color texture*

The area of the texture surrounding the leaf is transparent, yet it renders as either white or black, depending on the device.

To make the leaves look more natural, you'll render the transparent part of the texture as transparent in the scene. However, before making this change, it's important to understand the difference between transparent, translucent, and opaque objects.

A **transparent** object allows light to entirely pass through it. A **translucent** object distorts light as it passes through it. An **opaque** object does not allow any light to pass through it. Most objects in nature are opaque. Objects like water, glass and plastic are translucent.

Digital colors are formed using a combination of the three primary colors: red, green and blue — hence the color scheme RGB. However, there's a fourth component you can add to the color definition: **alpha**. Alpha ranges from 0 (fully transparent) to 1 (fully opaque). A common practice in determining transparency is to check the alpha property and ignore values below a certain threshold. This technique is known as **alpha testing**.

- Open **PBR.metal**. In `fragment_PBR`, locate the conditional closure where you set `material.baseColor`.
- Replace the contents of `if (!is_null_texture(baseColorTexture)) {}` with:

```
float4 color = baseColorTexture.sample(  
    textureSampler,  
    in.uv * params.tiling);  
if (params.alphaTesting && color.a < 0.1) {  
    discard_fragment();  
    return 0;  
}  
material.baseColor = color.rgb;
```

With this change, you now read in the alpha value of the color as well as the RGB values. If the alpha is less than a `0.1` threshold, and you're performing alpha testing, then you discard the fragment. The GPU ignores the returned `0` and stops processing the fragment.

- Build and run the app, and toggle Alpha Testing to see the difference.



*Alpha testing*

That's much better! Now, when you get closer to the tree, you'll notice the white background around the leaves is gone.

## Depth Testing

Depth testing compares the depth value of the current fragment to one stored in the framebuffer. If a fragment is farther away than the current depth value, this fragment fails the depth test and is discarded since it's occluded by another fragment. You learned about depth testing in Chapter 7, “The Fragment Function”.

## Stencil Testing

Stencil testing compares the value stored in a stencil attachment to a masked reference value. If a fragment makes it through the mask it's kept, otherwise it's discarded. You learned about stencil testing in Chapter 15, “Tile-Based Deferred Rendering”.

## Scissor Testing

If you only want to render part of the screen, you can tell the GPU to render only within a particular rectangle. This is much more efficient than rendering the entire screen. The scissor test checks whether a fragment is inside a defined 2D area known as the **scissor rectangle**. If the fragment falls outside of this rectangle, it's discarded.

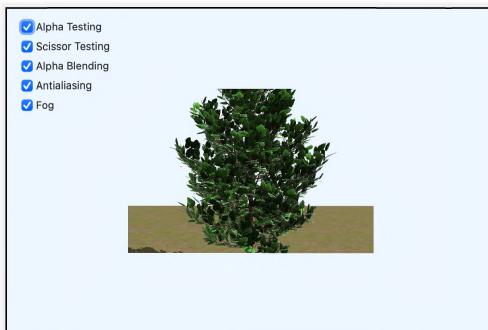
► Open **ForwardRenderPass.swift**, which is where you set up your render command encoder to draw the models.

► In `draw(commandBuffer:scene:uniforms:params:)`, before `for model in scene.models`, add this:

```
if params.scissorTesting {  
    let marginWidth = Int(params.width) / 4  
    let marginHeight = Int(params.height) / 4  
    let width = Int(params.width) / 2  
    let height = Int(params.height) / 2  
    let rect = MTLScissorRect(  
        x: marginWidth, y: marginHeight, width: width, height:  
        height)  
    renderEncoder.setScissorRect(rect)  
}
```

Here, you set the scissor rectangle to half the width and height of the current metal view.

► Build and run the app, and turn on **Scissor Testing**.



*Scissor testing*

Keep in mind that any objects rendered before you set the scissor rectangle are not affected. This means that you can choose to render within a scissor rectangle only for selected models.

## Alpha Blending

Alpha blending is different from alpha testing in that the latter only works with total transparency. In that case, all you have to do is discard fragments. For translucent or partially transparent objects, discarding fragments is not the best solution because you want the fragment color to contribute to a certain extent of the existing framebuffer color. You don't just want to replace it. You had a taste of blending in Chapter 14, "Deferred Rendering", when you blended the result of your point lights.

The formula for alpha blending is as follows:

$$\vec{C}_b = \alpha_1 * \vec{C}_s + \alpha_2 * \vec{C}_d$$

Going over this formula:

- **C<sub>s</sub>**: Source color. The current color you just added to the scene.
- **C<sub>d</sub>**: Destination color. The color that already exists in the framebuffer.
- **C<sub>b</sub>**: Final blended color.
- **α1** and **α2**: The alpha (opacity) factors for the source and destination color, respectively.

The final blended color is the result of adding the products between the two colors and their opacity factors. The source color is the fragment color you put in front, and the destination color is the color already existing in the framebuffer.

Often the two factors are the inverse of each other transforming this equation into linear color interpolation:

$$\vec{C}_b = \alpha * \vec{C}_s + (1 - \alpha) * \vec{C}_d$$

All right, time to install a glass window in front of the tree.

► Open **GameScene.swift**. In **init()**, change **models = [ground, tree]** to:

```
window.position = [0, 3, -1]
models = [window, ground, tree]
```



- Build and run the app, and you'll see the window:

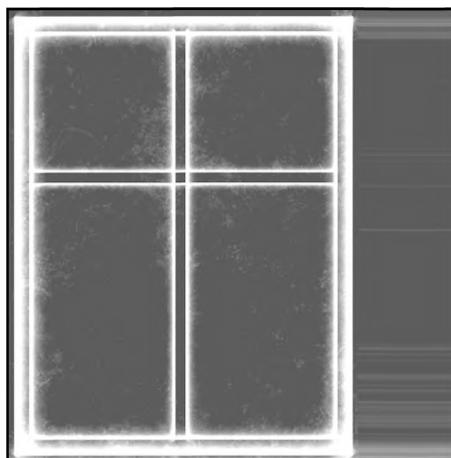


*The window in the scene*

You can't yet view the tree through the window, but you'll fix that with blending. There are two ways to work with blending: the programmable way and the fixed-function way. You used programmable blending with color attachments in Chapter 15, “Tile-Based Deferred Rendering”. In this chapter, you'll use fixed-function blending.

## Opacity

To define transparency in models, you either create a grayscale texture known as an **opacity map**, or you define **opacity** in the submesh's material. The window's glass group has an opacity map where white means fully opaque, and black means fully transparent.



*The window's opacity map*

# Blending

To implement blending, you need a second pipeline state in your render pass. You'll still use the same shader functions, but you'll turn on blending in the GPU.

- Open **Pipelines.swift**, and copy `createForwardPSO()` to a new method.
- Rename the new method to `createForwardTransparentPSO()`.
- In `createForwardTransparentPSO()`, after setting `pipelineDescriptor.colorAttachments[0].pixelFormat`, add this:

```
// 1
let attachment = pipelineDescriptor.colorAttachments[0]
// 2
attachment?.isBlendingEnabled = true
// 3
attachment?.rgbBlendOperation = .add
// 4
attachment?.sourceRGBBlendFactor = .sourceAlpha
// 5
attachment?.destinationRGBBlendFactor = .oneMinusSourceAlpha
```

With this code, you:

1. Grab the first color attachment from the render pipeline descriptor. The color attachment is a color render target that specifies the color configuration and color operations associated with a render pipeline. The render target holds the drawable texture where the rendering output goes.
2. Enable blending on the attachment.
3. Specify the blending type of operation used for color. Blend operations determine how a source fragment is combined with a destination value in a color attachment to determine the pixel value to be written.
4. Specify the blend factor used by the source color. A blend factor is how much the color will contribute to the final blended color. If not specified, this value is always 1 (`.one`) by default.
5. Specify the blend factor used by the destination color. If not specified, this value is always 0 (`.zero`) by default.



**Note:** There are quite a few blend factors available to use other than `sourceAlpha` and `oneMinusSourceAlpha`. For a complete list of options, consult Apple's official page for Blend Factors (<https://developer.apple.com/documentation/metal/mltblendfactor>).

- Open `ForwardRenderPass.swift`, and add a new property to `ForwardRenderPass`:

```
var transparentPSO: MTLRenderPipelineState
```

- In `init(view:)`, add this:

```
transparentPSO = PipelineStates.createForwardTransparentPSO()
```

You initialized the new pipeline state object with your new pipeline creation method.

- In `draw(commandBuffer:scene:uniforms:params:)`, replace `renderEncoder.setRenderPipelineState(pipelineState)` with:

```
renderEncoder.setRenderPipelineState(transparentPSO)
```

You temporarily replace the pipeline state with your new one. Blending is always enabled now.

- Open `PBR.metal`. In `fragment_PBR`, after the conditional where you set `material.baseColor`, add this:

```
if (params.alphaBlending) {
    if (!is_null_texture(opacityTexture)) {
        material.opacity =
            opacityTexture.sample(textureSampler, in.uv).r;
    }
}
```

If you have the alpha blending option turned on, read the value from a provided opacity texture. If no texture is provided, you'll use the default from `material`, loaded with the model's submesh.

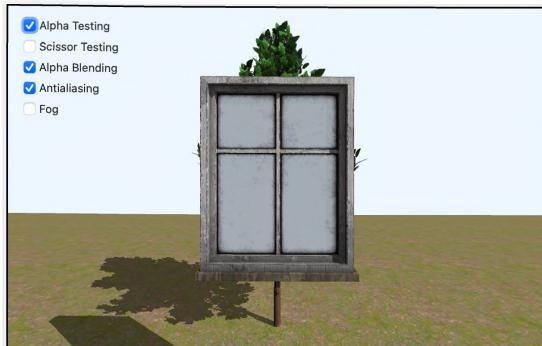
- At the end of `fragment_PBR`, replace the return value with:

```
return float4(diffuseColor + specularColor, material.opacity);
```

You return the opacity value in the alpha channel.



- Build and run the app.



*Opacity not working*

Even though you've set blending in the pipeline state and changed the opacity in the fragment function, the opacity doesn't appear to be working. The glass has changed color from the previous render. This indicates that the transparency is actually working. It's showing the view's clear color through the glass, rather than the tree and the ground.

- Open **GameScene.swift**. In `init()`, change `models = [window, ground, tree]` to:

```
models = [ground, tree, window]
```

- Build and run the app.



*Opacity is working*

The opacity is working, and if you zoom in, you can see the weathering on the old glass.

## Transparent Mesh Rendering Order

The blending order is important. Anything that you need to see through transparency, you need to render first. However, it may not always be convenient to work out exactly which models require blending. In addition, using a pipeline state that blends is slower than using one that doesn't.

- Undo the previous change to `models` so that you render the window first again.

First set up your models to indicate whether any of the submeshes aren't opaque.

- Open `Submesh.swift`, and add a computed property to `Submesh`:

```
var transparency: Bool {  
    return textures.opacity != nil || material.opacity < 1.0  
}
```

`transparency` is true if the submesh textures or material indicate transparency.

- Open `Model.swift`, and add a new property:

```
let hasTransparency: Bool
```

To initialize this property, you'll process all of the model's submeshes, and if any of them have transparency set to `true`, then the model is not fully opaque.

- At the end of `init(name:)`, add this:

```
hasTransparency = meshes.contains { mesh in  
    mesh.submeshes.contains { $0.transparency }  
}
```

If any of the model's submeshes have transparency, you'll process the model during the transparency render phase.

- In `render(encoder:uniforms:params:)`, locate for `submesh` in `mesh.submeshes`.

- At the top of the `for` loop, add this:

```
if submesh.transparency != params.transparency { continue }
```

You only render the submesh if its transparency matches the current transparency in the render loop.



- Open **Common.h**, and add a new property to **Params**:

```
bool transparency;
```

You'll use this property to track when you're currently rendering transparent submeshes.

- Open **ForwardRenderPass.swift**. In

`draw(commandBuffer:scene:uniforms:params:)`, change the pipeline state back to what it was originally:

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

- Locate `for model in scene.models`. Before the `for` loop, add this:

```
var params = params
params.transparency = false
```

In the render loop, you'll only render submeshes with no transparency.

- Now, before `renderEncoder.endEncoding()`, add this:

```
// transparent mesh
renderEncoder.pushDebugGroup("Transparency")
let models = scene.models.filter {
    $0.hasTransparency
}
params.transparency = true
if params.alphaBlending {
    renderEncoder.setRenderPipelineState(transparentPSO)
}
for model in models {
    model.render(
        encoder: renderEncoder,
        uniforms: uniforms,
        params: params)
}
renderEncoder.popDebugGroup()
```

Here, you filter the `scene.models` array to find only those models that have a transparent submesh. You then change the pipeline state to use alpha blending, and render the filtered models.



► Build and run the app.



*Alpha blending*

You can now see through your window.

**Note:** If you have several transparent meshes overlaying each other, you'll need to sort them to ensure that you render them in strict order from back to front.

► In the app, turn off **Alpha Blending**.

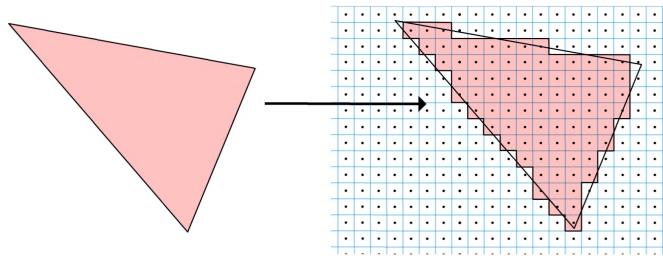
At the end of the render loop, the pipeline state doesn't switch to the blending one, so the window becomes opaque again.



*Alpha blending turned off*

# Antialiasing

Often, rendered models show slightly jagged edges that are visible when you zoom in. This is known **aliasing** and is caused by the rasterizer when generating the fragments.



*Rasterizing a triangle*

If you look at the edges of a triangle — or any straight line with a slope — you'll notice the line doesn't always go precisely through the center of a pixel. Some pixels are colored above the line and some below it. The solution to fixing aliasing is to use **antialiasing**. Antialiasing applies techniques to render smoother edges.

By default, the pipeline uses one sample point (subpixel) for each pixel that is close to the line to determine if they meet. However, it's possible to use four or more points for increased accuracy of intersection determination. This is known as **Multisample Antialiasing** (MSAA), and it's more expensive to compute.

Next, you're going to configure the fixed-function MSAA on the pipeline and enable antialiasing on both the tree and the window.

- Open **Pipelines.swift**, and duplicate both `createForwardPSO()` and `createForwardTransparentPSO()`.
- Name them `createForwardPSO_MSAA()` and `createForwardTransparentPSO_MSAA()`, respectively.
- In both new methods, before the return, add this:

```
pipelineDescriptor.sampleCount = 4
```

- Open **ForwardRenderPass.swift**, and add two new properties:

```
var pipelineState_MSAA: MTLRenderPipelineState  
var transparentPSO_MSAA: MTLRenderPipelineState
```

- In `init()`, initialize the new pipeline states with your new pipeline state creation methods:

```
pipelineState_MSAA = PipelineStates.createForwardPSO_MSAA()
transparentPSO_MSAA =
    PipelineStates.createForwardTransparentPSO_MSAA()
```

- At the top of `draw(commandBuffer:scene:uniforms:params:)`, add this:

```
let pipelineState = params.antialiasing ?
    pipelineState_MSAA : pipelineState
let transparentPSO = params.antialiasing ?
    transparentPSO_MSAA : transparentPSO
```

Depending upon whether the user has selected **Antialiasing**, you set the different pipeline state.

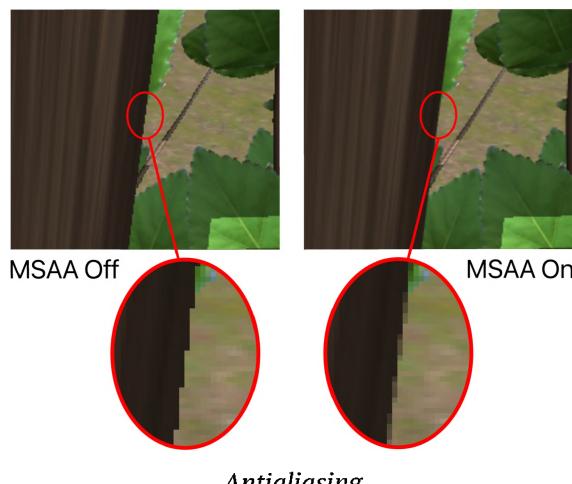
The render target texture must match the same sample count as the pipeline state.

- Open **Renderer.swift**, and at the top of `draw(scene:in:)` before the guard, add this:

```
view.sampleCount = options.antialiasing ? 4 : 1
```

The current render pass descriptor will use the sample count to create the render target texture with the correct antialiasing.

- Build and run the app. On modern retina devices, this effect can be quite difficult to see. But, if you zoom in to a straight line on a slope — such as the tree trunk — and toggle **Antialiasing**, you may notice the difference.



# Fog

Let's have a bit more fun and add some fog to the scene!

Fog is quite useful in rendering. First, it serves as a far delimiter for rendered content. The renderer can ignore objects that get lost in the fog since they're not visible anymore. Second, fog helps you avoid the popping-up effect that can happen when objects that are farther away from the camera "pop" into the scene as the camera moves closer. With fog, you can make their appearance into the scene more gradual.

**Note:** Fog isn't a post-processing effect, it's added in the fragment shader.

► Open **PBR.metal**, and add a new function before **fragment\_PBR**:

```
float4 fog(float4 position, float4 color) {
    // 1
    float distance = position.z / position.w;
    // 2
    float density = 0.2;
    float fog = 1.0 - clamp(exp(-density * distance), 0.0, 1.0);
    // 3
    float4 fogColor = float4(1.0);
    color = mix(color, fogColor, fog);
    return color;
}
```

With this code, you:

1. Calculate the depth of the fragment position.
2. Define a distribution function that the fog will use next. It's the inverse of the clamped (between 0 and 1) product between the fog density and the depth calculated in the previous step.
3. Mix the current color with the fog color (which you deliberately set to white) using the distribution function defined in the previous step.

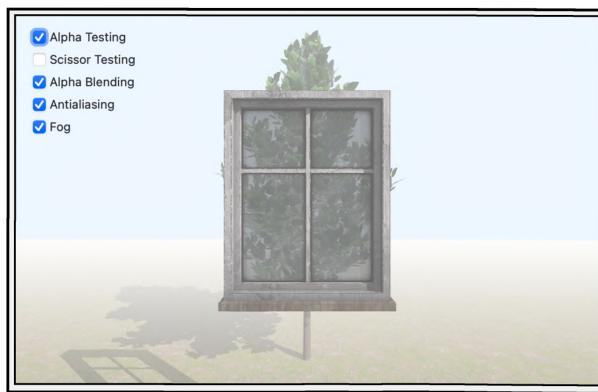


- Change the return value of `fragment_PBR` to:

```
float4 color =  
    float4(diffuseColor + specularColor, material.opacity);  
if (params.fog) {  
    color = fog(in.position, color);  
}  
return color;
```

Here, you include the fog value in the final color.

- Build and run the app.



Perfect, the entire scene is foggy. The closer you get to the tree, the less dense the fog. The same happens to the ground. Like with real fog, the closer you get to an object, the easier it is to see it. Check it out: get closer to the tree, and you'll see it a lot better.

Because this effect is worked in the fragment shader, the sky is not affected by fog. The sky color is coming from the MTKView instead of being rendered. In the next chapter, you'll create a rendered sky that you can affect with fog.

## Key Points

- Per-sample processing takes place in the GPU pipeline after the GPU processes fragments.
- Using `discard_fragment()` in the fragment function halts further processing on the fragment.
- To render only part of the texture, you can define a 2D scissor rectangle. The GPU discards any fragments outside of this rectangle.
- You set up the pipeline state object with blending when you require transparency. You can then set the alpha value of the fragment in the fragment function. Without blending in the pipeline state object, all fragments are fully opaque, no matter their alpha value.
- Multisample antialiasing improves render quality. You set up MSAA with the `sampleCount` in the pipeline state descriptor.
- You can add fog with some clever distance shading in the fragment function.

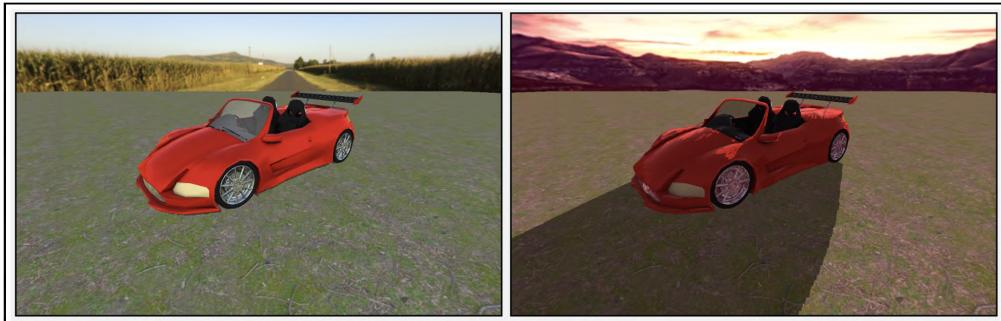
## Where to Go From Here?

**Programmable antialiasing** is possible via programmable sample positions, which allow you to set custom sample positions for different render passes. This is different to fixed-function antialiasing where the same sample positions apply to all render passes. For further reading, you can review Apple's Positioning Samples Programmatically ([https://developer.apple.com/documentation/metal/mlrenderpassdescriptor/positioning\\_samples\\_programmatically](https://developer.apple.com/documentation/metal/mlrenderpassdescriptor/positioning_samples_programmatically)) article.

# Chapter 21: Imaged-Based Lighting

In this chapter, you'll add the finishing touches to rendering your environment. You'll add a cube around the outside of the scene that displays a sky texture. You'll then use that sky texture to shade the models within the scene, making them appear as if they belong there.

Look at the following comparison of two renders.



*The final and challenge renders*

This comparison demonstrates how you can use the same shader code, but change the sky image to create different lighting environments.

## The Starter Project

- In Xcode, open the starter project for this chapter.
- Build and run the app.



*The starter project*

The project contains the forward renderer with transparency from the previous chapter. The scene uses an arcball camera, and contains a ground plane and car. The scene lighting consists of one sunlight and the PBR shader provides the shading.

There are a few additional files that you'll use throughout the chapter. **Common.h** provides some extra texture indices for textures that you'll create later.

Aside from the darkness of the lighting, there are some glaring problems with the render:

- All metals, such as the metallic wheel hubs, look dull. Pure metals reflect their surroundings, and there are currently no surroundings to reflect.
- Where the light doesn't directly hit the car, the color is pure black. This happens because the app doesn't provide any ambient light. Later in this chapter, you'll use the skylight as global ambient light.

## The Skybox

Currently, the sky is a single color, which looks unrealistic. By adding a 360° image surrounding the scene, you can easily place the action in a desert or have snowy mountains as a backdrop. To do this, you'll create a skybox cube that surrounds the entire scene.

This skybox cube is the same as an ordinary model, but instead of viewing it from the outside, the camera is at the center of the cube looking out. You'll texture the cube with a **cube texture**, which gives you a cheap way of creating a complete environment.

You may think the cube will be distorted at the corners, but as you'll see, each fragment of the cube will render at an effectively infinite distance, and no distortion will occur. Cube maps are much easier to create than spherical ones and are hardware optimized.

► In the **Geometry** group, create a new Swift file for the skybox class named **Skybox.swift**.

► Replace the default code with:

```
import MetalKit

struct Skybox {
    let mesh: MTKMesh
    var skyTexture: MTLTexture?
    let pipelineState: MTLRenderPipelineState
    let depthStencilState: MTLDepthStencilState?
}
```

Going through the skybox properties:

- **mesh**: A cube that you'll create using a Model I/O primitive.
- **skyTexture**: A cube texture of the name given in the initializer. This is the texture that you'll see in the background.
- **pipelineState**: The skybox needs a simple vertex and fragment function, therefore it needs its own pipeline.
- **depthStencilState**: Each pixel of the skybox will be positioned at the very edge of normalized clip space. The default depth stencil state in **RenderPass.swift** renders the fragment if the fragment is less than the current depth value. The skybox depth stencil should test less than or equal to the current depth value. You'll see why shortly.

Your project won't compile until you've initialized all stored properties.

► Add the initializer to Skybox:

```
init(textureName: String?) {
    let allocator =
        MTKMeshBufferAllocator(device: Renderer.device)
```

```

let cube = MDLMesh(
    boxWithExtent: [1, 1, 1],
    segments: [1, 1, 1],
    inwardNormals: true,
    geometryType: .triangles,
    allocator: allocator)
do {
    mesh = try MTKMesh(
        mesh: cube, device: Renderer.device)
} catch {
    fatalError("failed to create skybox mesh")
}
}

```

Here, you create a cube mesh. Notice that you set the normals to face inwards. That's because the whole scene will appear to be *inside* the cube.

- In the **Render Passes** group, open **Pipelines.swift**, and add a new method:

```

static func createSkyboxPS0(
    vertexDescriptor: MTLVertexDescriptor?
) -> MTLRenderPipelineState {
    let vertexFunction =
        Renderer.library?.makeFunction(name: "vertex_skybox")
    let fragmentFunction =
        Renderer.library?.makeFunction(name: "fragment_skybox")
    let pipelineDescriptor = MTLRenderPipelineDescriptor()
    pipelineDescriptor.vertexFunction = vertexFunction
    pipelineDescriptor.fragmentFunction = fragmentFunction
    pipelineDescriptor.colorAttachments[0].pixelFormat =
        Renderer.colorPixelFormat
    pipelineDescriptor.depthAttachmentPixelFormat = .depth32Float
    pipelineDescriptor.vertexDescriptor = vertexDescriptor
    return createPS0(descriptor: pipelineDescriptor)
}

```

When you create the pipeline state, you'll pass in the skybox cube's Model I/O vertex descriptor. You'll write the two new shader functions shortly.

- Open **Skybox.swift**, and add a new method to create the depth stencil state:

```

static func buildDepthStencilState() -> MTLDepthStencilState? {
    let descriptor = MTLDepthStencilDescriptor()
    descriptor.depthCompareFunction = .lessEqual
    descriptor.isDepthWriteEnabled = true
    return Renderer.device.makeDepthStencilState(
        descriptor: descriptor)
}

```

This creates the depth stencil state with the less than or equal comparison method mentioned earlier.

- Complete the initialization by adding the following code to the end of `init(textureName:)`:

```
pipelineState = PipelineStates.createSkyboxPSO(  
    vertexDescriptor: MTKMetalVertexDescriptorFromModelIO(  
        cube.vertexDescriptor))  
depthStencilState = Self.buildDepthStencilState()
```

You initialize the skybox's pipeline state with the vertex descriptor provided by Model I/O.

## Rendering the Skybox

- Still in `Skybox.swift`, create a new method to perform the skybox rendering:

```
func render(  
    encoder: MTLRenderCommandEncoder,  
    uniforms: Uniforms  
) {  
    encoder.pushDebugGroup("Skybox")  
    encoder.setRenderPipelineState(pipelineState)  
    // encoder.setDepthStencilState(depthStencilState)  
    encoder.setVertexBuffer(  
        mesh.vertexBuffers[0].buffer,  
        offset: 0,  
        index: 0)  
}
```

Here, you set up the render command encoder with the properties you initialized. Leave the depth stencil state line commented out for the moment.

- Add this code at the end of `render(encoder:uniforms:)`:

```
var uniforms = uniforms  
uniforms.viewMatrix.columns.3 = [0, 0, 0, 1]  
encoder.setVertexBytes(  
    &uniforms,  
    length: MemoryLayout<Uniforms>.stride,  
    index: UniformsBuffer.index)
```

When you render a scene, you multiply each model's matrix with the view matrix and the projection matrix. As you move through the scene, it appears as if the camera is moving through the scene, but in fact, the whole scene is moving around the camera.

You don't want the skybox to move, so you zero out column 3 of `viewMatrix` to remove the camera's translation.

However, you do still want the skybox to rotate with the rest of the scene, and also render with projection, so you send the uniform matrices to the GPU.

- Add the following code after the code you just added:

```
let submesh = mesh.submeshes[0]
encoder.drawIndexedPrimitives(
    type: .triangle,
    indexCount: submesh.indexCount,
    indexType: submesh.indexType,
    indexBuffer: submesh.indexBuffer.buffer,
    indexBufferOffset: 0)
encoder.popDebugGroup()
```

Here, you draw the cube's submesh.

## The Skybox Shader Functions

In the **Shaders** group, create a new Metal file named **Skybox.metal**.

- Add the following code to the new file:

```
#import "Common.h"

struct VertexIn {
    float4 position [[attribute(Position)]];
};

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

The structures are simple so far – you need a position in and a position out.

- Add the shader functions:

```
vertex VertexOut vertex_skybox(
    const VertexIn in [[stage_in]],
    constant Uniforms &uniforms [[buffer(UniformsBuffer)]])
{
    VertexOut out;
    float4x4 vp = uniforms.projectionMatrix * uniforms.viewMatrix;
    out.position = (vp * in.position).xyww;
    return out;
}
```

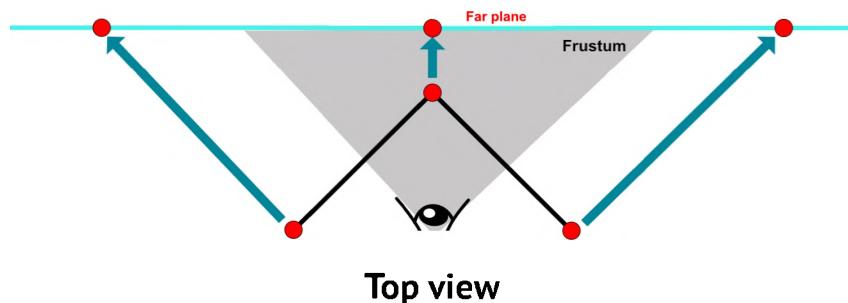
```
fragment half4 fragment_skybox(
    VertexOut in [[stage_in]]) {
    return half4(1, 1, 0, 1);
}
```

Here, you create two very simple shaders — the vertex function moves the vertices to the projected position, and the fragment function returns yellow. This is a temporary color, which is starting enough that you'll be able to see where the skybox renders.

Notice in the vertex function that you **swizzled** the `xyzw` position to `xyww`. To place the sky as far away as possible, it needs to be at the very edge of NDC.

During the change from clip space to NDC, the coordinates are all divided by `w` during the perspective divide stage. This will now result in the `z` coordinate being `1`, which will ensure that the skybox renders *behind* everything else within the scene.

The following diagram shows the skybox in camera space rotated by 45°. After projection and the perspective divide, the vertices will be flat against the far NDC plane.



## Integrating the Skybox Into the Scene

- Open `GameScene.swift`, and add a new property to `GameScene`:

```
let skybox: Skybox?
```

- Add the following code to the top of `init()`:

```
skybox = Skybox(textureName: nil)
```

You haven't written the code for the skybox texture yet, but soon you'll set it up so that `nil` will generate a physically simulated sky, and providing a texture name will load that sky texture.

- In the **Render Passes** group, open **ForwardRenderPass.swift**, and in `draw(commandBuffer:scene:uniforms:params:)`, locate `// transparent mesh.`

- Add the following code *before* that comment:

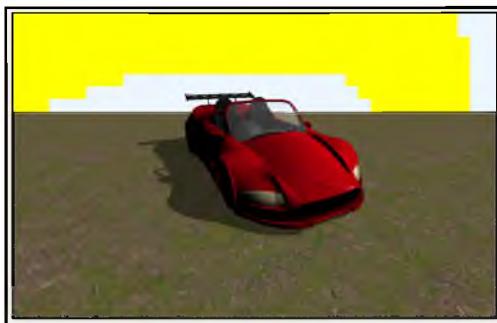
```
scene.skybox?.render(  
    encoder: renderEncoder,  
    uniforms: uniforms)
```

You render the skybox only during the opaque pass, after the opaque meshes.

It may seem odd that you're rendering the skybox *after* rendering the scene models, when it's going to be the object that's behind everything else. Remember early-Z testing from Chapter 3, "The Rendering Pipeline": when objects are rendered, most of the skybox fragments will be behind them and will fail the depth test. Therefore, it's more efficient to render the skybox as late as possible. You have to render before the transparent pass, so that any transparency will include the skybox texture.

You've now integrated the skybox into the rendering process.

- Build and run the app to see the new yellow sky.

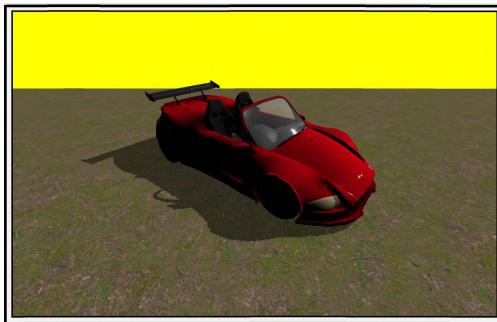


*A flickering sky*

As you rotate the scene, the yellow sky flickers and shows the blue of the metal view's clear color. This happens because the current depth stencil state is from **ForwardRenderPass**, and it's comparing new fragments to *less* than the current depth buffer. The skybox coordinates are right on the edge, so sometimes they're *equal* to the edge of clip space.

- Open **Skybox.swift**, and in `render(encoder:uniforms:)`, uncomment `encoder.setDepthStencilState(depthStencilState)`, and build and run the app again.

This time, the depth comparison is correct, and the sky is the solid yellow returned from the skybox fragment shader.



*A yellow sky*

## Procedural Skies

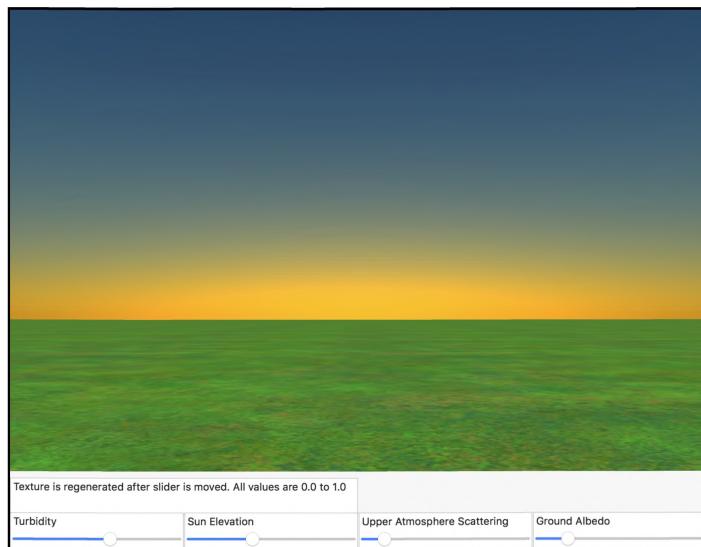
Yellow skies might be appropriate on a different planet, but how about a procedural sky? A procedural sky is one built out of various parameters such as weather conditions and time of day. Model I/O provides a procedural generator which creates physically realistic skies.

- Before exploring this API further, open and run **skybox.playground** in the resources folder for this chapter.

This scene contains only a ground plane and a skybox. Use your mouse or trackpad to reorient the scene, and experiment with the sliders under the view to see how you can change the sky depending on:

- **turbidity:** Haze in the sky. 0.0 is a clear sky. 1.0 spreads the sun's color.
- **sun elevation:** How high the sun is in the sky. 0.5 is on the horizon. 1.0 is overhead.
- **upper atmosphere scattering:** Atmospheric scattering influences the color of the sky from reddish through orange tones to the sky at midday.
- **ground albedo:** How clear the sky is. 0 is clear, while 10 can produce intense colors. It's best to keep turbidity and upper atmosphere scattering low if you have high albedo.

As you move the sliders, the result is printed in the debug console so you can record these values for later use. See if you can create a sunrise:



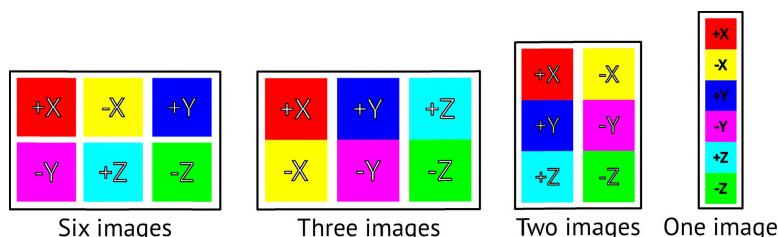
*A sunrise*

This playground uses Model I/O to create an `MDLSkyCubeTexture`. From this, the playground creates an `MTLTexture` and applies this as a cube texture to the sky cube. You'll now do this in your project.

## Cube Textures

Cube textures are similar to the 2D textures that you've already been using. 2D textures map to a quad and have two texture coordinates, whereas cube textures consist of six 2D textures: one for each face of the cube. You sample the textures with a 3D vector.

The easiest way to load a cube texture into Metal is to use Model I/O's `MDLTexture` initializer. When creating cube textures, you can arrange the images in various combinations:

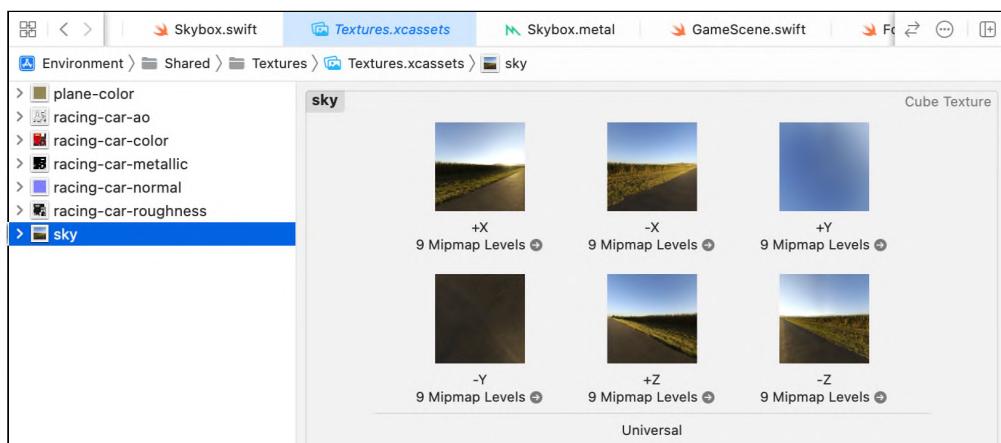


Alternatively, you can create a cube texture in an asset catalog and load the six images there.

- Back in your project, in the **Textures** group, open **Textures.xcassets**. **sky** is a sky texture complete with mipmaps.

The sky should always render on the base mipmap level 0, but you'll see later why you would use the other mipmaps.

Aside from there being six images to one texture, moving the images into the asset catalog and creating the mipmaps is the same process as described in Chapter 8, "Textures".



*The sky texture in the asset catalog*

## Adding the Procedural Sky

You'll use these sky textures shortly, but for now, you'll add a procedural sky to your scene.

- Open **Skybox.swift**, and add these properties to **Skybox**:

```
struct SkySettings {
    var turbidity: Float = 0.28
    var sunElevation: Float = 0.6
    var upperAtmosphereScattering: Float = 0.4
    var groundAlbedo: Float = 0.8
}
var skySettings = SkySettings()
```

You can use the values from the appropriate sliders in the playground if you prefer.

- Now, add the following method:

```
func loadGeneratedSkyboxTexture(dimensions: SIMD2<Int32>)
    -> MTLTexture? {
    var texture: MTLTexture?
    let skyTexture = MDLSkyCubeTexture(
        name: "sky",
        channelEncoding: .float16,
        textureDimensions: dimensions,
        turbidity: skySettings.turbidity,
        sunElevation: skySettings.sunElevation,
        upperAtmosphereScattering:
            skySettings.upperAtmosphereScattering,
        groundAlbedo: skySettings.groundAlbedo)
    do {
        let textureLoader =
            MTKTextureLoader(device: Renderer.device)
        texture = try textureLoader.newTexture(
            texture: skyTexture,
            options: nil)
    } catch {
        print(error.localizedDescription)
    }
    return texture
}
```

Model I/O uses your settings to create the sky texture. That's all there is to creating a procedurally generated sky texture!

- Call the new method at the end of `init(textureName:)`:

```
if let textureName = textureName {
} else {
    skyTexture = loadGeneratedSkyboxTexture(dimensions: [256,
256])
}
```

You'll add the `if` part of this conditional shortly and load the named texture. The `nil` option provides a default sky.

To render the texture, you'll change the skybox shader function and ensure that the texture gets to the GPU.

- Still in `Skybox.swift`, in `render(encoder:uniforms:)`, add the following code before the draw call:

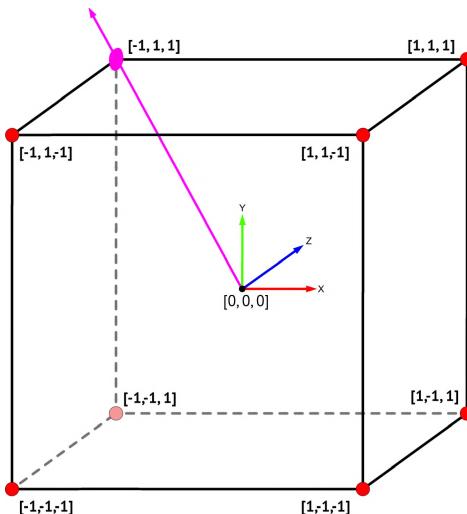
```
encoder.setFragmentTexture(
    skyTexture,
    index: SkyboxTexture.index)
```

The starter project already has the necessary texture enumeration indices set up in **Common.h** for the skybox textures.

► Open **Skybox.metal**, and add a new property to **VertexOut**:

```
float3 textureCoordinates;
```

Generally, when you load a model, you also load its texture coordinates. However, when sampling texels from a cube texture, instead of using a *uv coordinate*, you use a *3D vector*. For example, a vector from the center of any unit cube passes through the far top left corner at  $[-1, 1, 1]$ .



*Skybox coordinates*

Conveniently, even though the skybox's far top-left vertex position is  $[-0.5, 0.5, 0.5]$ , it still lies on the same vector, so you can use the skybox vertex position for the texture coordinates.

► Add this code to **vertex\_skybox** before `return out;`:

```
out.textureCoordinates = in.position.xyz;
```

► Change **fragment\_skybox** to:

```
fragment half4 fragment_skybox(
    VertexOut in [[stage_in]],
    texturecube<half> cubeTexture [[texture(SkyboxTexture)]])
{
    constexpr sampler default_sampler(filter::linear);
```

```
half4 color = cubeTexture.sample(  
    default_sampler,  
    in.textureCoordinates);  
return color;  
}
```

Accessing a cube texture is similar to accessing a 2D texture. You mark the cube texture as `texturecube` in the shader function parameters and sample it using the `textureCoordinates` vector that you set up in the vertex function.

- Build and run the app, and you now have a realistic sky, simulating physics:



*A procedural sky*

## Custom Sky Textures

As mentioned earlier, you can use your own 360° sky textures. The textures included in the starter project were downloaded from Poly Haven (<https://polyhaven.com/hdris>) — a great place to find environment maps. The HDRI has been converted into six tone mapped sky cube textures before adding them to the asset catalog.

**Note:** If you want to create your own skybox textures or load HDRIs (high dynamic range images), you can find out how to do it in `references.markdown` included with this chapter's files.

Loading a cube texture is almost the same as loading a 2D texture.

- Open **TextureController.swift**, and examine `loadCubeTexture(imageName:)`.

Just as with `loadTexture(imageName:)`, you can load either a **cube** texture from the asset catalog or one 2D image consisting of the six faces vertically.

- Open **Skybox.swift**, and at the end of `init(textureName:)`, in the first half of the incomplete conditional, add this:

```
do {
    skyTexture = try TextureController.loadCubeTexture(
        imageName: textureName)
} catch {
    fatalError(error.localizedDescription)
}
```

You load the texture using the given texture name.

- Open **GameScene.swift**, and in `init()`, change the skybox initialization to:

```
skybox = Skybox(textureName: "sky")
```

- Build and run the app to see your new skybox texture.



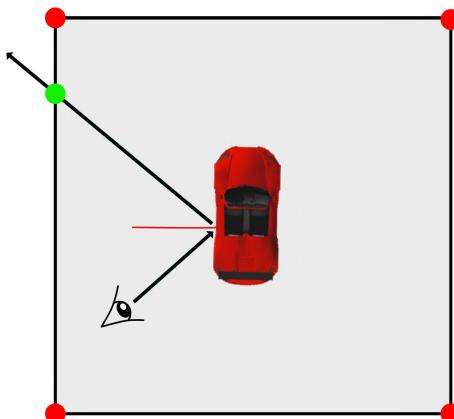
*The skybox*

Notice that as you move about the scene, although the skybox rotates with the rest of the scene, it does not reposition.

You should be careful that the sky textures you use don't have objects that appear to be close, as they will always appear to stay at the same distance from the camera. Sky textures should be for background only. This skybox texture is not a great fit as the background does not match the ground plane.

## Reflection

Now that you have something to reflect, you can easily implement reflection of the sky onto the car. When rendering the car, all you have to do is take the camera view direction, reflect it about the surface normal, and sample the skycube along the reflected vector for the fragment color for the car.



*Reflection*

Included in the starter project is a new fragment shader already set up with textures.

- In the **Shaders** group, open **IBL.metal**, and examine **fragment\_IBL** in that file.

The shader reads all of the possible model textures and sets values for:

- base color
- normal
- roughness
- metallic
- ambient occlusion
- opacity

All of these maps will interact with the sky texture and eventually provide a beautiful render. Currently, the fragment function returns just the base color.

- Open **Pipelines.swift**, and change the fragment function name in `createForwardPS0()` to:

```
let fragmentFunction =  
    Renderer.library?.makeFunction(name: "fragment_IBL")
```

- Build and run the app.



*Color textures only*

The fragment shader renders the car and ground returning the texture base color. The glass windshield has transparency, so uses a different pipeline. It will render transparently with specular highlights, as it did before.

- Open **Skybox.swift**, and add this to Skybox:

```
func update(encoder: MTLRenderCommandEncoder) {  
    encoder.setFragmentTexture(  
        skyTexture,  
        index: SkyboxTexture.index)  
}
```

You send the skybox texture to the fragment shader. You'll add other skybox textures to this method soon.

- Open **ForwardRenderPass.swift**, and in `draw(commandBuffer:scene:uniforms:params:)`, add the following code before `var params = params:`

```
scene.skybox?.update(encoder: renderEncoder)
```

The sky texture is now available to the GPU.

- Open **IBL.metal**, and add the skybox texture to the parameter list for `fragment_IBL`:

```
texturecube<float> skybox [[texture(SkyboxTexture)]]
```

You read the skybox texture using the type `texturecube`. Instead of providing 2D uv coordinates, you'll provide 3D coordinates.

You now calculate the camera's reflection vector about the surface normal to get a vector for sampling the skybox texture. To get the camera's view vector, you subtract the fragment world position from the camera position.

- At the end of `fragment_IBL`, before `return color;`, add this:

```
float3 viewDirection =
    in.worldPosition.xyz - params.cameraPosition;
viewDirection = normalize(viewDirection);
float3 textureCoordinates =
    reflect(viewDirection, normal);
```

Here, you calculate the view vector and reflect it about the surface normal to get the vector for the cube texture coordinates.

- Now, add this:

```
constexpr sampler defaultSampler(filter::linear);
color = skybox.sample(
    defaultSampler, textureCoordinates);
float4 copper = float4(0.86, 0.7, 0.48, 1);
color = color * copper;
```

Here, you sample the skybox texture for a color and multiply it by a copper color.

- Build and run the app.



*Reflections*

The rendered scene now appears to be made of beautifully shiny copper. As you rotate the scene, using your mouse or trackpad, you can see the sky reflected in the scene models.

**Note:** This is not a true reflection since you're only reflecting the sky texture. If you place any objects in the scene, they won't be reflected. However, this reflection is a fast and easy effect, and is often sufficient.

- In `fragment_IBL`, remove the code you just added, i.e., the lines from `constexpr sampler defaultSampler... to color = color * copper;.`

You'll replace this with new lighting code. You'll get a compiler warning on `textureCoordinates` until you use it later.

## Image-Based Lighting

At the beginning of the chapter, there were two problems with the original car render. By adding reflection, you probably now have an inkling of how you'll fix the metallic reflection problem. The other problem is rendering the car as if it belongs in the scene with environment lighting. **IBL** or **Image-Based Lighting** is one way of dealing with this problem.

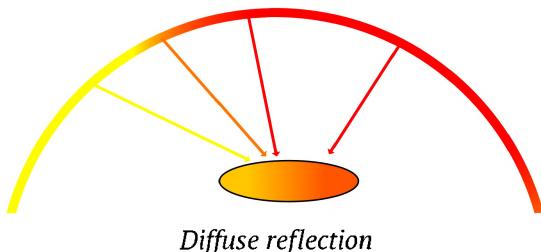
Using the sky image you can extract lighting information. For example, the parts of the car that face the sun in the sky texture should shine more than the parts that face away. The parts that face away shouldn't be entirely dark but should have ambient light filled in from the sky texture.

Epic Games developed a technique for *Fortnite*, which they adapted from Disney's research, and this has become the standard technique for IBL in games today. If you want to be as physically correct as possible, there's a link to their article on how to achieve this included with the **references.markdown** for this chapter.

You'll be doing an approximation of their technique, making use of Model I/O for the diffuse.

## Diffuse Reflection

Light comes from all around us. Sunlight bounces around and colors reflect. When rendering an object, you should take into account the color of the light coming from every direction.



This is somewhat of an impossible task, but you can use **convolution** to compute a cube map called an **irradiance map** from which you can extract lighting information. You won't need to know the mathematics behind this: Model I/O comes to the rescue again!

The diffuse reflection for the car will come from a second texture derived from the sky texture.

- Open **Skybox.swift**, and add a new property to Skybox to hold this diffuse texture:

```
var diffuseTexture: MTLTexture?
```

- To create the diffuse irradiance texture, add this temporary method:

```
mutating func loadIrradianceMap() {
    // 1
    guard let skyCube =
        MDLTexture(cubeWithImagesNamed: ["cube-sky.png"])
    else { return }
    // 2
    let irradiance =
        MDLTexture.irradianceTextureCube(
            with: skyCube,
            name: nil,
            dimensions: [64, 64],
            roughness: 0.6)
    // 3
    let loader = MTKTextureLoader(device: Renderer.device)
    do {
        diffuseTexture = try loader.newTexture(
            texture: irradiance,
```

```
        options: nil)
    } catch {
        fatalError(error.localizedDescription)
    }
}
```

Going through this code:

1. Model I/O currently doesn't load cube textures from the asset catalog, so, in the **Textures** group, your project has an image named **cube-sky.png** with the six faces included in it. Each of the faces is 128 x 128 pixels.
2. Use Model I/O to create the irradiance texture from the source image. Neither source nor destination textures have to be large, as the diffuse color is spread out.
3. Load the resultant MDLTexture to `diffuseTexture`.

► In **Skybox.swift**, add the following to the end of `init(textureName:)`:

```
loadIrradianceMap()
```

► In `update(encoder:)`, add the following:

```
encoder.setFragmentTexture(
    diffuseTexture,
    index: SkyboxDiffuseTexture.index)
```

This code will send the diffuse texture to the GPU.

► Open **IBL.metal**, and add the diffuse texture to the parameter list for `fragment_IBL`:

```
texturecube<float> skyboxDiffuse
[[texture(SkyboxDiffuseTexture)]]
```

► At the end of `fragment_IBL`, add this before `return color;`:

```
float4 diffuse = skyboxDiffuse.sample(textureSampler, normal);
color = diffuse * float4(material.baseColor, 1);
```

The diffuse value doesn't depend on the angle of view, so you sample the diffuse texture using the surface normal. You then multiply the result by the base color.

► Build and run the app. Because of the irradiance convolution, the app may take a minute or so to start. As you rotate about the car, you'll notice it's very slightly brighter where it faces the skybox sun.

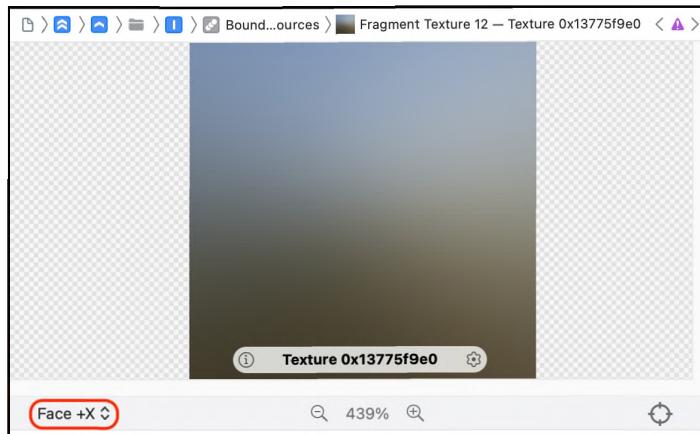


*Diffuse from irradiance*

► Click the **Capture GPU frame** icon to enter the GPU Debugger, and look at the generated irradiance map.

Label	Type	Size	Details
R Render Pipeline 0x1380f...			
Vertex			
box-position	Buffer 0	768 bytes	Offset: 0x0
Buffer 0x13775d510	Buffer 1	189 KB	Offset: 0x0
Buffer 0x13775d670	Buffer 2	379 KB	Offset: 0x0
Buffer 0x13775d7d0	Buffer 3	379 KB	Offset: 0x0
Buffer 0x13775d930	Buffer 4	379 KB	Offset: 0x0
Vertex Bytes	Buffer 11 (Bytes)	64 bytes	
vertex_skybox	Vertex Function		Library 0x60000000c000...
Fragment			
racing-car-color	Texture 0	2048 x 2048	RGBAB8Unorm
racing-car-normal	Texture 1	2048 x 2048	RGBAB8Unorm
racing-car-roughness	Texture 2	2048 x 2048	R8Unorm
racing-car-metallic	Texture 3	2048 x 2048	R8Unorm
racing-car-ao	Texture 4	1024 x 1024	RGBAB8Unorm
Shadow Depth Texture	Texture 5	4096 x 4096	Depth32Float
sky	Texture 11	(Cube) 1024 x ...	RGBAB8Unorm
Texture 0x13775f9e0	Texture 12	(Cube) 64 x 64	RGBAB8Unorm
Fragment Bytes	Buffer 12 (Bytes)	48 bytes	
Buffer 0x137758750	Buffer 13	112 bytes	Offset: 0x0
Fragment Bytes	Buffer 14 (Bytes)	64 bytes	
fragment_skybox	Fragment Function		Library 0x60000000c000...
Attachments			

You can choose which of the six faces to show below the texture.



Instead of generating the irradiance texture each time, you can save the irradiance map to a file and load it from there. Included in the resources folder for this chapter is a project named **IrradianceGenerator**. You can use this app to generate your irradiance maps.

In your project, in the **Textures** group, there's a touched-up irradiance map named **irradiance.png** that matches and brightens the sky texture. It's time to switch to using this irradiance map for the diffuse texture instead of generating it.

► Open **Skybox.swift**, and in `init(textureName:)`, locate where you load `skyTexture` in the `do...catch`, and add the following code immediately after loading `skyTexture`:

```
diffuseTexture =  
try TextureController.loadCubeTexture(  
imageName: "irradiance.png")
```

► Remove the method `loadIrradianceMap()`, and also remove the call to `loadIrradianceMap()` at the end of `init(textureName)`.

- Build and run the app to see results using the brighter prebuilt irradiance map.

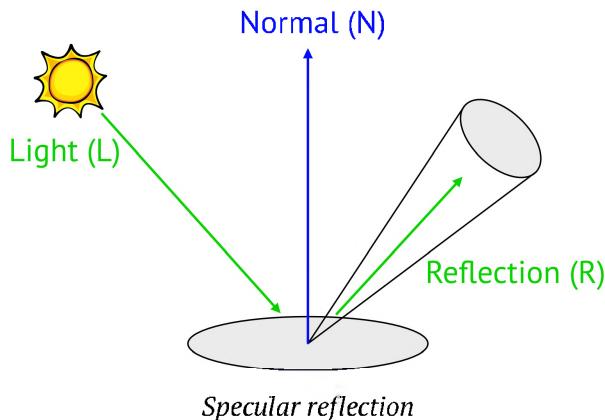


*Brighter irradiance*

## Specular Reflection

The irradiance map provides the diffuse and ambient reflection, but the specular reflection is a bit more difficult.

You may remember from Chapter 10, “Lighting Fundamentals”, that, whereas the diffuse reflection comes from all light directions, specular reflection depends upon the angle of view and the roughness of the material.



*Specular reflection*

In Chapter 11, “Maps & Materials”, you had a foretaste of physically based rendering using the Cook-Torrance microfacet specular shading model. This model is defined as:

$$\frac{f(I, v) = D(h)F(v, h)G(I, v, h)}{4(n \cdot I)(n \cdot v)}$$

Where you provide the light direction ( $\mathbf{l}$ ), view direction ( $\mathbf{v}$ ) and the half vector ( $\mathbf{h}$ ) between  $\mathbf{l}$  and  $\mathbf{v}$ . As described, the functions are:

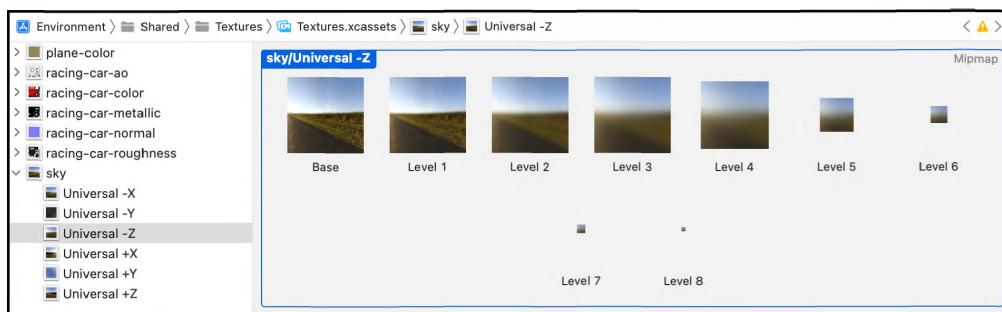
- **D**: Geometric micro-facet slope distribution
- **F**: Fresnel
- **G**: Geometric attenuation

Just as with the diffuse light, to get the accuracy of the incoming specular light, you need to take many samples, which is impractical in real-time rendering. Epic Games's approach in their paper, Real Shading in Unreal Engine 4 ([http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013\\_pbs\\_epic\\_notes\\_v2.pdf](http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf)), is to split up the shading model calculation. They pre-filter the sky cube texture with the geometry distribution for various roughness values. For each roughness level, the texture gets smaller and blurrier, and you can store these pre-filtered environment maps as different mipmap levels in the sky cube texture.

**Note:** In the resources for this chapter, there's a project named **Specular**, which uses the code from Epic Games's paper. This project takes in six images — one for each cube face — and will generate pre-filtered environment maps for as many levels as you specify in the code. The results are placed in a subdirectory of **Documents** named **specular**, which you should create before running the project. You can then add the created .png files to the mipmap levels of the sky cube texture in your asset catalog.

► In **Textures.xcassets**, look at the **sky** texture.

**sky** already contains the pre-filtered environment maps for each mip level.

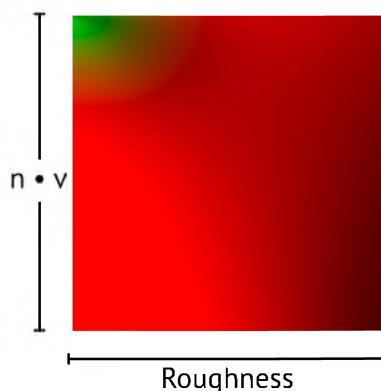


*Pre-filtered environment maps*

## BRDF Look-Up Table

To calculate the final color, you use a **Bidirectional Reflectance Distribution Function (BRDF)** that takes in the actual roughness of the model and the current viewing angle and returns the scale and bias for the Fresnel and geometric attenuation contributions.

You can encapsulate this BRDF in a look-up table (LUT) as a texture that behaves as a two-dimensional array. One axis is the roughness value of the object, and the other is the angle between the normal and the view direction. You input these two values as the UV coordinates and receive back a color. The red value contains the scale, and the green value contains the bias.



A BRDF LUT

The more photorealistic you want your scene to be, the higher the level of mathematics you'll need to know. In the resources folder for this chapter, in **references.markdown**, you'll find links with suggested reading that explain the Cook-Torrance microfacet specular shading model.

In the **Utility/BRDF** group, your project contains functions provided by Epic Games to create the BRDF look-up texture. You'll now implement the compute shader that builds the BRDF look-up texture.

► Open **Skybox.swift**, and add a property for the new texture:

```
var brdfLut: MTLTexture?
```

► At the end of `init(textureName:)`, call the method supplied in the starter project to build the texture:

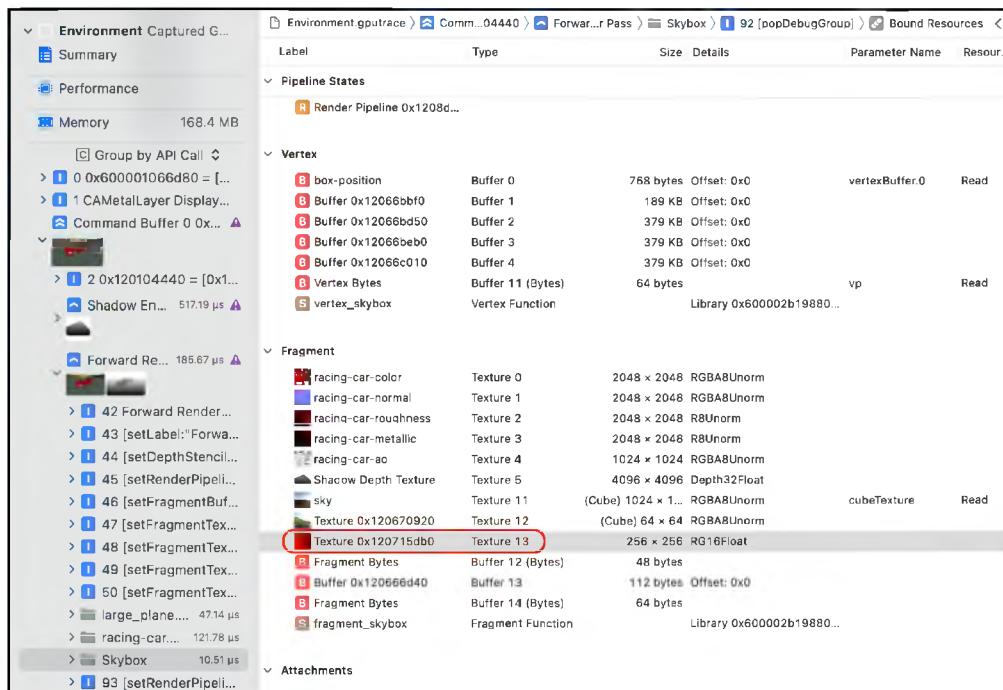
```
brdfLut = Renderer.buildBRDF()
```

`Renderer.buildBRDF()` uses a complex compute shader in `BRDF.metal` to create a new texture.

- Still in `Skybox.swift`, in `update(encoder:)`, add the following code to send the texture to the GPU:

```
encoder.setFragmentTexture(
    brdfLut,
    index: BRDFLutTexture.index)
```

- Build and run the app, and click the **Capture GPU frame** icon to verify the look-up texture created by the BRDF compute shader is available to the GPU.



*BRDF LUT is on the GPU*

Notice the texture format is `RG16Float`. As a float format, this pixel format has a greater accuracy than `RGBA8Unorm`.

All the necessary information is now on the GPU, so you need to receive the new BRDF look-up texture into the fragment shader and do the shader math.

- Open `IBL.metal`, and add the new parameter to `fragment_IBL`:

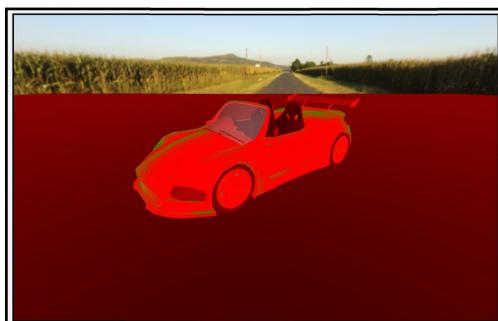
```
texture2d<float> brdfLut [[texture(BRDFLutTexture)]]
```

- At the end of `fragment_IBL`, replace `return color;` with this:

```
// 1
constexpr sampler s(filter::linear, mip_filter::linear);
float3 prefilteredColor
= skybox.sample(s,
    textureCoordinates,
    level(material.roughness * 10)).rgb;
// 2
float nDotV = saturate(dot(normal, -viewDirection));
float2 envBRDF
= brdfLut.sample(s, float2(material.roughness, nDotV)).rg;
return float4(envBRDF, 0, 1);
```

Going through the code:

1. Read the skybox texture along the reflected vector as you did earlier. Using the extra parameter `level(n)`, you can specify the mip level to read. You sample the appropriate mipmap for the roughness of the fragment.
  2. Calculate the angle between the view direction and the surface normal, and use this as one of the UV coordinates to read the BRDF look-up texture. The other coordinate is the roughness of the surface. You receive back the red and green values which you'll use to calculate the second part of the Cook Torrence equation.
- Build and run the app to see the result of the BRDF look-up.



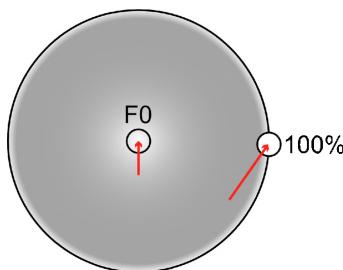
*The BRDF look up result*

At glancing angles on the car, the result is green.

## Fresnel Reflectance

When light hits an object straight on, some of the light is reflected. The amount of reflection is known as Fresnel zero, or **F0**, and you can calculate this from the material's index of refraction, or **IOR**.

When you view an object, at the viewing angle of 90°, the surface becomes nearly 100% reflective. For example, when you look across the water, it's reflective; but when you look straight down into the water, it's non-reflective.



Most dielectric (non-metal) materials have an F0 of about 4%, so most rendering engines use this amount as standard. For metals, F0 is the base color.

- Replace `return float4(envBRDF, 0, 1);` with:

```
float3 f0 = mix(0.04, material.baseColor.rgb,  
material.metallic);  
float3 specularIBL = f0 * envBRDF.r + envBRDF.g;
```

Here, you choose F0 as 0.04 for non-metals and the base color for metals. `metallic` **should** be a binary value of 0 or 1, but it's best practice to avoid conditional branching in shaders, so you use `mix()`. You then calculate the second part of the rendering equation using the values from the look-up table.

- Add the following code after the code you just added:

```
float3 specular = prefilteredColor * specularIBL;  
color += float4(specular, 1);  
return color;
```

`color` now includes the diffuse from the irradiance skybox texture, the material base color and the specular value.

- Build and run the app.



*Diffuse and specular*

Your car render is almost complete. Non-metals take the roughness value — the seats are matte, and the car paint is shiny. Metals reflect but take on the base color — the base color of the wheel hubs and the steel bar behind the seats is gray.

## Tweaking

Being able to tweak shaders gives you complete power over how your renders look. Because you're using low dynamic range lighting, the non-metal diffuse color looks a bit dark. You can tweak the color very easily.

- In `fragment_IBL`, after `float4 diffuse = skyboxDiffuse.sample(textureSampler, normal);`, add this:

```
diffuse = mix(pow(diffuse, 0.2), diffuse, material.metallic);
diffuse *= calculateShadow(in.shadowPosition, shadowTexture);
```

This code raises the power of the diffuse value but only for non-metals. You also reinstate the shadow.

- Build and run, and the car body is a lot brighter.



*Tweaking the shader*

The finishing touch will be to add a fake shadow effect using ambient occlusion. At the rear of the car, the exhausts look as if they are self-lit:

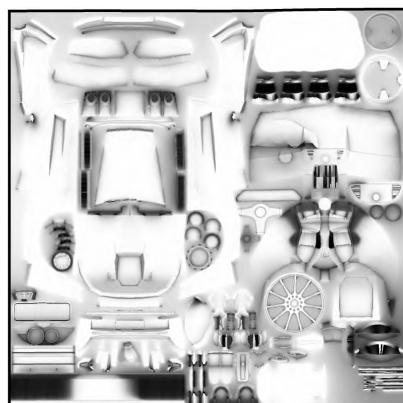


They should be shadowed because they are recessed. This is where ambient occlusion maps come in handy.

## Ambient Occlusion Maps

**Ambient occlusion** is a technique that approximates how much light should fall on a surface. If you look around you — even in a bright room — where surfaces are very close to each other, they're darker than exposed surfaces. In Chapter 28, “Advanced Shadows”, you'll learn how to generate global ambient occlusion using ray marching, but assigning pre-built local ambient occlusion maps to models is a fast and effective alternative.

Apps such as **Adobe Substance Painter** can examine the model for proximate surfaces and produce an ambient occlusion map. This is the AO map for the car, which is included in your project.



The white areas on the left, with a color value of 1.0, are UV mapped to the car paint. These are fully exposed areas. When you multiply the final render color by 1.0, it'll be unaffected. However, you can identify the wheel at the bottom right of the AO map, where the spokes are recessed. Those areas have a color value of perhaps 0.8, which darkens the final render color.

The ambient occlusion map is all set up in the starter project and ready for you to use.

- Open **IBL.metal**, and in **fragment\_IBL**, just before the final **return**, add this:

```
color *= material.ambientOcclusion;
```

- Build and run the app, and compare the exhaust pipes to the previous render.



All of the recessed areas are darker, which gives more natural lighting to the model.

## Challenge

On the first page of this chapter is a comparison of the car rendered in two different lighting situations. Your challenge is to create the red lighting scene.

Provided in the resources folder for this chapter, are six cube face png images converted from an HDRI downloaded from Poly Haven (<https://polyhaven.com>).

1. Create an irradiance map using the included **IrradianceGenerator** project, and import the generated map into the project.
2. Create specular mipmap levels using the included **Specular** project.
3. Create a new cube texture in the asset catalog.
4. Assign this new cube texture the appropriate generated mipmap images.
5. Change the sun light's position to  $[-1, 0.5, 2]$  to match the skybox.

Aside from the light position, there's no code to change — it's all imagery! You'll find the completed project in the challenge directory for this chapter.



## Key Points

- Using a cuboid skybox, you can surround your scene with a texture.
- Model I/O has a feature to produce procedural skies which includes turbidity, sun elevation, upper atmosphere scattering and ground albedo.
- Cube textures have six faces. Each of the faces can have mipmaps.
- Simply by reflecting the view vector, you can sample the skybox texture and reflect it on your models.
- Image-based lighting uses the sky texture for lighting. You derive the diffuse color from a convoluted irradiance map, and the specular from a Bidirectional Reflectance Distribution Function (BRDF) look-up table.

## Where to Go From Here?

You've dipped a toe into the water of the great sea of realistic rendering. If you want to explore more about this fascinating topic, **references.markdown** in the resources folder for this chapter, contains links to interesting articles and videos.

This chapter did not touch on spherical harmonics, which is an alternative method to using an irradiance texture map for diffuse reflection. Mathematically, you can approximate that irradiance map with 27 floats. Hopefully, the links in **references.markdown** will get you interested in this amazing technique.

Before you try to achieve the ultimate realistic render, one question you should ask yourself is whether your game will actually benefit from realism. One way to stand out from the crowd is to create your own rendering style. Games such as *Fortnite* aren't entirely realistic and have a style all of their own. Experiment with shaders to see what you can create.



# Chapter 22: Reflection & Refraction

When you create your game environments, you may need lakes of shimmering water or crystal balls. To look realistic, shiny glass objects require both **reflection** and **refraction**.

Reflection is one of the most common interactions between light and objects. Imagine looking into a mirror. Not only would you see your image being reflected, but you'd also see the reflection of any nearby objects.

Refraction is another common interaction between light and objects that you often see in nature. While it's true that most objects in nature are opaque — thus absorbing most of the light they get — the few objects that are translucent, or transparent, allow for the light to propagate through them.



*Reflection and refraction*

Later, in the final section of this book, you'll investigate ray tracing and global illumination, which allow advanced effects such as bounced reflections and realistic refraction. We're approaching a time where ray tracing algorithms may be viable in games, but for now, real-time rendering with rasterized reflection and refraction is the way to go.

An exemplary algorithm for creating realistic water was developed by Michael Horsch in 2005 (<https://bit.ly/3H2P1ix>). This realistic water algorithm is purely based on lighting and its optical properties, as opposed to having a water simulation based on physics.

## The Starter Project

- In Xcode, open the starter project for this chapter.

The starter project is similar to the project at the end of the previous chapter, with a few additions which include:

- **GameScene.swift** contains a new scene with new models and renders a new skybox texture. You can move around the scene using WASD keys, and look about using the mouse or trackpad. Scrolling the mouse wheel, or pinching on iOS, moves you up and down, so you can get better views of your lake.
- **WaterRenderPass.swift**, in the **Render Passes** group, contains a new render pass. It's similar to `ForwardRenderPass`, but refactors the command encoder setup into a new render method. `WaterRenderPass` is all set up and ready to render in `Renderer`.
- **Water.swift**, in the **Geometry** group, contains a new `Water` class, similar to `Model`. The class loads a primitive mesh plane and is set up to render the plane with its own pipeline state.
- **Pipelines.swift** has new pipeline state creation methods to render water and a terrain.

- Build and run the app.



*The starter app*

Visitors to this quaint cottage would love a recreational lake for swimming and fishing.

## Terrains

Many game scenes will have a ground terrain, or landscape, and this terrain may need its own shader. The starter project includes **Terrain.swift**, which contains **Terrain**, a subclass of **Model**. Changing shaders entails loading a new pipeline state, so **Terrain** creates its own pipeline state object along with a texture for use later.

**Terrain.metal** holds the fragment function to render the terrain. After you've added some water, you'll change the terrain texture to blend with an underwater texture.

Instead of including the ground when rendering the scene models, **ForwardRenderPass** renders the terrain separately, as you did for the skybox.

## Rendering Rippling Water

Here's the plan on how you'll proceed through the chapter:

1. Render a large horizontal quad that will be the surface of the water.
2. Render the scene to a reflection texture.
3. Use a clipping plane to limit what geometry you render.

4. Distort the reflection using a normal map to create ripples on the surface.
5. Render the scene to a refraction texture.
6. Apply the Fresnel effect so that the dominance of each texture will change depending on the viewing angle.
7. Add smoothness to the water depth visibility using a depth texture.

Ready? It's going to be a wild ride but stick around until the end, because you won't want to miss this.

## 1. Creating the Water Surface

- In the **Geometry** group, open **Water.swift**, and examine the code.

Similar to **Model**, **Water** initializes a mesh and is **Transformable**, so you can position, rotate and scale the mesh. The mesh is a plane primitive. **Water** also has a render method where you'll add textures and render the mesh plane.

- Open **Pipelines.swift**, and locate `createWaterPSO(vertexDescriptor:)`.

The water pipeline will need new shader functions. You'll name the vertex function `vertex_water` and the fragment function `fragment_water`.

## Creating the Water Shaders

- In the **Shaders** group, create a new Metal file named **Water.metal**, and add this:

```
#import "Common.h"

struct VertexIn {
    float4 position [[attribute(Position)]];
    float2 uv [[attribute(UV)]];
};

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

vertex VertexOut vertex_water(
    const VertexIn in [[stage_in]],
    constant Uniforms &uniforms [[buffer(UniformsBuffer)]])
{
```



```
float4x4 mvp = uniforms.projectionMatrix * uniforms.viewMatrix
    * uniforms.modelMatrix;
VertexOut out {
    .position = mvp * in.position,
    .uv = in.uv,
    .worldPosition = uniforms.modelMatrix * in.position
};
return out;
}

fragment float4 fragment_water(
    VertexOut in [[stage_in]],
    constant Params &params [[buffer(ParamsBuffer)]])
{
    return float4(0.0, 0.3, 0.5, 1.0);
}
```

This code provides a minimal configuration for rendering the water surface quad. The vertex function moves the mesh into position, and the fragment function shades the mesh a bluish color.

## Adding the Water to Your Scene

- Open **GameScene.swift**, and add a new property:

```
var water: Water?
```

- Add the following code to `init()`:

```
water = Water()
water?.position = [0, -1, 0]
```

With this code, you initialize and position the water plane.

- Open **ForwardRenderPass.swift**, and in `draw(commandBuffer:scene:uniforms:params:)`, locate where you render the skybox.

- Add the following code immediately before rendering the skybox:

```
scene.water?.render(
    encoder: renderEncoder,
    uniforms: uniforms,
    params: params)
```



- Build and run the app.



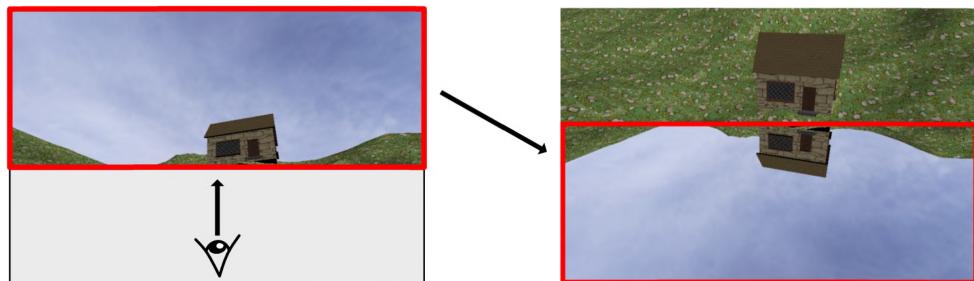
*The initial water plane*

The cottage is now a waterfront vacation home.

## 2. Rendering the Reflection

The water plane should reflect its surroundings. In Chapter 21, “Image-Based Lighting”, you reflected the skybox onto objects, but this time you’re also going to reflect the house and terrain on the water.

You’re going to render the scene to a texture from a point underneath the water pointing upwards. You’ll then take this texture and render it flipped on the water surface.



You’ll do all this in `WaterRenderPass`.

- Open `WaterRenderPass.swift`.

WaterRenderPass initializes the pipeline and depth stencil states. Each frame, Renderer calls

`waterRenderPass.draw(commandBuffer:scene:uniforms:params:)`, which renders the entire scene (minus the water). Currently the render pass doesn't render anything, because `descriptor` is `nil` and the app exits the method.

► In `WaterRenderPass`, add this to `init()`:

```
descriptor = MTLRenderPassDescriptor()
```

You initialize a new render pass descriptor.

► Add some new texture properties to `WaterRenderPass`:

```
var reflectionTexture: MTLTexture?  
var refractionTexture: MTLTexture?  
var depthTexture: MTLTexture?
```

You'll keep both a reflection and a refraction texture, as you'll render these textures from different camera positions. Although you're setting up the refraction texture, you won't be using it until later in the chapter.

► In `resize(view:size:)`, add this:

```
let size = CGSize(  
    width: size.width / 2, height: size.height / 2)  
reflectionTexture = Self.makeTexture(  
    size: size,  
    pixelFormat: view.colorPixelFormat,  
    label: "Reflection Texture")  
refractionTexture = Self.makeTexture(  
    size: size,  
    pixelFormat: view.colorPixelFormat,  
    label: "Refraction Texture")  
depthTexture = Self.makeTexture(  
    size: size,  
    pixelFormat: .depth32Float,  
    label: "Reflection Depth Texture")
```

Any time you can save on memory, you should. For reflection and refraction you don't really need sharp images, so you create the textures at half the usual size.

- Add the following code to the top of

```
draw(commandBuffer:scene:uniforms:params:):
```

```
let attachment = descriptor?.colorAttachments[0]
attachment?.texture = reflectionTexture
attachment?.storeAction = .store
let depthAttachment = descriptor?.depthAttachment
depthAttachment?.texture = depthTexture
depthAttachment?.storeAction = .store
```

You'll render color to the `reflectionTexture`, depth to `depthTexture` and ensure that the GPU stores the textures for later use. Because you'll render the skybox — which will cover the whole render target — you don't care what the load actions are.

The water plane will use these render textures, so you'll store them to `Water`.

- In the `Geometry` group, open `Water.swift`, and add the new properties to `Water`:

```
weak var reflectionTexture: MTLTexture?
weak var refractionTexture: MTLTexture?
weak var refractionDepthTexture: MTLTexture?
```

- In `render(encoder:uniforms:params:)`, add the following code before the draw call:

```
encoder.setFragmentTexture(
    reflectionTexture,
    index: 0)
encoder.setFragmentTexture(
    refractionTexture,
    index: 1)
```

Soon, you'll change the `fragment_water` shader to use these textures.

- Open `WaterRenderPass.swift`, and add the following code to the top of

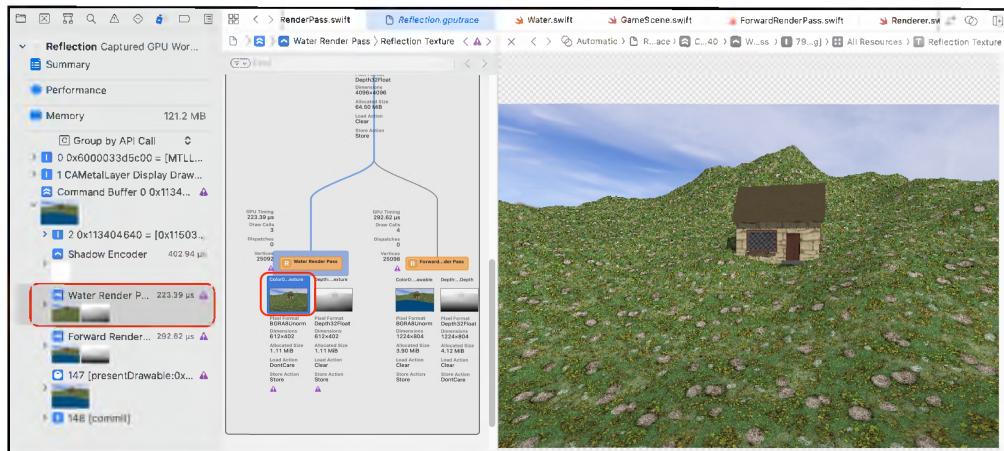
```
draw(commandBuffer:scene:uniforms:params:):
```

```
guard let water = scene.water else { return }
water.reflectionTexture = reflectionTexture
water.refractionTexture = refractionTexture
water.refractionDepthTexture = depthTexture
```

Here, you pass on the render target textures to `Water` for texturing the water plane.

- Build and run the app to see progress so far.

You won't see any obvious changes, but capture the GPU workload and check out the water render pass:



`WaterRenderPass` renders an exact duplicate of the scene, not including the water plane, to the reflection render target texture.

- In the **Shaders** group, open `Water.metal`, and add new parameters to `fragment_water`:

```
texture2d<float> reflectionTexture [[texture(0)]],  
texture2d<float> refractionTexture [[texture(1)]]
```

You're concentrating on reflection for now, but you'll use the refraction texture later.

- In `fragment_water`, replace the return line with this:

```
// 1  
constexpr sampler s(filter::linear, address::repeat);  
// 2  
float width = float(reflectionTexture.get_width() * 2.0);  
float height = float(reflectionTexture.get_height() * 2.0);  
float x = in.position.x / width;  
float y = in.position.y / height;  
float2 reflectionCoords = float2(x, 1 - y);  
// 3  
float4 color = reflectionTexture.sample(s, reflectionCoords);  
color = mix(color, float4(0.0, 0.3, 0.5, 1.0), 0.3);  
return color;
```

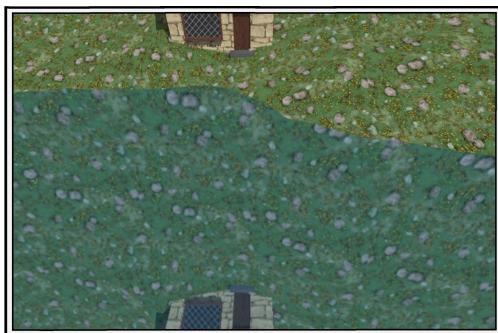
Going through the code:

1. Create a new sampler with linear filtering and repeat addressing mode, so that the texture tiles at the edge if necessary.
  2. Determine the reflection coordinates which will use an inverted y value because the reflected image is a mirror of the scene above the water surface. Notice you multiply by 2.0. You do this because the texture is only half-size.
  3. Sample the color from the reflection texture, and mix it a little with the previous bluish color the water plane had before.
- Build and run the app, and you'll see the house, terrain and sky reflected on the water surface.



*Initial reflection*

It looks nice now, but rotate the camera with your mouse or trackpad, and you'll see the reflection is incorrect. You need to render the reflection target from a different camera position.



*Incorrect reflection*

- Open **WaterRenderPass.swift**, and in `draw(commandBuffer:scene:uniforms:params:)`, add the following code before calling `render(renderEncoder:scene:uniforms:params:)`.

```
var reflectionCamera = scene.camera
reflectionCamera.rotation.x *= -1
let position = (scene.camera.position.y - water.position.y) * 2
reflectionCamera.position.y -= position

var uniforms = uniforms
uniforms.viewMatrix = reflectionCamera.viewMatrix
```

Here, you use a separate camera specially for reflection, and position it below the surface of the water to capture what's above the surface.

- Build and run the app to see the updated result:



*Reflected camera position*

That didn't work out too well.

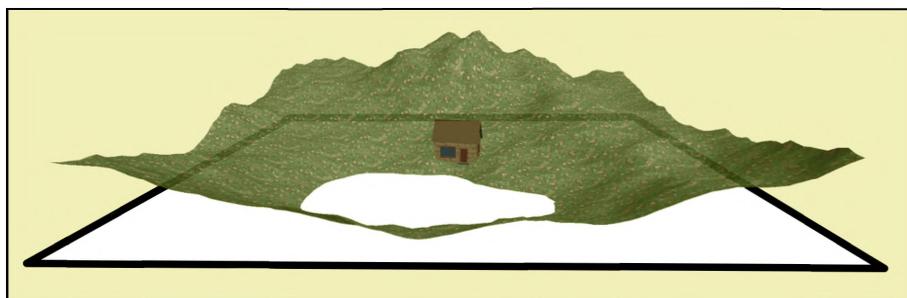
As the main camera moves *up* the y-axis, the reflection camera moves *down* the y-axis to below the terrain surface which blocks the view to the sky. You could temporarily solve this by culling the terrain's back faces when you render, but this may introduce other rendering artifacts. A better way of dealing with this issue is to clip the geometry you don't want to render.

## 3. Creating Clipping Planes

A clipping plane, as its name suggests, clips the scene using a plane. It's hardware accelerated, meaning that if geometry is not within the clip range, the GPU immediately discards the vertex and doesn't put it through the entire pipeline. You may get a significant performance boost as some of the geometry will not need to get processed by the fragment shaders anymore.

For the reflection texture, you only need to render the scene as if from under the water, flip it, and add it to the final render.

Placing the clipping plane at the level of the water, ensures that only the scene geometry above the water is rendered to the reflection texture.



*The clipping plane*

- Still in **WaterRenderPass.swift**, in `draw(commandBuffer:scene:uniforms:params:)`, after the previous code, add this:

```
var clipPlane = float4(0, 1, 0, -water.position.y)
uniforms.clipPlane = clipPlane
```

With this code, you create `clipPlane` as a `var` because you'll adjust it shortly for refraction.

The clipping plane `xyz` is a direction vector that denotes the clipping direction. The last component is the level of the water.

- In the **Shaders** group, open **Common.h**, and add a new member to **Uniforms**:

```
vector_float4 clipPlane;
```

- Open **Vertex.h**, and add a new member to **VertexOut**:

```
float clip_distance [[clip_distance]] [1];
```

Notice the Metal Shading Language attribute, `clip_distance`, which is one of the built-in attributes exclusively used by vertex shaders. The `clip_distance` attribute is an array of distances, and the [1] argument represents its size — a 1 in this case because you only need one member in the array.

You may have noticed that `FragmentIn` is a duplicate of `VertexOut`. `[[clip_distance]]` is a vertex-only attribute, and fragment functions now won't compile if they use `VertexOut`.

**Note:** You can read more about matching vertex and fragment attributes in the Metal Shading Language specification (<https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf>), in Section 5.7.1 “Vertex-Fragment Signature Matching.”

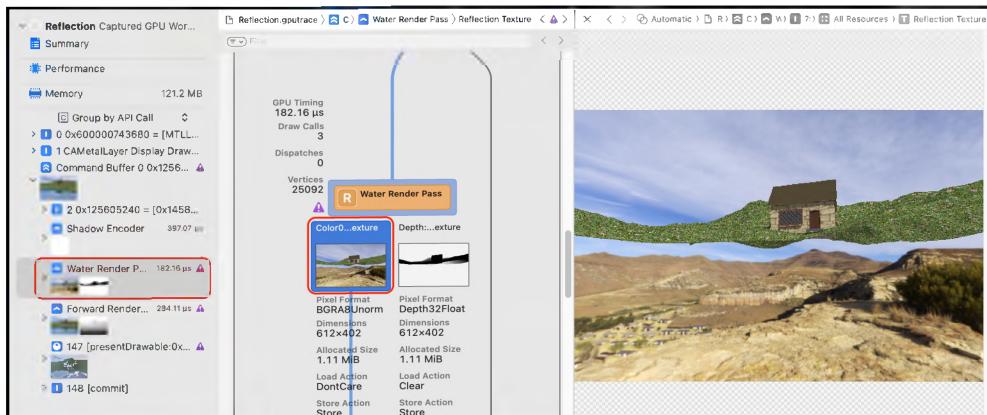
► Open `Shaders.metal`, and add this to `vertex_main` before `return`:

```
out.clip_distance[0] =
    dot(uniforms.modelMatrix * in.position, uniforms.clipPlane);
```

Any negative result in `vertex_out.clip_distance[0]` will result in the vertex being clipped.

You're now clipping any geometry processed by `vertex_main`. You could also clip the skybox in the same way, but when you later add ripples, they may go below the clipping plane, leaving nothing to reflect.

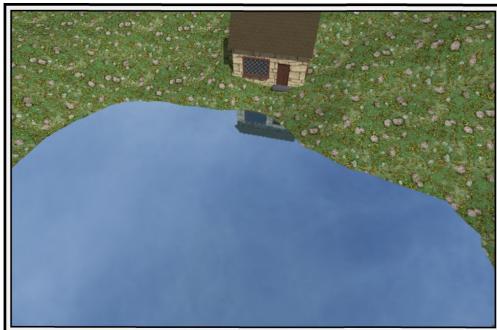
► Build and run the app, capture the GPU workload and select the Water Render Pass.



Rendering above the clipping plane

All rendered scene model geometry is clipped, but the skybox still renders.

- Continue running the app, move the camera up and rotate it to look downwards.



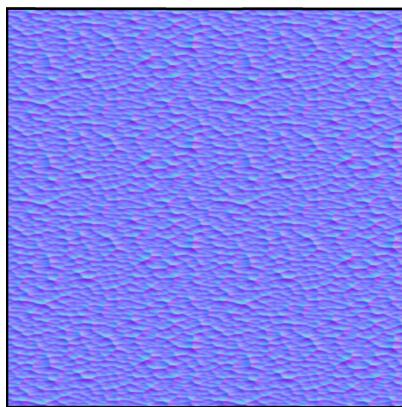
*Reflecting the sky correctly*

The sky now reflects correctly and your water reflection appears smooth and calm. As you move the camera up and down, the reflection is now consistent.

Still water, no matter how calming, isn't realistic. Time to give that water some ripples.

## 4. Rippling Normal Maps

- Open **Textures.xcassets**, and select **normal-water**. This is a normal map that you'll use for the water ripples.



*The water ripple normal map*

You'll tile this map across the water and move it, perturbing the water normals, which will make the water appear to ripple.

- In the **Geometry** group, open **Water.swift**, and add these new properties to **Water**:

```
var waterMovementTexture: MTLTexture?  
var timer: Float = 0
```

You add the texture, and a timer so that you can animate the normals.

- Add the following code to the end of **init()**:

```
waterMovementTexture =  
    try? TextureController.loadTexture(filename: "normal-water")
```

- In **render(encoder:uniforms:params:)**, add the following code where you set the other fragment textures:

```
encoder.setFragmentTexture(  
    waterMovementTexture,  
    index: 2)  
var timer = timer  
encoder.setFragmentBytes(  
    &timer,  
    length: MemoryLayout<Float>.size,  
    index: 3)
```

Here, you send the texture and the timer to the fragment shader.

- Add this new method to **Water**:

```
func update(deltaTime: Float) {  
    let sensitivity: Float = 0.005  
    timer += deltaTime * sensitivity  
}
```

**deltaTime** will be too fast, so you include a sensitivity modifier.

- Open **GameScene.swift**, and add this to the top of **update(deltaTime:)**:

```
water?.update(deltaTime: deltaTime)
```

**GameScene** will update the water timer every frame.

You've now set up the texture and the timer on the CPU side.

- Open **Water.metal**, and add two new parameters for the texture and timer to **fragment\_water**:

```
texture2d<float> normalTexture [[texture(2)]],
```



```
constant float& timer [[buffer(3)]]
```

► Add the following code before you define color:

```
// 1
float2 uv = in.uv * 2.0;
// 2
float waveStrength = 0.1;
float2 rippleX = float2(uv.x + timer, uv.y);
float2 rippleY = float2(-uv.x, uv.y) + timer;
float2 ripple =
    ((normalTexture.sample(s, rippleX).rg * 2.0 - 1.0) +
     (normalTexture.sample(s, rippleY).rg * 2.0 - 1.0))
    * waveStrength;
reflectionCoords += ripple;
// 3
reflectionCoords = clamp(reflectionCoords, 0.001, 0.999);
```

Going through the code:

1. Get the texture coordinates and multiply them by a tiling value. For 2 you get huge, ample ripples, while for 16 you get quite small ripples. Pick a value that suits your needs.
2. Calculate ripples by distorting the texture coordinates with the timer value. Only grab the R and G values from the sampled texture because they are the U and V coordinates that determine the horizontal plane where the ripples will be. The B value is not important here. `waveStrength` is an attenuator value, that gives you weaker or stronger waves.
3. Clamp the reflection coordinates to eliminate anomalies around the margins of the screen.

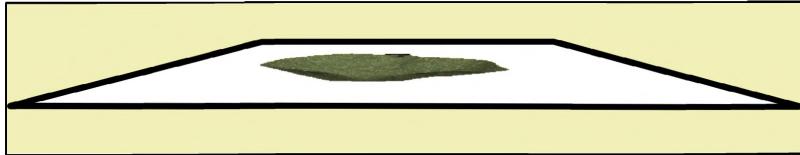
► Build and run the app, and you'll see gorgeous ripples on the water surface.



*Calm water ripples*

## 5. Adding Refraction

Implementing refraction is very similar to reflection, except that you only need to preserve the part of the scene below the clipping plane.



*Rendering below the clipping plane*

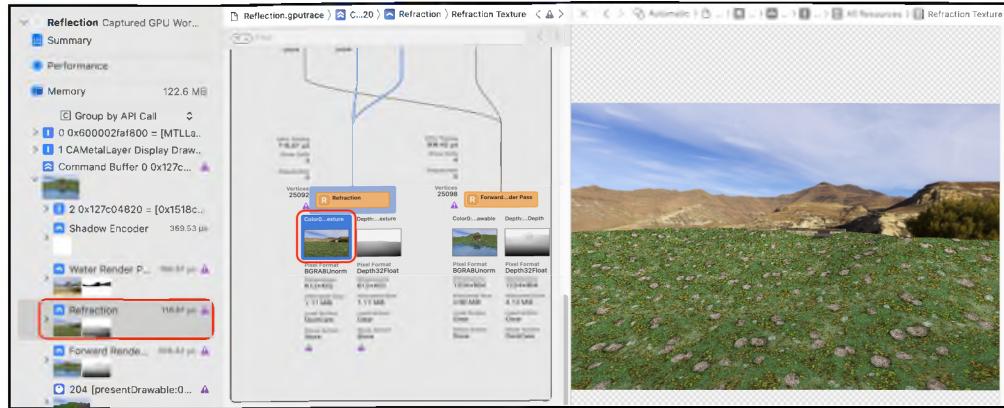
► Open **WaterRenderPass.swift**, and add this to the end of `draw(commandBuffer:scene:uniforms:params:)`:

```
// 1
descriptor.colorAttachments[0].texture = refractionTexture
// 2
guard let refractEncoder =
commandBuffer.makeRenderCommandEncoder(
    descriptor: descriptor) else {
    return
}
refractEncoder.label = "Refraction"
// 3
uniforms.viewMatrix = scene.camera.viewMatrix
clipPlane = float4(0, -1, 0, -water.position.y)
uniforms.clipPlane = clipPlane
// 4
render(
    renderEncoder: refractEncoder,
    scene: scene,
    uniforms: uniforms,
    params: params)
refractEncoder.endEncoding()
```

Going through the code:

1. Set the refraction texture as the render target.
2. Create a new render command encoder.
3. Set the y value of the clip plane to -1 since the camera is now in its original position and pointing down toward the water.
4. Render all the elements of the scene again.

- Build and run the app, capture the GPU workload and check out the refraction texture.



*The refraction texture*

This time, the GPU has only rendered the scene geometry below the clipping plane.

You're already sending the refraction texture to the GPU, so you can work on the calculations straightaway.

- Open **Water.metal**, and, in `fragment_water`, add this below where you define `reflectionCoords`:

```
float2 refractionCoords = float2(x, y);
```

For refraction you don't have to flip the y coordinate.

- Similarly, add this below `reflectionCoords += ripple;`:

```
refractionCoords += ripple;
```

- And once more, add this after the reflection line preventing edge anomalies:

```
refractionCoords = clamp(refractionCoords, 0.001, 0.999);
```

- Finally, replace:

```
float4 color = reflectionTexture.sample(s, reflectionCoords);
```

- With:

```
float4 color = refractionTexture.sample(s, refractionCoords);
```

You temporarily show only the refraction texture. You'll return and include reflection shortly.

- Build and run the app.



*Refraction*

The reflection on the water surface is gone, and instead, you have refraction through the water.

There's one more visual enhancement you can make to your water to make it more realistic: adding rocks and grime. Fortunately, the project already has a texture that can simulate this.

- Open **Terrain.metal**, and in **fragment\_terrain**, uncomment the section under `// uncomment this for pebbles.`

- Build and run the app, and you'll now see a pebbled texture underwater.



*Pebbles*

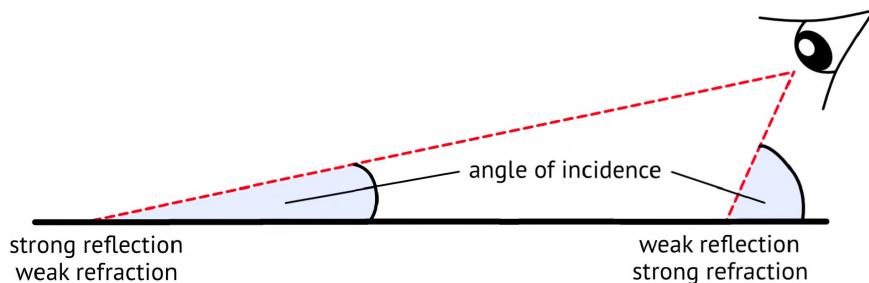
The holy grail of realistic water, however, is having a Fresnel effect that harmoniously combines reflection and refraction based on the viewing angle.

## 6. The Fresnel Effect

The **Fresnel effect** is a concept you've met with in previous chapters. As you may remember, the viewing angle plays a significant role in the amount of reflection you can see. What's new in this chapter is that the viewing angle also affects refraction but in inverse proportion:

- The steeper the viewing angle is, the weaker the reflection and the stronger the refraction.
- The shallower the viewing angle is, the stronger the reflection and the weaker the refraction.

The Fresnel effect in action:

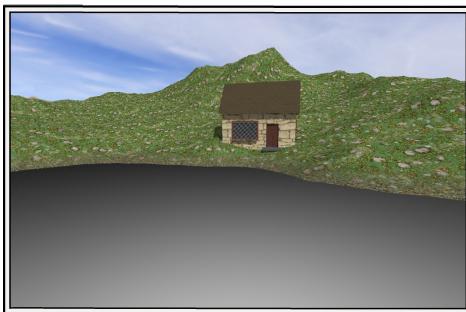


► Open `Water.metal`, and in `fragment_water`, before you define `color`, add this:

```
float3 viewVector =  
    normalize(params.cameraPosition - in.worldPosition.xyz);  
float mixRatio = dot(viewVector, float3(0, 1, 0));  
return mixRatio;
```

Here, you work out the view vector between the camera and the water fragment. The mix ratio will be the blend between reflection and refraction.

- Build and run the app.



*The mix ratio between refraction and reflection*

As you move about the scene, the greater the angle between the camera and the water, the whiter the water becomes. A view across the water, down close to the water, returns black.

Instead of rendering black and white, you'll mix between the refraction and reflection textures. Where the mix ratio is black, you'll render the reflection texture, and where it's white, refraction. A ratio of `0.5` would mean that reflection and refraction are mixed equally.

- Replace:

```
return mixRatio;  
float4 color = refractionTexture.sample(s, refractionCoords);
```

- With:

```
float4 color =  
    mix(reflectionTexture.sample(s, reflectionCoords),  
        refractionTexture.sample(s, refractionCoords),  
        mixRatio);
```

- Build and run the app.



Move the camera around and notice how reflection predominates for a small viewing angle while refraction predominates when the viewing angle is getting closer to 90 degrees (perpendicular to the water surface).

## 7. Adding Smoothness Using a Depth Texture

Light propagation varies for different transparent media, but for water, the colors with longer wavelengths (closer to infrared) quickly fade away as the light ray goes deeper. The bluish colors (closer to ultraviolet) tend to be visible at greater depths because they have shorter wavelengths.

At very shallow depths, however, most light should still be visible. You'll make the water look smoother as depth gets smaller. You can improve the way the water surface blends with the terrain by using a depth map.

► Open **Water.swift**, and add the following code to `render(encoder:uniforms:params:)` when you set the other fragment textures:

```
encoder.setFragmentTexture(  
    refractionDepthTexture,  
    index: 3)
```

As well as sending the refraction texture from the refraction render pass, you're now sending the depth texture too.

► Open **Pipelines.swift**, and add this to `createWaterPSO(vertexDescriptor:)` before the return:

```
let attachment = pipelineDescriptor.colorAttachments[0]  
attachment?.isBlendingEnabled = true  
attachment?.rgbBlendOperation = .add  
attachment?.sourceRGBBlendFactor = .sourceAlpha  
attachment?.destinationRGBBlendFactor = .oneMinusSourceAlpha
```

Here, you configure the blending options on the color attachment just as you did back in Chapter 20, “Fragment Post-Processing.”

► Open **Water.metal**, and add the depth texture parameter to `fragment_water`:

```
depth2d<float> depthMap [[texture(3)]]
```

- Add the following code before you set `rippleX` and `rippleY`:

```
float far = 100;      // the camera's far plane
float near = 0.1;     // the camera's near plane
float proj33 = far / (far - near);
float proj43 = proj33 * -near;
float depth = depthMap.sample(s, refractionCoords);
float floorDistance = proj43 / (depth - proj33);
depth = in.position.z;
float waterDistance = proj43 / (depth - proj33);
depth = floorDistance - waterDistance;
```

You convert the non-linear depth to a linear value.

**Note:** Why and how you convert from non-linear to linear, is mathematically complex. gamedev.net forums (<https://bit.ly/3r086fK>) has an explanation of converting a non-linear depth buffer value to a linear depth value.

Finally, add this before `return`:

```
color.a = clamp(depth * 0.75, 0.0, 1.0);
```

Here, you change the alpha channel so that blending goes into effect.

- Build and run the app, and you'll now see a smoother blending of the shore with the terrain.



*Blending at the water's edge*

## Key Points

- Reflection and refraction are important for realistic water and glass.
- Rasterizing reflections and refraction will not produce as good a result as ray tracing. But when speed is a concern, then ray tracing is not often viable.
- Use separate render passes to render textures. For reflection, move the camera in the inverse direction from the plane to be reflected and flip the result.
- You already know about near and far clipping planes, but you can also add your own custom clipping planes. A negative clip distance from in the vertex function will result in the GPU discarding the vertex.
- You can animate normal maps to provide water turbulence.
- The Fresnel effect depends upon viewing angle and affects reflection and refraction in inverse proportion.

## Where to Go From Here?

You've certainly made a splash with this chapter! If you want to explore more about water rendering, **references.markdown** file in the resources folder for this chapter contains links to interesting articles and videos.



# 23

## Chapter 23: Animation

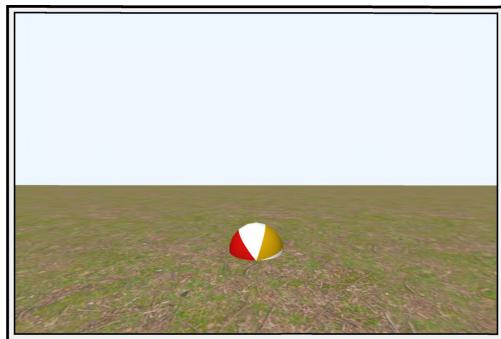
Rendering models that don't move is a wonderful achievement, but *animating* models takes things to an entirely new level.

To *animate* means to *bring to life*. So what better way to play with animation than to render characters with personality and body movement. In this chapter, you'll find out how to do basic animation using keyframes.



## The Starter Project

- In Xcode, open the starter project for this chapter, and build and run the app.

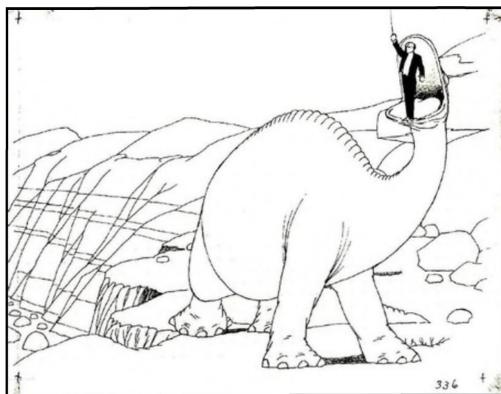


The scene contains a ground plane and a ball. Because there's no skybox, the renderer will use the forward renderer with PBR shading.

In the **Animation** group, **BallAnimations.swift** contains a few pre-built animations. At the moment, the ball animation is a bit unnatural looking — it's just sitting there embedded into the ground. To liven things up, you'll start off by making it roll around the scene.

## Animation

Animators like Winsor McCay and Walt Disney brought life to still images by filming a series of hand-drawn pictures one frame at a time.



*Winsor McCay: Gertie the Dinosaur*

This frame-by-frame animation was — and still is — very time consuming. With the rise of computer animation, artists can now create 3D models and record their positions at specific points in time. From there, the computer could interpolate, or **tween**, the values between those positions, making the animation process a lot less time consuming. But there is another option: procedural animation.

## Procedural Animation

Procedural animation uses mathematics to calculate transformations over time. In this chapter, you'll first animate the ball using the sine function, just as you did earlier in Chapter 7, “The Fragment Function”, when you animated a quad with trigonometric functions.

The started project contains a scene with a ball. To begin, you'll create a structure that controls the ball's animation.

- In the **Game** group, add a new Swift file named **Beachball.swift**, and add this:

```
struct Beachball {  
    var ball: Model  
    var currentTime: Float = 0  
  
    init(model: Model) {  
        self.ball = model  
        ball.position.y = 1  
    }  
  
    mutating func update(deltaTime: Float) {  
        currentTime += deltaTime  
    }  
}
```

Here, you initialize **Beachball** with the model reference, and create a method that **GameScene** will call every frame. (You'll use the timer to animate your model over time.)

- Open **GameScene.swift**, and add a new property:

```
lazy var beachball = Beachball(model: ball)
```

- Then, add the following code to the top of **update(deltaTime:)**:

```
beachball.update(deltaTime: deltaTime)
```

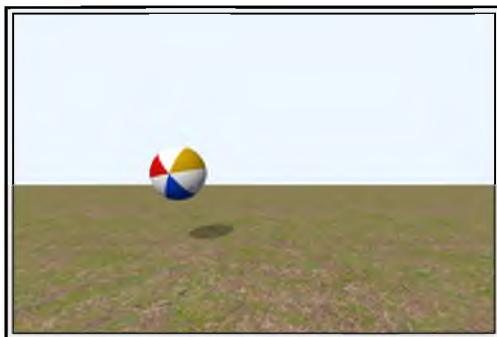
All of the ball's movement and animation will now take place in **Beachball**.

- Open **Beachball.swift**, and add the following code to the end of `update(deltaTime:)`:

```
ball.position.x = sin(currentTime)
```

This code updates the ball's `x` position every frame by the sine of the accumulated current time.

- Build and run the app.



*Side to side sine animation*

The ball now moves from side-to-side.

Sine is useful for procedural animation. By changing the amplitude, period and frequency, you can create waves of motion — although, for a ball, that's not very realistic. However, with some physics, you can add a little bounce to its movement.

## Animation Using Physics

Instead of creating animation by hand using an animation app, you can use physics-based animation, which means that your models can simulate the real world. In this next exercise, you're going to simulate only gravity and a collision. However, a full physics engine can simulate all sorts of effects, such as fluid dynamics, cloth and soft body (rag doll) dynamics.

- Create a new property in `Beachball` to track the ball's velocity:

```
var ballVelocity: Float = 0
```

- Remove the following code from `update(deltaTime:)`:

```
ball.position.x = sin(currentTime)
```

- In `update(deltaTime:)`, set some constants for the individual physics that you'll need for the simulation:

```
let gravity: Float = 9.8 // meter / sec2
let mass: Float = 0.05
let acceleration = gravity / mass
let airFriction: Float = 0.2
let bounciness: Float = 0.9
let timeStep: Float = 1 / 600
```

`gravity` represents the acceleration of an object falling to Earth. If you're simulating gravity elsewhere in the universe, for example, Mars, this value would be different. Newton's Second Law of Motion is  $F = ma$  or `force = mass * acceleration`. Rearranging the equation gives `acceleration = force (gravity) / mass`. The other constants describe the surroundings and properties of the ball. If this were a bowling ball, it would have a higher mass and less bounce.

- Add this at the end of `update(deltaTime:)`:

```
ballVelocity += (acceleration * timeStep) / airFriction
ball.position.y -= ballVelocity * timeStep

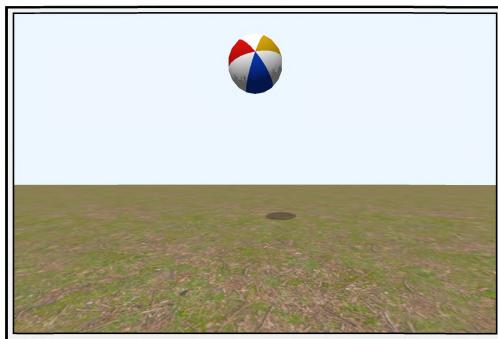
// collision with ground
if ball.position.y <= 0.35 {
    ball.position.y = 0.35
    ballVelocity = ballVelocity * -1 * bounciness
}
```

Here, you calculate the position of the ball based on its current velocity. The ball's origin is at its center, and it's approximately 0.7 units in diameter. So when the ball's center is 0.35 units above the ground, that's when you reverse the velocity and travel upward.

- In `init(model:)`, change the ball's initial position to be higher up in the air:

```
ball.position = [0, 3, 0]
```

- Build and run the app, and watch the ball bounce around.

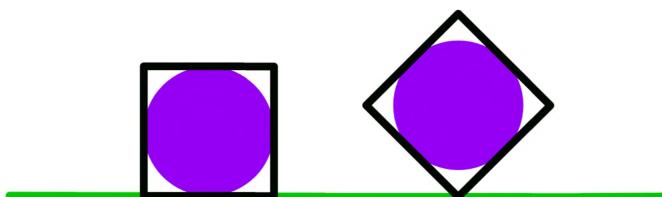


*A bouncing ball*

This is a simple physics animation, but it demonstrates what you can do with very little code.

## Axis-Aligned Bounding Box

You hard-coded the ball's radius so that it collides with the ground, but collision systems generally require some kind of bounding box to test whether an object collides with another object.



*Axis aligned bounding box*

A ball would benefit from a spherical bounding volume. Because your ball is simply using the y-axis, you can determine the ball's height using an **axis-aligned bounding box** that Model I/O calculates.

- In the **Geometry** group, open **Model.swift**. Then, add a bounding box property and a computed size property to **Model**:

```
var boundingBox = MDLAxisAlignedBoundingBox()  
var size: float3 {  
    return boundingBox.maxBounds - boundingBox.minBounds  
}
```

- Next, add the following code at the end of `init(name:)`:

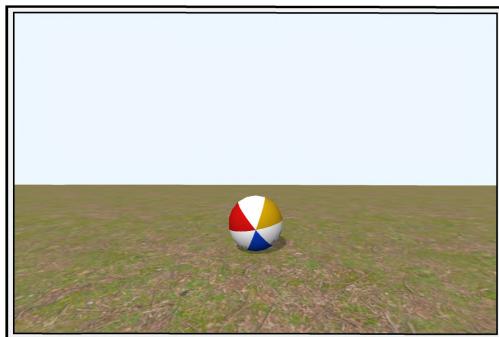
```
boundingBox = asset.boundingBox
```

This code extracts the bounding box information from Model I/O.

- Open **Beachball.swift**, and in `update(deltaTime:)`, update the collision code with the correct size calculated from the bounding box value:

```
// collision with ground
if ball.position.y <= ball.size.y / 2 {
    ball.position.y = ball.size.y / 2
    ballVelocity = ballVelocity * -1 * bounciness
}
```

- Build and run the app, and your ball will collide with the ground, precisely at the edge of the ball.



*Collision with the ground*

## Keyframes

Let's animate the ball getting tossed around by adding some input information about its position over time. For this input, you'll need an array of positions so that you can extract the correct position for the specified time.

In the **Animation** group, in **BallAnimations.swift**, there's an array already set up: `ballPositionXArray`. This array contains 60 values ranging from -1 to 1, then back to -1. By calculating the current frame, you can grab the correct x position from the array.

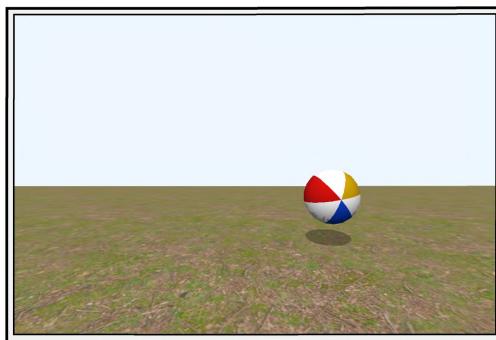
- Open **Beachball.swift**, and replace `update(deltaTime:)` with:

```
mutating func update(deltaTime: Float) {
    currentTime += deltaTime
    ball.position.y = 1

    let fps: Float = 60
    let currentFrame =
        Int(currentTime * fps) % (ballPositionXArray.count)
    ball.position.x = ballPositionXArray[currentFrame]
}
```

Here, you calculate the current frame (based on 60 fps) and extract the correct value from the array.

- Build and run the app.



*Frame by frame animation*

Watch as the ball moves around in a mechanical, backward and forward motion over 60 frames. This is almost the same result as the sine animation, but instead, it's animated using an array of values that you can control and re-use.

Here, you're setting the position 60 times per second — but even at that speed, the animation appears jerky. If you were to lower the speed to 30 fps, the animation would look awful.

## Interpolation

It's a lot of work inputting a value for each frame. If you're just moving an object in a straight line from point A to B, you can **interpolate** the value. Interpolation is where you calculate a value given a range of values and a current location within the range. When animating, the current location is the current time as a percentage of the animation duration.

- To work out the time percentage, use the following formula:

$$\text{time} = \frac{(\text{currentTime} - \text{min})}{(\text{max} - \text{min})}$$

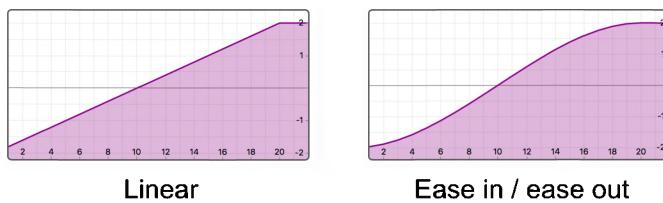
This formula results in a time value between 0 and 1.

For example, with a start value of 5, an end value of 10 and a duration of 2 seconds, after 1 second passes, the interpolated value is 7.5.

$$\text{time} = \frac{(1.0 - 0)}{(2.0 - 0)}$$

$$\text{value} = (10 - 5) * \text{time} + 5$$

That's a linear interpolation, but you can use other formulas for interpolating.



In the above image, linear interpolation is on the left. The x-axis is time, and the y-axis is value. You sample the value at the appropriate time. You can improve the ball's animation using ease in / ease out interpolation (shown on the right) to make the animation less mechanical. With ease in / ease out, the ball gains speed slowly at the start and then slows down toward the end of the animation.

Instead of creating one value for every frame to animate your ball, you'll hold only key positions. These positions will be the **extreme** of a pose. In the ball example, the extreme pose positions are -1 and 1. You'll also hold key times that match the time of that key value. For example, if your animation is 2 seconds long, your extremes would be at 0 seconds for the starting pose on the left, 1 second for the ball to be on the right, and 2 seconds to take the ball back to the left again. All of the frames in between these times are interpolated.

- In the **Animation** group, create a new Swift file named **Animation.swift**.

In this file, you'll hold your animation data and create methods to return the interpolated value at a given time.

- Add the following:

```
struct Keyframe<Value> {
    var time: Float = 0
```

```
    var value: Value  
}
```

This code creates a structure to hold the animation key values and times. The value can be one of various types, so you make it generic.

► Now, add this:

```
struct Animation {  
    var translations: [Keyframe<float3>] = []  
    var repeatAnimation = true  
}
```

This structure holds an array of keyframes where each translation will be a `float3` value. `repeatAnimation` indicates whether to repeat the animation clip forever or play it just once.

► Add the following new method to `Animation`:

```
func getTranslation(at time: Float) -> float3? {  
    // 1  
    guard let lastKeyframe = translations.last else {  
        return nil  
    }  
    // 2  
    var currentTime = time  
    if let first = translations.first,  
        first.time >= currentTime {  
        return first.value  
    }  
    // 3  
    if currentTime >= lastKeyframe.time,  
        !repeatAnimation {  
        return lastKeyframe.value  
    }  
}
```

This method returns the interpolated keyframe.

Here's the breakdown:

1. Ensure that there are translation keys in the array, otherwise, return a `nil` value.
2. If the first keyframe occurs on or after the time given, then return the first key value. The first frame of an animation clip *should* be at keyframe `0` to give a starting pose.
3. If the time given is greater than the last key time in the array, then check whether you should repeat the animation. If not, then return the last value.



- Add the following code to the bottom of `getTranslation(at:)`:

```
// 1
currentTime = fmod(currentTime, lastKeyframe.time)
// 2
let keyFramePairs = translations.indices.dropFirst().map {
    (previous: translations[$0 - 1], next: translations[$0])
}
// 3
guard let (previousKey, nextKey) = (keyFramePairs.first {
    currentTime < $0.next.time
})
else { return nil }
// 4
let interpolant =
    (currentTime - previousKey.time) /
    (nextKey.time - previousKey.time)
// 5
return simd_mix(
    previousKey.value,
    nextKey.value,
    float3(repeating: interpolant))
```

Going through this code:

1. Use the modulo operation to get the current time within the clip.
2. Create a new array of tuples containing the previous and next keys for all keyframes, except the first one.
3. Find the first tuple of previous and next keyframes where the current time is less than the next keyframe time. The current time will, therefore, be between the previous and next keyframe times.
4. Use the interpolation formula to get a value between 0 and 1 for the progress percentage between the previous and next keyframe times.
5. Use `simd_mix` to interpolate between the two keyframes. (`interpolant` must be a value between 0 and 1.)

- Open `BallAnimations.swift`, and uncomment `ballTranslations`.

`ballTranslations` is an array of `Keyframes` with seven keys. The length of the clip is two seconds. You can see this by looking at the key time of the last keyframe.

In the x-axis, the ball will start off at position -1 and then move to position 1 at 0.35 seconds. It will hold its position until 1 second has passed, then return to -1 at 1.35 seconds. It will then hold its position until the end of the clip.



By changing the values in the array, you can speed up the throw and hold the ball for longer at either end.

► Open **Beachball.swift**, and replace `update(deltaTime:)` with:

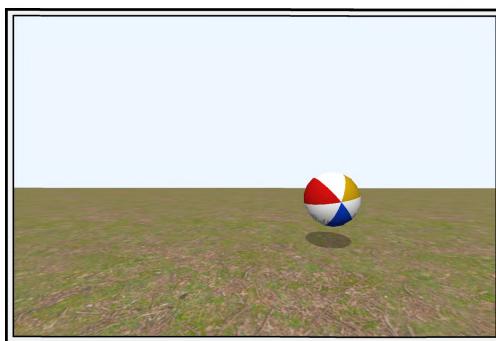
```
mutating func update(deltaTime: Float) {
    currentTime += deltaTime
    var animation = Animation()
    animation.translations = ballTranslations
    ball.position =
        animation.getTranslation(at: currentTime) ?? [0, 0, 0]
    ball.position.y += ball.size.y
}
```

Here, you load the animation clip with the generated keyframe translations.

Generally, you'll want to load the animation clip outside of the update, but for the sake of simplicity, in this example, handling things within `update(deltaTime:)` is fine.

You then extract the ball's position from the animation clip for the current time.

► Build and run the app, and watch as creepy invisible hands toss your ball around.



*Tossing the ball*

**Note:** Notice the trajectory of the ball on the y-axis. It currently goes up and down in diagonal straight lines. Better keyframing can fix this.

## Euler Angle Rotations

Now that you have the ball *translating* through the air, you probably want to *rotate* it as well. To express rotation of an object, you currently hold a `float3` with rotation angles on x, y and z axes. These are known as **Euler angles** after the mathematician Leonhard Euler. Euler is the man behind Euler's rotation theorem — a theorem which states that any rotation can be described using three rotation angles. This is OK for a single rotation, but interpolating between these three values doesn't work in a way that you may think.

- To create a rotation matrix, you've been calling this function, hidden in the math library in `Utility/MathLibrary.swift`:

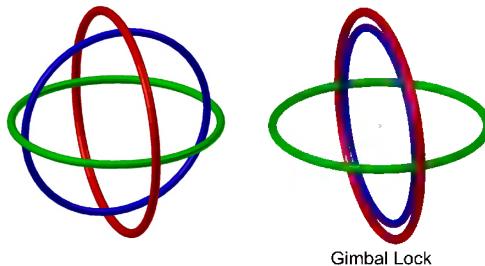
```
init(rotationAngle: float3) {
    let rotationX = float4x4(rotationX: angle.x)
    let rotationY = float4x4(rotationY: angle.y)
    let rotationZ = float4x4(rotationZ: angle.z)
    self = rotationX * rotationY * rotationZ
}
```

Here, the final rotation matrix is made up of three rotation matrices multiplied in a particular order. This order is not set in stone and is one of six possible orders. Depending on the multiplication order, you'll get a different rotation.

**Note:** Sometimes, these rotations are referred to as Yaw-Pitch-Roll. You'll see these names a lot in flight simulators. Depending on your frame of reference, if you're using the y-axis as up and down (remember that's not universal), then Yawing is about the y-axis, Pitching is about the x-axis and Rolling is about the z-axis.

For static objects within one rendering engine, this is fine. The main problem comes with animation and interpolating these angles.

As you proceed through a rotation interpolation if two axes become aligned you get the terrifyingly named **gimbal lock**.



Gimbal lock means that you've lost one axis of rotation. Because the inner axis rotations build on the outer axis rotation, the two rotations overlap and cause odd interpolation.

## Quaternions

Multiplying x, y and z rotations without compelling a sequence on them is impossible unless you involve the fourth dimension. In 1843, Sir William Rowan Hamilton did just that: he inscribed his fundamental formula for quaternion multiplication on to a stone on a bridge in Dublin.

$$i^2 = j^2 = k^2 = ijk = -1$$

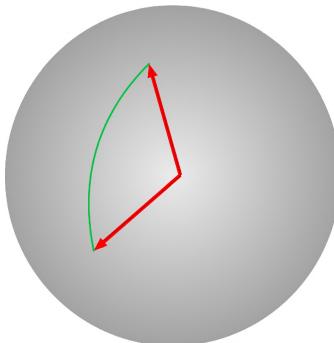
The formula uses four-dimensional vectors and complex numbers to describe rotations. The mathematics is complicated, but fortunately, you don't have to understand how quaternions work to use them.

The main benefit of quaternions are:

- They interpolate correctly when using spherical linear interpolation (or **slerp**).
- They never lose any axes of control.
- They always take the shortest path between two rotations unless you specifically ask for the longest path.

**Note:** If you're interested in studying the internals of quaternions, [references.markdown](#) contains further reading.

You don't have to write any complicated interpolation code, as `simd` has quaternion classes and methods that handle everything for you using `simd_slerp()`. The quaternions perform a spherical interpolation along the shortest path as shown in the following image.



*Spherical interpolation*

Internally in `simd`, quaternions are vectors of four elements, but Apple suggests that you treat them as abstract mathematical objects rather than delving into internal storage. That lets you off the hook for learning that the last element of the quaternion is the **real** part, and the first three elements are the **imaginary** part.

You'll switch from using Euler rotations to using quaternions for your rotations. Taking advantage of `simd` conversion of quaternions to and from rotation matrices, this switch is almost effortless.

- In the **Geometry** group, open **Transform.swift**, and add this property to **Transform**:

```
var quaternion = simd_quatf()
```

- In the extension where you define `modelMatrix`, change the definition of `rotation` to:

```
let rotation = float4x4(quaternion)
```

Your `Transforms` will now support quaternions instead of Euler angles. You should also change `rotation` so that it updates the quaternion.

- In `Transform`, change the definition of `rotation` to:

```
var rotation: float3 = [0, 0, 0] {
    didSet {
        let rotationMatrix = float4x4(rotation: rotation)
        quaternion = simd_quatf(rotationMatrix)
    }
}
```

The quaternion value will now stay in sync when you set a model's rotation.

- In the `Transformable` extension, add this:

```
var quaternion: simd_quatf {
    get { transform.quaternion }
    set { transform.quaternion = newValue }
}
```

With this syntactic sugar, when you refer to the `model.transform.quaternion`, you can now instead shorten it to `model.quaternion`.

To animate using quaternion rotation, you'll duplicate what you did for translations.

- Open `Animation.swift`, and add a new property to `Animation`:

```
var rotations: [Keyframe<simd_quatf>] = []
```

This time the keyframes will be quaternion values.

- Duplicate `getTranslation(at:)` to a new method called `getRotation(at:)`, that uses `rotations` and `quaternions` instead of `translations` and `floats`:

```
func getRotation(at time: Float) -> simd_quatf? {
    guard let lastKeyframe = rotations.last else {
        return nil
    }
    var currentTime = time
    if let first = rotations.first,
       first.time >= currentTime {
        return first.value
    }
    if currentTime >= lastKeyframe.time,
       !repeatAnimation {
        return lastKeyframe.value
    }
    currentTime = fmod(currentTime, lastKeyframe.time)
    let keyFramePairs = rotations.indices.dropFirst().map {
        (previous: rotations[$0 - 1], next: rotations[$0])
    }
```



```
guard let (previousKey, nextKey) = (keyFramePairs.first {
    currentTime < $0.next.time
}) else { return nil }
let interpolant =
    (currentTime - previousKey.time) /
    (nextKey.time - previousKey.time)
return simd_slerp(
    previousKey.value,
    nextKey.value,
    interpolant)
}
```

Note that you change the interpolation function to use `simd_slerp` instead of `simd_mix`. This does the necessary spherical interpolation.

► Open `BallAnimations.swift`, and uncomment `ballRotations`.

`ballRotations` is an array of rotation keyframes. The rotation starts out at 0, then rotates by 90° on the z-axis over several keyframes to a rotation of 0 at 0.35 seconds. The reason for rotating several times by 90° is because if you rotate from 0° to 360°, the shortest distance between those is 0°, so the ball won't rotate at all.

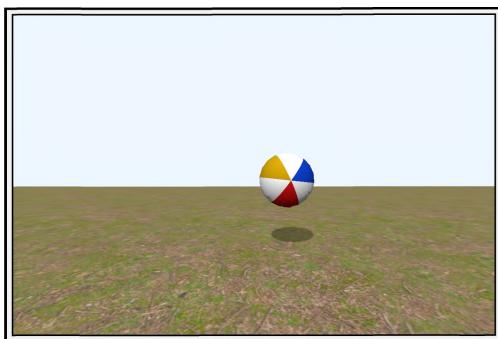
► Open `Beachball.swift`, and replace `update(deltaTime:)` with this:

```
mutating func update(deltaTime: Float) {
    currentTime += deltaTime
    var animation = Animation()
    animation.translations = ballTranslations
    animation.rotations = ballRotations
    ball.position =
        animation.getTranslation(at: currentTime)
        ?? float3(repeating: 0)
    ball.position.y += ball.size.y / 2
    ball.quaternion =
        animation.getRotation(at: currentTime)
        ?? simd_quatf()
}
```

You load the animation clip with both translation and rotation values. You then extract the values for `position` and `quaternion` for the given time.



- Build and run the app, and your ball moves back and forth with rotation.



*The ball rotates as it moves*

If you need more complex animations, you'll probably want to create the animation in a 3D app. The ball actually holds some transformation animation (made in Blender) in its USD file.

## USD and USDZ Files

One major problem to overcome is how to import animation from 3D apps. Model I/O can import .obj files, but they only hold static information, not animation. USD is a format devised by Pixar, which can hold massive scenes with textures, animation and lighting information. There are various file extensions:

- **.usd:** A Universal Scene Description (USD) file consists of assets or links to assets which allows multiple artists to work on the same scene. The file can contain mesh geometry, shading information, models, cameras and lighting.
- **.usdz:** A single archive file that contains all the files - not just links - necessary for rendering a model.
- **.usda:** This file is the USD file in text format. The ball included in this chapter's project is in .usda format so that you can open the file withTextEdit and inspect the contents.
- **.usdc:** This file is the USD file in binary format.

Apple has adopted USDZ — the archive derivation of the USD format — as their preferred augmented reality 3D format. Maya 2022, Houdini 19.0 and Blender 3.0 can import and export USD formats.

**Note:** As at the time of writing, Blender 3.0 can't export skeletal animation directly to USD. You can animate meshes, such as this ball, in Blender 3.0 and export the animation because the mesh is not attached to a skeleton. You'll read more about skeletal animation in the following chapter.

Apple also provides Reality Converter (<https://apple.co/3H8FKWd>), which allows you to convert from other formats, view and customize your USDZ files. Currently the supported formats are .obj, .fbx, .abc and .glTF.

Sketchfab (<http://sketchfab.com>) is a major provider and showcase of 3D models. All of their downloadable models are available and converted to the USDZ format.

## Animating Meshes

The file **beachball.usda** holds translation and rotation animation, and Model I/O can extract this animation. There are several ways to approach initializing this information, and you'll use the first in this chapter.

Model I/O transform components don't allow you to access the rotation and translation values directly, but provides you with a method that returns a transform matrix at a particular time. So for mesh transform animation you'll extract the animation data for every frame of the animation during the model loading process.

In the next chapter, when you work on skeletal animation, you'll have access to joint rotation and translation, so you'll load data only where there are keyframes, and use your interpolation methods to interpolate each frame.

**Note:** When writing your own engine, you'll have the choice to load this animation data up front for every frame, to match the transformation animation. You should consider the requirements of your game and what information your models hold. Generally it is more efficient to extract the loading code to a separate app which loads models and saves materials, textures and animation data into a more efficient format that matches your game engine. A good example of this asset pipeline is Apple's video and sample code From Art to Engine with Model I/O from WWDC 2017 (<https://developer.apple.com/videos/play/wwdc2017/610/>).

You'll be running your game at a fixed fps - generally 60, and you'll hold a transform matrix for every frame of animation.

- In the **Game** group, open **GameController.swift**, and add a new property:

```
static var fps: Double = 0
```

- At the top of `init(metalView:options:)`, add this:

```
Self.fps = Double(metalView.preferredFramesPerSecond)
```

You'll use `fps` as the standard frames per second for your app. You set `fps` right at the top of `init(metalView:options:)` because the models will use it when you initialize `GameScene`.

Model I/O can hold transform information on all objects within the `MDLAsset`. For simplicity, you'll hold a transform component on each `Mesh`, and just animate the transforms for the duration given by the asset.

- In the **Animation** group, create a new file named **TransformComponent.swift**, and replace the default code with:

```
import ModelIO

struct TransformComponent {
    let keyTransforms: [float4x4]
    let duration: Float
    var currentTransform: float4x4 = .identity
}
```

You'll hold all the transform matrices for each frame for the duration of the animation. For example, if the animation has a duration of 2.5 seconds at 60 frames per second, `keyTransforms` will have 150 elements. Later, you'll update all of the `Meshes`' `currentTransform` of every frame with the transform for the current frame taken from `keyTransforms`.

- Now, add the following to `TransformComponent`:

```
init(
    transform: MDLTransformComponent,
    object: MDLObject,
    startTime: TimeInterval,
    endTime: TimeInterval
) {
    duration = Float(endTime - startTime)
    let timeStride = stride(
        from: startTime,
```



```
        to: endTime,
        by: 1 / TimeInterval(GameController.fps))
keyTransforms = Array(timeStride).map { time in
    MDLTransform.globalTransform(
        with: object,
        atTime: time)
}
```

This initializer receives an `MDLTransformComponent` from either an asset or a mesh and then create all the transform matrices for every frame for the duration of the animation.

- Add the following to `TransformComponent`:

```
mutating func getCurrentTransform(at time: Float) {
    guard duration > 0 else {
        currentTransform = .identity
        return
    }
    let frame = Int(fmod(time, duration) *
Float(GameController.fps))
    if frame < keyTransforms.count {
        currentTransform = keyTransforms[frame]
    } else {
        currentTransform = keyTransforms.last ?? .identity
    }
}
```

You retrieve a transform matrix at a particular, given, time. You calculate the current frame of the animation from the time. Using the floating point modulo operation `fmod` function, you can loop the animation. For example, if the animation is 2.5 seconds long, at 60 frames per second, that would mean there are 150 frames in the animation. If the current time is 5 seconds, that will be the last frame of the animation looped for a second time, and the current frame will be 150.

You save the current transform on the transform component. You'll use the transform to update the position of the mesh vertices shortly.

The animation will need the start and end time from the asset.

- Open `Mesh.swift`, and add a new initializer:

```
init(
    mdlMesh: MDLMesh,
    mtkMesh: MTKMesh,
    startTime: TimeInterval,
    endTime: TimeInterval
```



```
) {  
    self.init(mdlMesh: mdlMesh, mtkMesh: mtkMesh)  
}
```

► Open **Model.swift**, and in `init(name:)`, locate `Mesh(mdlMesh: $0.0, mtkMesh: $0.1)` inside the `meshes` assignment.

► Change `Mesh(mdlMesh: $0.0, mtkMesh: $0.1)` to:

```
Mesh(  
    mdlMesh: $0.0,  
    mtkMesh: $0.1,  
    startTime: asset.startTime,  
    endTime: asset.endTime)
```

You use your new initializer in place of the old one.

► Back in **Mesh.swift**, add a new property to `Mesh`:

```
var transform: TransformComponent?
```

► Add the following code to the end of  
`init(mdlMesh:mtkMesh:startTime:endTime:)`:

```
if let mdlMeshTransform = mdlMesh.transform {  
    transform = TransformComponent(  
        transform: mdlMeshTransform,  
        object: mdlMesh,  
        startTime: startTime,  
        endTime: endTime)  
} else {  
    transform = nil  
}
```

Now that you've set up the transform component with animation, you'll be able to use it when rendering each frame.

► Open **Model.swift**, and add a new property to keep track of elapsed game time:

```
var currentTime: Float = 0
```

► Change `let meshes: [Mesh]` to:

```
var meshes: [Mesh]
```

You'll update the meshes every frame, so you need write access.

- Add a new method to Model:

```
func update(deltaTime: Float) {
    currentTime += deltaTime
    for i in 0..
```

Here, you update all the transforms in the model ready for rendering.

- In `render(encoder:uniforms:params:)`, remove:

```
uniforms.modelMatrix = transform.modelMatrix
uniforms.normalMatrix = uniforms.modelMatrix.upperLeft
```

- And:

```
encoder.setVertexBytes(
    &uniforms,
    length: MemoryLayout<Uniforms>.stride,
    index: UniformsBuffer.index)
```

Some models may have multiple meshes, and you'll have to change the model matrix for each mesh.

- At the top of the `for mesh in meshes` loop, add this:

```
let currentLocalTransform =
    mesh.transform?.currentTransform ?? .identity
uniforms.modelMatrix =
    transform.modelMatrix * currentLocalTransform
uniforms.normalMatrix = uniforms.modelMatrix.upperLeft
encoder.setVertexBytes(
    &uniforms,
    length: MemoryLayout<Uniforms>.stride,
    index: UniformsBuffer.index)
```

Here, you combine the model's world transform with the mesh's transform and send the uniforms to the vertex shader.

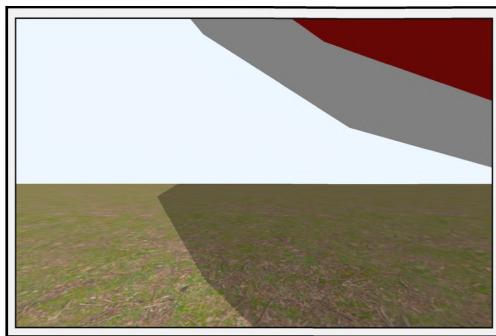
- Open `GameScene.swift`, and in `update(deltaTime:)`, replace `beachball.update(deltaTime:)` with:

```
for model in models {
    model.update(deltaTime: deltaTime)
}
```



With this change, you'll use the animation from the model file rather than from Beachball

- Build and run the app.



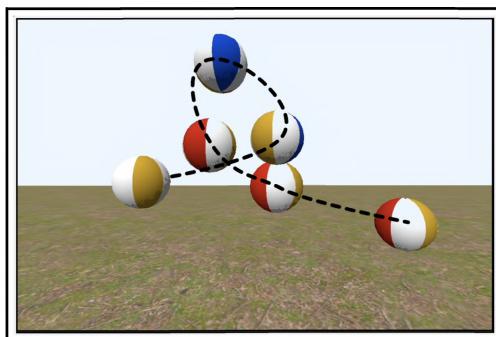
*The ball animates out of frame*

The ball is huge and mostly out of frame because of its initial scale. When you weren't using the transform information from the file, you needed to set the initial scale in `GameScene`, but now you can remove this.

- Still in `GameScene.swift`, remove:

```
ball.scale = 100
```

- Build and run the app, and you'll see an epic animation.



*The beachball USD animation*

## Challenge

For this challenge, you'll download and add some of Apple's animated USDZ samples to your scene.

1. Download the toy robot, the drummer and the biplane from Apple's AR Quick Look Gallery (<https://apple.co/3H9nagR>).
2. Add the models to your project, and load them into GameScene. Add them to the `models` array.
3. The scale is too big for your scene, so you'll need to set the scale to `0.1` and rotate on `y` by `.pi` (that's  $180^\circ$ ).

**Note:** There's no challenge sample project available for this challenge due to Apple's copyright on the models.

Now that you've learned about simple mesh animation, you're ready to move on to animating a jointed figure.

## Key Points

- Animation used to be done using frame-by-frame, but nowadays, animation is created on computers and is usually done using keyframes and interpolation.
- Procedural animation uses physics to compute values at a given time.
- Axis-aligned bounding boxes are useful when calculating collisions between aligned objects.
- Keyframes are generally extreme values between which the computer interpolates. This chapter demonstrates keyframing transformations, but you can animate anything. For example, you can set keyframes for color values over time.
- You can use any formula for interpolation, such as linear, or ease-in / ease-out.
- Interpolating quaternions is preferable to interpolating Euler angles.
- USD files are common throughout the 3D industry because you can keep the entire pipeline stored in the flexible format that USD provides.



# Chapter 24: Character Animation

In the previous chapter, you learned how to move objects over time using keyframes. Imagine how long it would take to create a walk cycle for a human figure by typing out keyframes. This is the reason why you generally use a 3D app, like Blender or Maya, to create your models and animations. You then export those animations to your game or rendering engine of choice.



## Skeletal Animation

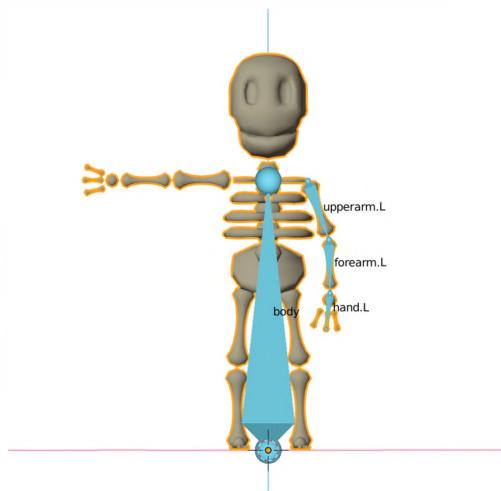
Rarely, will you move the entire character when you’re animating it. Instead, you’ll move parts of the mesh, such as an arm, rather than the whole thing. Using a 3D app, the rigger creates a **skeleton** — in Blender, this is known as an **armature**. She assigns bones and other controls to parts of the mesh so that the **animator** can transform the bones and record the movement into an **animation clip**.

You’ll use **Blender 3.0** to examine an animated model and understand the principles and concepts behind 3D animation.

**Note:** If you haven’t installed Blender 3.0 yet, it’s free, and you can download it from <https://www.blender.org>.

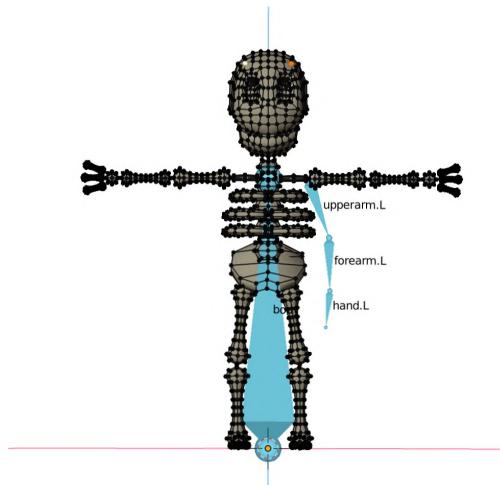
- Go to the resources folder for this chapter, and open **skeleton.blend** in **Blender 3.0**.

You’ll see something like this:



*The skeleton model in Blender 3.0*

- Before examining the bones further, left-click on the skeleton's head to select the skeleton object. Press the **Tab** key to switch to **Edit Mode**:



*The skeleton mesh*

Here, you can see all of the skeleton's vertices. This is the original model, which you can export as a static .obj file. The skeleton has its arms stretched out in what's known as the **bind pose**. This is a standard pose for figures as it makes it easier to add animation bones to the figure.

- Press the **Tab** key to go back to **Object Mode**.

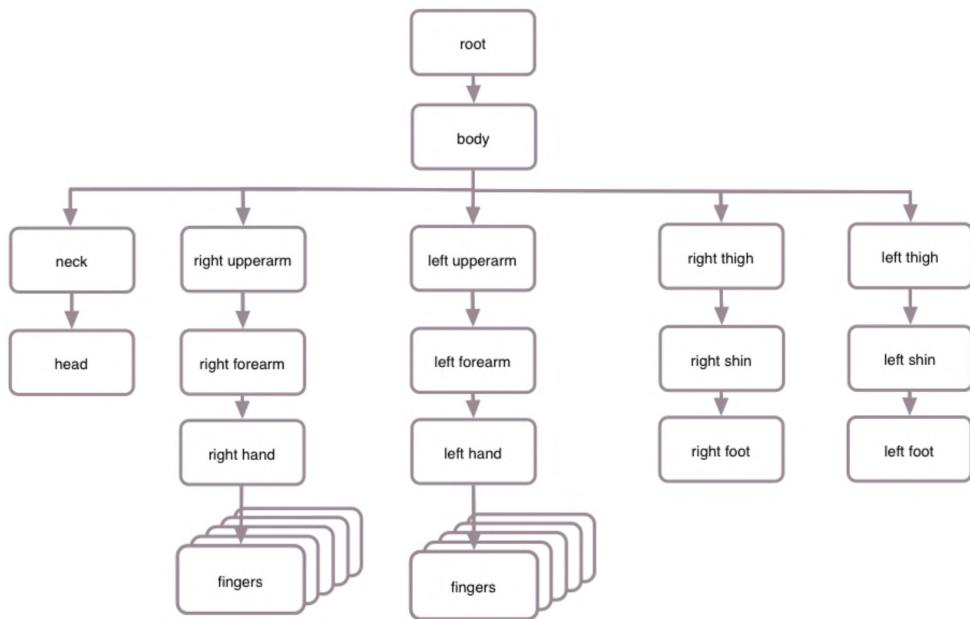
To animate the figure, you need to control groups of vertices. For example, to rotate the head, you'd rotate all of the head's vertices.

Rigging a figure in Blender means creating an armature with a hierarchy of **joints**. Joints and bones are generally used synonymously, but a bone is simply a visual cue to see which joint affects which vertices.

The general process of creating a figure for animation goes like this:

1. Create the model.
2. Create an armature with a hierarchy of joints.
3. Apply the armature to the model with automatic weights.
4. Use weight painting to change which vertices go with each joint.

Just as in the song *Dem Bones*, “The toe bone’s connected to the foot bone,” this is how a typical rigged figure’s joint hierarchy might look:



A joint hierarchy

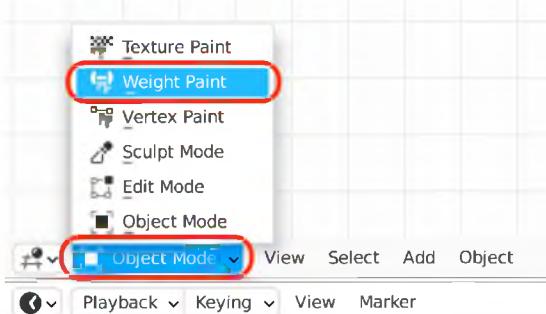
In character animation, it’s (usually) all about rotation — your bones don’t translate unless you have some kind of disjointing skeleton. With this hierarchy of joints, when you rotate one joint, all the child joints follow. Try bending your elbow without moving your wrist. Because your wrist is lower in the hierarchy, even though you haven’t actively changed the wrist’s position and rotation, it still follows the movement of your elbow. This type of movement is known as **forward kinematics** and is what you’ll be using in this chapter. It’s a fancy name for making all child joints follow.

**Note:** Inverse kinematics allows the animator to make actions, such as walk cycles, more easily. Place your hand on a table or in a fixed position. Now, rotate your elbow and shoulder joint with your hand fixed. The hierarchical chain no longer moves your hand as in forward kinematics. As opposed to forward kinematics, the mathematics of inverse kinematics is quite complicated.

The skeleton model that you’re looking at in Blender has a limited rig for simplicity. It has four bones: the body, left upper arm, left forearm and left hand. Each of these joints controls a group of vertices.

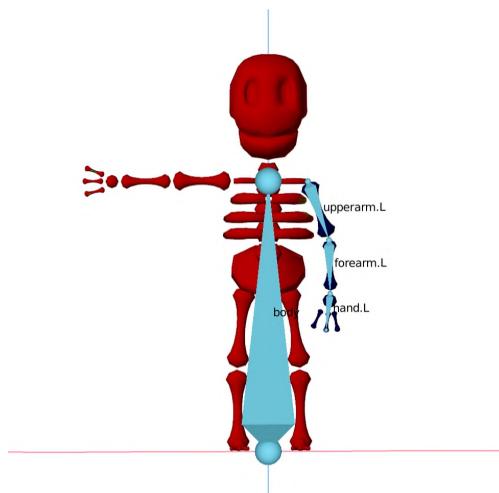
## Weight Painting in Blender

- Left-click the skeleton’s head.
- At the bottom of the Blender window, click on the drop-down that currently reads **Object Mode**, and change it to **Weight Paint**.



*Weight Paint Dropdown*

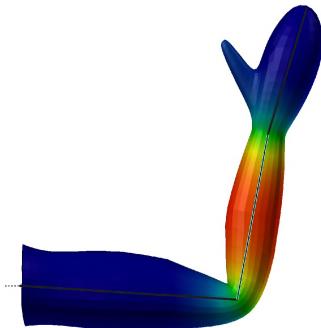
The weight painting editor shows you how each bone affects the vertices. Currently the body vertex group is selected, which is attached to the body bone. All vertices affected by the body bone are shown in red. The arm mesh has its own bones and are shown in blue.



*The skeleton's body bone weights*

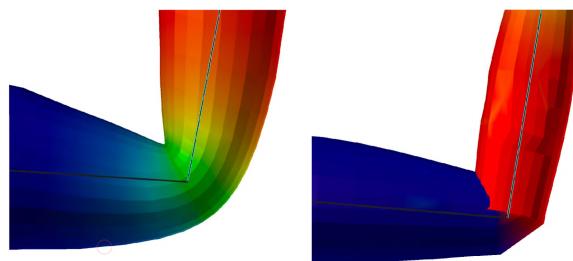
The process of weight painting and binding each bone to the vertices is called **skinning**. Unlike human arms, the skeleton's arm bones here have space between them, so, in this case, all mesh is assigned to only one bone. However, if you're rigging a human arm, you would typically weight the vertices to multiple bones.

Here's a typically weighted arm with the forearm selected to show gradual blending of weights at the elbow and the wrist.



*A weighted arm*

This is a side-by-side example of blended and non-blended weights at the elbow joint with the forearm selected:



*Blended and non-blended weights*

The blue area indicates no weighting, whereas the red area indicates total weighting. You can see in the right image, the forearm vertices dig uncomfortably into the upper arm vertices, but in the left image, the vertices move more evenly around the elbow joint.

At the elbow, where the vertices are green, the vertex weighting would be 50% to the upper arm, and 50% to the forearm. When the forearm rotates, the green vertices will rotate at 50% of the forearm's rotation. By blending the weights gradually, you can achieve an even deformation of vertices over the joint.

## Animation in Blender

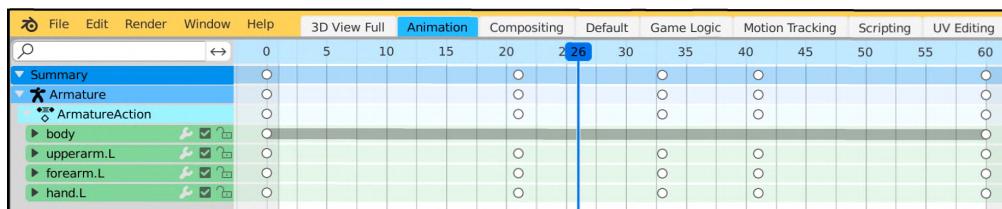
- Select the drop-down at the bottom of the window that currently reads **Weight Paint**, and go back into **Object Mode**.
- Press the **space bar** to start an animation.

Your skeleton will now get friendly and wave at you. This wave animation is a 60 frame looping animation clip.

- At the top of Blender's window, click the **Animation** tab to show the Animation workspace.



You can now see the animation keys at the top left in the **Dope Sheet**. The dope sheet is a summary of the keyframes in the scene. It lists the joints on the left, and each circle in the dope sheet means there's a keyframe at that frame.



*The dope sheet*

**Note:** Although animated transformations are generally rotations, the keyframe can be a translation or a scale. You can click the arrow on the left of the joint name to see the specific channel the key is set on.

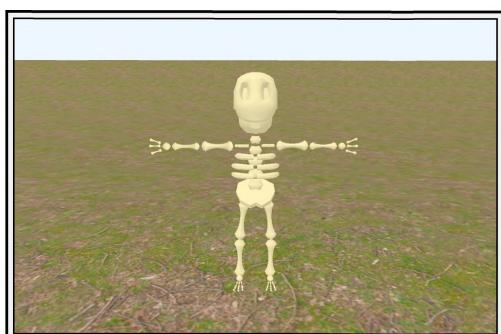
► Press **space bar** to stop the animation if it's still going. **Scrub** through the animation by dragging the playhead at the top of the pane (the blue rectangle with 26 in it in the above image). Pause the playhead at each set of keyframes. Notice the position of the arm. At each keyframe, the arm is in an extreme position. Blender interpolates all the frames between the extremes.

Now that you've had a whirlwind tour of how to create a rigged figure and animate it in Blender, you'll move on to learning how to render it in your rendering engine.

**Note:** You've only skimmed the surface of creating animated models. If you're interested in creating your own, you'll find some additional resources in [references.markdown](#).

## The Starter App

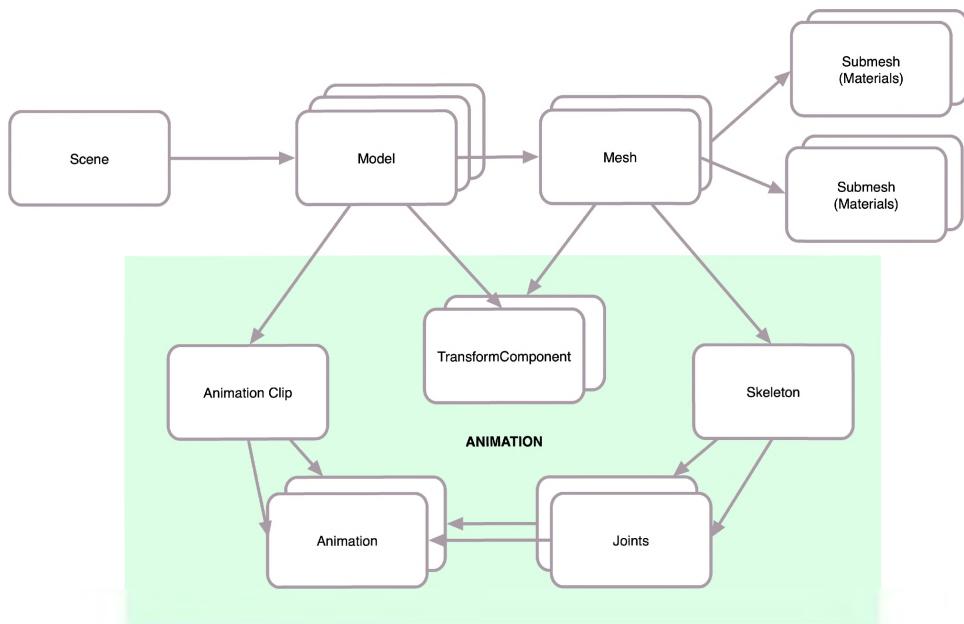
- In Xcode, open the starter project. The scene is a skeleton character rendered with the forward renderer, using the PBR shader, without shadows.
- Build and run the app.



You see a skeleton model, which was exported from Blender in **.fbx** format and converted to **.usda** format. You can open **skeletonWave.usda**, in the **Models / Skeleton** group, with a text editor to see what's inside. The file contains an animation, but the skeleton won't wave until you've implemented the chapter code.

# Implementing Skeletal Animation

Importing a skeletal animation into your app is a bit more difficult than importing a simple .obj file or a USDZ file with transform animation, because you have to deal with the joint hierarchy and joint weighting. You'll read in the data from the USD file and restructure it to fit your rendering code. This is how the objects will fit together in your app:



*The code architecture*

Each model could have a number of animation clips, such as **walk** and **wave**. Each animation clip has a list of animations for a particular joint. Each mesh can have a skeleton that holds a list of joint names, and using the joint name as a key, you'll be able to access the correct animation for that joint.

**TransformComponent** that you created in the previous chapter still remains for any transform animation. The starter project has several extra helper files in the **Animation** group to aid with importing the skeleton and animations.

- **Skeleton.swift**: To create the **Mesh**'s skeleton, you'll use the **MDLAnimationBindComponent** from the **mdlMesh**, if there is one. **Skeleton** holds the joint names in an array, and also the joints' parent indices in another array.

- **AnimationComponent.swift:** To load the animations for the asset, `load(animation:)` iterates through the joints and loads up `Animations` for each joint. These are all combined into an `AnimationClip`.
  - **AnimationClip.swift:** `AnimationClip` is a collection of `Animations`. `Model` will hold a dictionary of these `AnimationClips` keyed on the animation's name.
  - **Animation.swift:** You created `Animation` in the previous chapter for rotations and translations. The loading code now includes scale transformations.
- In the **Geometry** group, open **Model.swift**, and add a new property to `Model` to hold the model's animation clips:

```
let animations: [String: AnimationClip]
```

- At the end of `init(name:)`, add the following to load the animations:

```
// animations
let assetAnimations = asset.animations.objects.compactMap {
    $0 as? MDLPackedJointAnimation
}
let animations
    = Dictionary(uniqueKeysWithValues: assetAnimations.map {
        ($0.name, AnimationComponent.load(animation: $0))
    })
self.animations = animations
```

Here, you extract all the `MDLPackedJointAnimation` objects from the asset and load them using the provided loading code. The result will be a dictionary of animation clips keyed by animation name, held in `animations`.

- After the previous code, add this:

```
animations.forEach {
    print("Animation:", $0.key)
}
```

A list of the available animations will print out in the debug console so that you can use the name later.

- Build and run the app.

```
loaded texture: plane-color
Animation: /skeletonWave/Animations/wave
```

*The debug console*

The skeleton is still in his bind pose, but you'll see the message in the debug console to show that the animation has loaded.

- Remove the previous print closure.

You've just loaded a set of animations listing translation, rotation and scaling on all joints. Now to set up the meshes' skeletons.

- In the **Geometry** group, open **Mesh.swift**, and add a new property to **Mesh**:

```
let skeleton: Skeleton?
```

- At the top of **init(mdlMesh:mtkMesh:)**, initialize the skeleton using **mdlMesh**'s **MDLAnimationBindComponent**:

```
let skeleton =  
    Skeleton(animationBindComponent:  
        (mdlMesh.componentConforming(to: MDLComponent.self)  
            as? MDLAnimationBindComponent))  
    self.skeleton = skeleton
```

You've now loaded up a skeleton with joints. When rendering the skeleton, you'll be able to access the model's current animation and apply it to the mesh's skeleton joints.

- Add a debug print statement after the previous code to show the joints:

```
skeleton?.jointPaths.forEach {  
    print($0)  
}
```

- Build and run the app, and in the debug console, you'll see a listing of the skeleton model's four joints:

```
/body  
/body/upperarm_L  
/body/upperarm_L/forearm_L  
/body/upperarm_L/forearm_L/hand_L
```

These joints correspond to the bones that you previously saw in Blender.

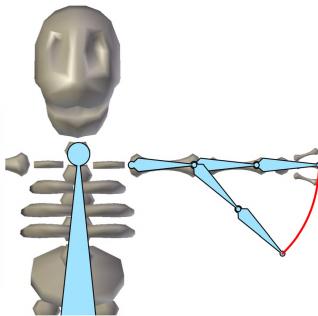
- Remove the **forEach print** debug closure.



## Loading the Animation

To update the skeleton's pose every frame, you'll create a method that takes the animation clip and iterates through the joints to update each joint's position for the frame.

First, you'll create a method on `AnimationClip` that gets the pose for a joint at a particular time. This will use the interpolation methods that you've already created in `Animation`. The main difference is that these poses will be in **joint space**. For example, in this animation, the forearm swings by 45°. All the other joints' rotations and translations will be 0.



- In the **Animations** group, open `AnimationClip.swift`, and add a new method to `AnimationClip`:

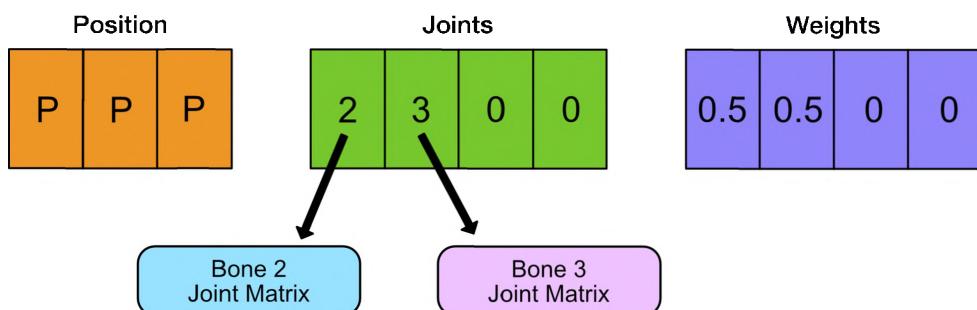
```
func getPose(at time: Float, jointPath: String) -> float4x4? {  
    guard let jointAnimation = jointAnimation[jointPath],  
        let jointAnimation = jointAnimation  
    else { return nil }  
    let rotation =  
        jointAnimation.getRotation(at: time) ?? simd_quatf()  
    let translation =  
        jointAnimation.getTranslation(at: time) ?? float3(repeating:  
    0)  
    let scale =  
        jointAnimation.getScale(at: time) ?? float3(repeating: 0)  
    let pose = float4x4(translation: translation) *  
    float4x4(rotation)  
        * float4x4(scaling: scale)  
    return pose  
}
```

Here, you retrieve the interpolated transformation, made up of rotation, translation and scale, for a given joint at a given time. You then create a transformation matrix for the joint and return it as the pose. This is much the same code as you used earlier for retrieving a transform at a particular time.

## The Joint Matrix Palette

You're now able to get the pose of a joint. However, each vertex is weighted to up to four joints. You saw this in the earlier elbow example, where some vertices belonging to the lower arm joint would get 50% of the upper arm joint's rotation. Soon, you'll change the default vertex descriptor to load vertex buffers with four joints and four weights for each vertex. This set of joints and weights is known as the **joint matrix palette**.

The vertex function will sample from each of these joint matrices and, using the weights, will apply the transformation matrix to each vertex. The following image shows a vertex that is assigned 50% to joint 2 and 50% to joint 3. The other two joint indices are unused.



After multiplying the vertex by the projection, view and model matrices, the vertex function will multiply the vertex by a weighting of each of the joint transforms. Using the example in the image above, the weighting will be 50% of Bone 2's joint matrix and 50% of Bone 3's joint matrix.

► Open **Skeleton.swift**, and create a new method in **Skeleton**:

```
func updatePose(
    animationClip: AnimationClip?,
    at time: Float
) {
```

This method, when you've completed it, will iterate through the joints and fill a joint matrix palette buffer with the current pose for each joint.

You'll send this buffer to the GPU's vertex function, so that each vertex will be able to access all of the joint matrices that it is weighted to.

- Add the following to updatePose(animationClip:at:):

```
guard let paletteBuffer = jointMatrixPaletteBuffer
else { return }
var palettePointer = paletteBuffer.contents().bindMemory(
    to: float4x4.self,
    capacity: jointPaths.count)
guard let animationClip = animationClip else {
    palettePointer.initialize(
        repeating: .identity,
        count: jointPaths.count)
    return
}
```

You initialize the buffer pointer and bind the contents of the buffer to an array of 4x4 matrices. If an animation clip is not loaded, initialize the buffer with identity matrices and return without updating the joints.

- Now, to iterate through the skeleton's joints, add the following:

```
var poses =
    [float4x4](repeatElement(.identity, count: jointPaths.count))
for (jointIndex, jointPath) in jointPaths.enumerated() {
    // 1
    let pose = animationClip.getPose(
        at: time * animationClip.speed,
        jointPath: jointPath) ?? restTransforms[jointIndex]
    // 2
    let parentPose: float4x4
    if let parentIndex = parentIndices[jointIndex] {
        parentPose = poses[parentIndex]
    } else {
        parentPose = .identity
    }
    poses[jointIndex] = parentPose * pose
}
```

Going through this code:

1. You retrieve the transformation pose, if there is one, for the joint for this frame. `restTransform` gives a default pose for the joint.
2. The poses array is in flattened hierarchical order, so you can be sure that the parent of any joint has already had its pose updated. You retrieve the current joint's parent pose, concatenate the pose with the current joint's pose and save it in the poses array.

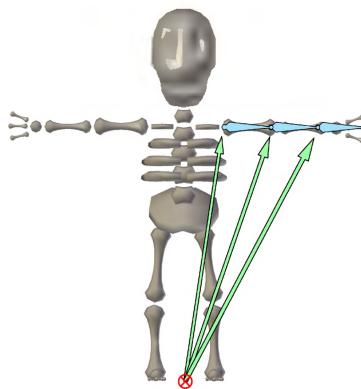


## The Inverse Bind Matrix

- Examine the properties held on Skeleton.

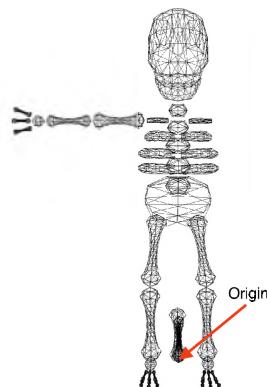
When you first instantiate the skeleton, you load these properties from the data loaded by Model I/O.

One of the properties on Skeleton is `bindTransforms`. This is an array of matrices, one element for each joint, that transforms vertices into the local joint space.



*The bind pose*

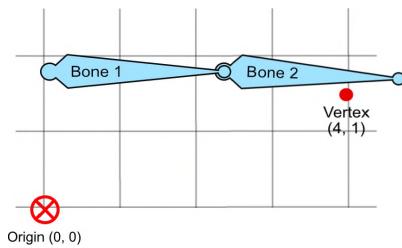
When all the joint transforms are set to identity, that's when you'll get the bind pose. If you apply the inverse bind matrix to each joint, it will move to the origin. The following image shows the skeleton's joints all multiplied by the inverse bind transform matrix.



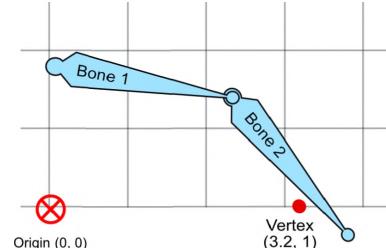
*The inverse bind matrix applied to all joints*

Why is this useful? Each joint should rotate around its base. To rotate an object around a particular point, you first need to translate the point to the origin, then do the rotation, then translate back again. (Review Chapter 5, “3D Transformations” if you’re unsure of this rotation sequence.)

In the following image, the vertex is located at (4, 1) and bound 100% to Bone 2. With rotations 10° on Bone 1 and 40° on Bone 2, the vertex should end up at about (3.2, 0) as shown in the right-hand image.



Before applying the pose



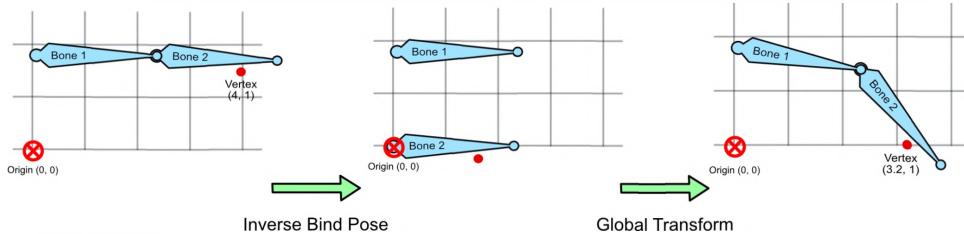
After applying the pose

When you currently render your vertices, you multiply each vertex position by the projection, view and model matrices in the vertex function. To get this example vertex in the correct position for the right-hand image, you’ll also have to multiply the vertex position by both Bone 1’s transform and Bone 2’s transform.

► Add the following code to the end of the previous `for` loop, after setting `poses[jointIndex]`:

```
palettePointer.pointee =
    poses[jointIndex] * bindTransforms[jointIndex].inverse
palettePointer = palettePointer.advanced(by: 1)
```

You translate the pose back to the origin with the inverse bind transform, and combine it with the current pose into the final joint palette matrix.



With all of the frame data set up, you can now set the pose.

- Open **Model.swift**, and in `update(deltaTime:)`, replace the existing animation code:

```
for i in 0..    meshes[i].transform?.getCurrentTransform(at: currentTime)  
}
```

- With:

```
for i in 0..    var mesh = meshes[i]  
    if let animationClip = animations.first?.value {  
        mesh.skeleton?.updatePose(  
            animationClip: animationClip,  
            at: currentTime)  
    }  
    mesh.transform?.getCurrentTransform(at: currentTime)  
    meshes[i] = mesh  
}
```

You take the first animation in the list of animations and, if there is an animation, update the pose for the current time. Update the current transform as well, as you were doing before.

**Note:** Currently USDZ files only hold one animation. With Blender not yet exporting skeletal animation as of version 3.0, it's difficult to get multiple animations into one USD file without hand editing a .usda file. Apple suggests, with some judicious coding, you could load multiple USDZ files, one with the geometry and skeleton, and others with solely the animation. As time goes on, and Blender improves, there will likely be better alternatives.

All of the meshes are now in position and ready to render.

- In `render(encoder:uniforms:params:)`, at the top of the loop `for mesh in meshes`, add this:

```
if let paletteBuffer = mesh.skeleton?.jointMatrixPaletteBuffer {  
    encoder.setVertexBuffer(  
        paletteBuffer,  
        offset: 0,  
        index: JointBuffer.index)  
}
```



You set up the joint matrix palette buffer so that the GPU can read it. The vertex shader function will take in this palette and apply the matrices to the vertices.

- In the **Shaders** group, open **Vertex.h**, and add two attributes to **VertexIn**:

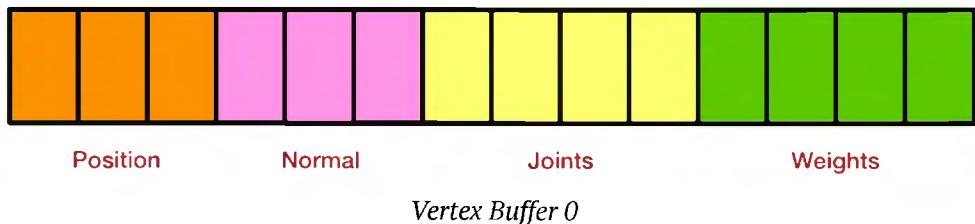
```
ushort4 joints [[attribute(Joints)]];  
float4 weights [[attribute(Weights)]];
```

The attribute constants **Joints** and **Weights** were set up for you in the starter project in **Common.h** in **Attributes**.

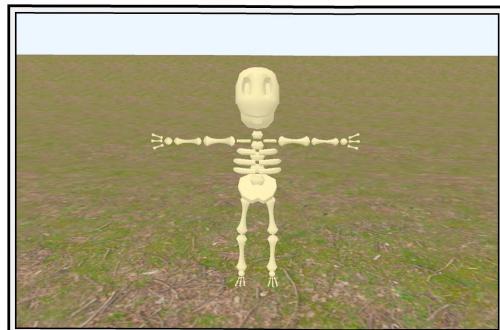
To match **VertexIn**, you'll need to update **Model**'s vertex descriptor.

- Open **VertexDescriptor.swift**, and uncomment the two extra vertex attributes for joints and weights. Model I/O, when loading the file, will now load joint index and joint weight information to the model's vertex buffers.

Your vertex buffer layout will now look like this:



- Build and run the app to ensure that everything still works:



*No obvious changes yet*

## Updating the Vertex Shader

- In the **Shaders** group, open **Shaders.metal**, and add a new parameter to **vertex\_main**:

```
constant float4x4 *jointMatrices [[buffer(JointBuffer)]]
```

- At the top of **vertex\_main**, replace the **float4 position** assignment with:

```
bool hasSkeleton = true;
float4 position = in.position;
float4 normal = float4(in.normal, 0);
```

Some models will have skeletons and joint matrices, but others, such as the ground plane won't. You'll have to set up a conditional to determine which type of model you are rendering. For the moment you assume that all models have a joint matrix palette.

- After the code you just added, add the following code to combine the joint matrix and weight data with the position and normal:

```
if (hasSkeleton) {
    float4 weights = in.weights;
    ushort4 joints = in.joints;
    position =
        weights.x * (jointMatrices[joints.x] * position) +
        weights.y * (jointMatrices[joints.y] * position) +
        weights.z * (jointMatrices[joints.z] * position) +
        weights.w * (jointMatrices[joints.w] * position);
    normal =
        weights.x * (jointMatrices[joints.x] * normal) +
        weights.y * (jointMatrices[joints.y] * normal) +
        weights.z * (jointMatrices[joints.z] * normal) +
        weights.w * (jointMatrices[joints.w] * normal);
}
```

You take each joint to which the vertex is bound, calculate the final position and normal, and then take the weighted part of that calculation.

If the function constant **hasSkeleton** is false, you'll just use the original **position** and **normal**.

- Change the `VertexOut` out assignment to:

```
VertexOut out {
    .position = uniforms.projectionMatrix * uniforms.viewMatrix
        * uniforms.modelMatrix * position,
    .uv = in.uv,
    .color = in.color,
    .worldPosition = (uniforms.modelMatrix * position).xyz,
    .worldNormal = uniforms.normalMatrix * normal.xyz,
    .worldTangent = 0,
    .worldBitangent = 0,
    .shadowPosition =
        uniforms.shadowProjectionMatrix * uniforms.shadowViewMatrix
        * uniforms.modelMatrix * position
};
```

You use `position` and `normal` instead of `in.position` and `in.normal`. You should also pre-multiply the tangent and bitangent properties too, but for brevity, you set `worldTangent` and `worldBitangent` properties to zero.

- Build and run the app.

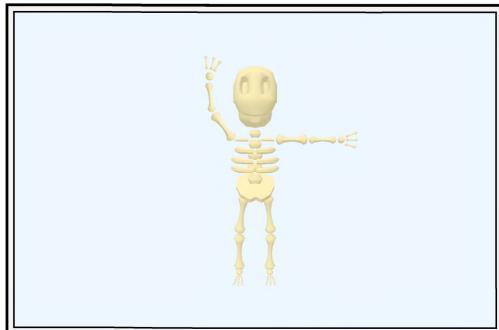
You'll get a run time error: **failed assertion Draw Errors Validation Vertex Function(vertex\_main): missing buffer binding at index 15 for jointMatrices[0].**

This is because you're rendering the ground, which doesn't have any joint matrices.

- Open `GameScene.swift`, and in `init()`, change `models = [ground, skeleton]` to:

```
models = [skeleton]
```

- Build and run the app, and your animated skeleton will now wave at you.



Skeleton waving

Of course you'll want to render the ground, so you'll need to tell the GPU pipeline

that it has to conditionally prepare two different vertex functions, depending on whether the mesh has a skeleton or not.

► Undo the previous change to `models`.

## Function Specialization

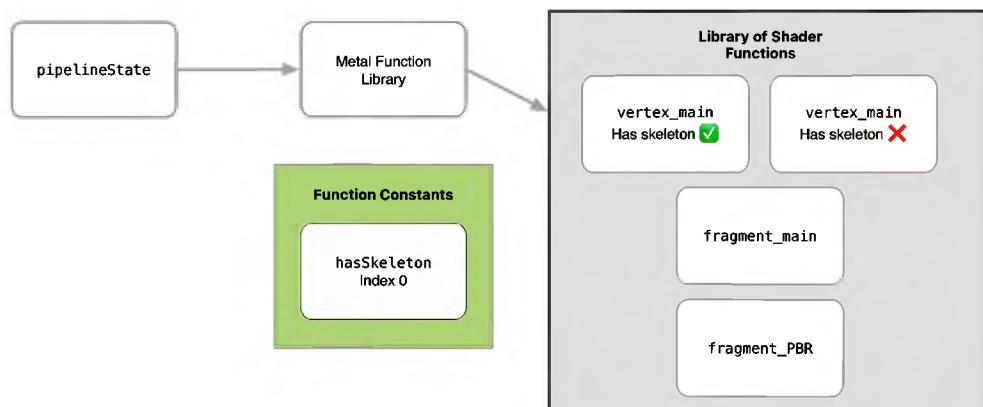
Over the years there has been much discussion about how to render conditionally. For example, in your fragment shaders when rendering textures, you use the Metal Shading Language function `is_null_texture(textureName)` to determine whether to use the value from the material or a texture.

To test whether or not you have a joint matrix, you don't have a convenient MSL function.

Should you create separate short fragment shaders for different conditionals? Or should you have one long “uber” shader with all of the possibilities listed conditionally? **Function specialization** deals with this problem, and allows you to create one shader that the compiler turns into separate shaders.

When you create the model's pipeline state, you set the Metal functions in the Metal Shading library, and the compiler packages them up. At this stage, you can create properties, and assign them index numbers to deal with conditional states. You can then pass these properties to the Metal library when you create the shader functions. The compiler will examine the functions and generate specialized versions of them.

In the shader file, you reference the properties by their index numbers.



*Function constants*

- In the **Render Passes** group, open **Pipelines.swift**.

You'll first create a set of function constant values that will indicate whether to render with animation.

- Add this new method to **PipelineStates**:

```
static func makeFunctionConstants(hasSkeleton: Bool)
-> MTLFunctionConstantValues {
    let functionConstants = MTLFunctionConstantValues()
    var property = hasSkeleton
    functionConstants.setConstantValue(
        &property,
        type: .bool,
        index: 0)
    return functionConstants
}
```

**MTLFunctionConstantValues** is a set that contains a Boolean value depending on whether a skeleton exists. You defined a Boolean value here, but values can be any type specified by **MTLDataType**. On the GPU side, you'll soon create a Boolean constant using the same index value. In functions that use these constants, you can conditionally perform tasks.

- Change the signature of **createForwardPS0()** to:

```
static func createForwardPS0(hasSkeleton: Bool = false)
-> MTLRenderPipelineState {
```

- At the top of **createForwardPS0(hasSkeleton:)**, change the assignment to **vertexFunction** to:

```
let functionConstants =
    makeFunctionConstants(hasSkeleton: hasSkeleton)
let vertexFunction = try? Renderer.library?.makeFunction(
    name: "vertex_main",
    constantValues: functionConstants)
```

Here, you tell the compiler to create a library of functions using the function constants set. The compiler creates multiple shader functions and optimizes any conditionals in the functions.

Repeat this for **createForwardTransparentPS0()**.

- Change the signature of `createForwardTransparentPSO()` to:

```
static func createForwardTransparentPSO(hasSkeleton: Bool =  
    false)  
-> MTLRenderPipelineState {
```

- At the top of `createForwardTransparentPSO(hasSkeleton:)`, change the assignment to `vertexFunction`:

```
let functionConstants =  
    makeFunctionConstants(hasSkeleton: hasSkeleton)  
let vertexFunction = try? Renderer.library?.makeFunction(  
    name: "vertex_main",  
    constantValues: functionConstants)
```

Currently, you set the same pipeline for all models. You create a standard pipeline state for rendering models, one for rendering with transparency and one for rendering shadows. Generally, when you have any complexity, you'll have to work out a system appropriate for your app to manage all your various pipeline states. You could use function specialization where possible, or create different vertex and fragment functions.

In this app, the time when you know whether your model has a skeleton or not, is when you load Mesh.

- Open `Mesh.swift`, and add a new property to `Mesh`:

```
var pipelineState: MTLRenderPipelineState
```

- Add the following code to the end of `init(mdlMesh:mtkMesh:)`:

```
let hasSkeleton = skeleton?.jointMatrixPaletteBuffer != nil  
pipelineState =  
    PipelineStates.createForwardPSO(hasSkeleton: hasSkeleton)
```

When you load the mesh, you'll create a pipeline state with the appropriate vertex function.

- Open `Model.swift`, and in `render(encoder:uniforms:params:)`, at the top of the `for mesh in meshes` loop, add this:

```
encoder.setRenderPipelineState(mesh.pipelineState)
```

Each time you render a model, you'll load the appropriate pipeline state object. As long as you do the creation of the pipeline states at the start of your app, they are lightweight to swap in and out.

Now for the GPU side!

- Open **Shaders.metal**, and add this code after the `import` statements:

```
constant bool hasSkeleton [[function_constant(0)]];
```

The function constant index matches the constant you just created in the `MTLFunctionConstantValues` set.

- In the `vertex_main` header, change the `jointMatrices` parameter to:

```
constant float4x4 *jointMatrices [[  
    buffer(JointBuffer),  
    function_constant(hasSkeleton)]]
```

You'll have two different `vertex_mains` in your Metal shader library, one for each condition of `hasSkeleton`. One `vertex_main` will have `jointMatrices` as a parameter, if `hasSkeleton` is `true`, and the other won't have that parameter at all.

- In `vertex_main`, remove:

```
bool hasSkeleton = true;
```

You use the pipeline constant in place of the local one.

Your animation may glitch because of synchronization issues. You'll find out how to optimize your CPU / GPU synchronization in Chapter 26, “GPU-Driven Rendering”.

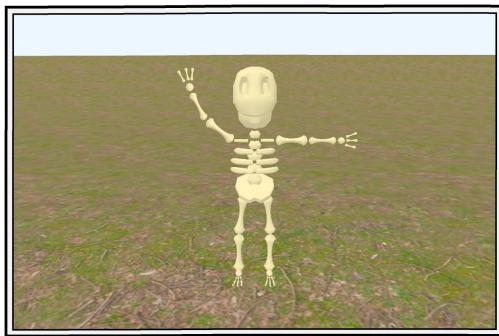
- For the moment, open **Renderer.swift**, and at the end of `draw(scene:in:)`, add this:

```
commandBuffer.waitUntilCompleted()
```

The thread will be blocked until the command buffer has finished executing all its commands.



- Build and run the app, and you'll see your full scene of animated skeleton and static ground.



## Key Points

- Character animation differs from transform animation. With transform animation, you deform the mesh directly. When animating characters, you use a skeleton with joints. The geometry mesh is attached to these joints and deforms when you rotate a joint.
- The skeleton consists of a hierarchy of joints. When you rotate one joint, all the child joints move appropriately.
- You attach the mesh to joints by weight painting in a 3D app. Up to four joints can influence each vertex (this is a limitation in your app, but generally weighting four joints is ample).
- Animation clips contain transformation data for keyframes. The app interpolates the transformations between keyframes.
- Each joint has an inverse bind matrix, which, when applied, moves the joint to the origin.
- When your shaders have different requirements depending on different situations, you can use function specialization. You indicate the different requirements in the pipeline state, and the compiler creates multiple versions of the shader function.

## Where to Go From Here?

This chapter took you through the basics of character animation. But don't stop there! There are so many different topics that you can investigate. For instance, you can:

- Learn how to animate your own characters in Blender and import them into your renderer. Start off with a simple robot arm, and work upward from there.
- Download models from <http://sketchfab.com>, convert them to USD and see what works and what doesn't.
- Watch Disney and Pixar movies... call it research. No, seriously! Animation is a skill all of its own. Watch how people move; good animators can capture personality in a simple walk cycle.



# Chapter 25: Managing Resources

So far, you've created an engine where you can load complex models with textures and materials, animate or update them per frame and render them. Your scenes will start to get more and more complicated as you develop your game, and you'll want to find more performant ways of doing things and organizing your game resources.

Instead of processing each submesh and laboriously moving each of the submesh's textures to the GPU, you'll take advantage of the centralization of your textures in the Texture Controller. By the end of the chapter, you'll be able to move all your textures to the GPU at once with just one render encoder command.

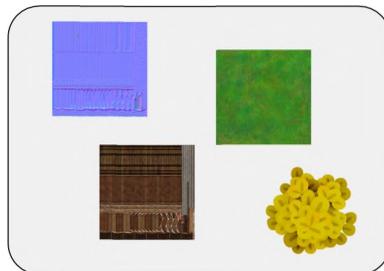
The secret sauce behind this process is indirection using **argument buffers** and a **texture heap**.

You'll learn more about these shortly, but in brief, an argument buffer represents data that can match a shader structure. You can send the argument buffer to a shader function with one command, instead of sending each of the structure components individually.



*An argument buffer containing resources*

A heap is exactly what it sounds like. You gather up your resources, such as textures and buffers, into an area of memory called a heap. You can then send this heap to the GPU with one command.



*A heap containing textures*

## The Starter Project

With the basic idea under your belt, you can now get started.

- In Xcode, open up the starter project for this chapter and build and run it.



You'll see medieval buildings with some skeletons roaming around menacingly.

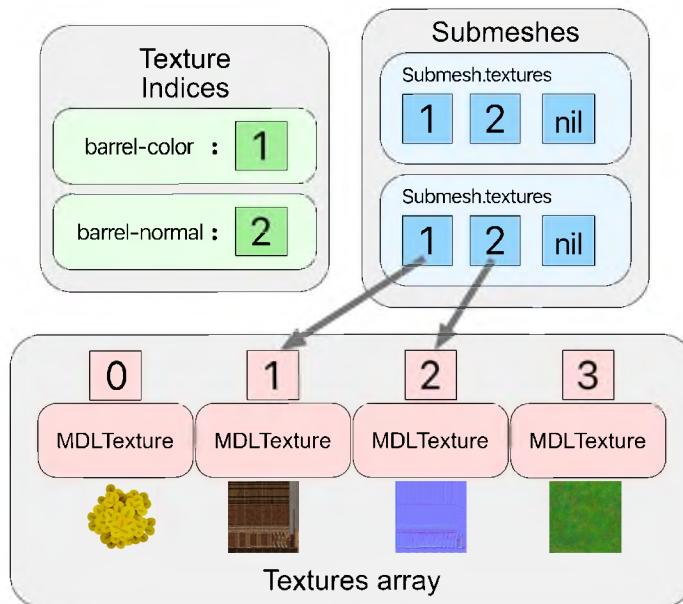
The project consolidates many of the features that you've learned so far:

- Shadows
- The PBR forward renderer

- Animation
- Alpha testing
- Textured models
- Models with materials but no textures

There are a couple of added nifty features.

Firstly, in the **Textures** group, in **TextureController.swift**, **TextureController** has an extra level of indirection. The old textures dictionary is now named **textureIndex** and it holds indices into an array of textures.



When you load a submesh texture using **TextureController**, if the texture doesn't exists by name already, **TextureController** adds the texture to the textures array, stores the array index and name into **textureIndices** and returns the index to the submesh. If the texture already exists by name, then the submesh simply holds the existing array index to the texture.

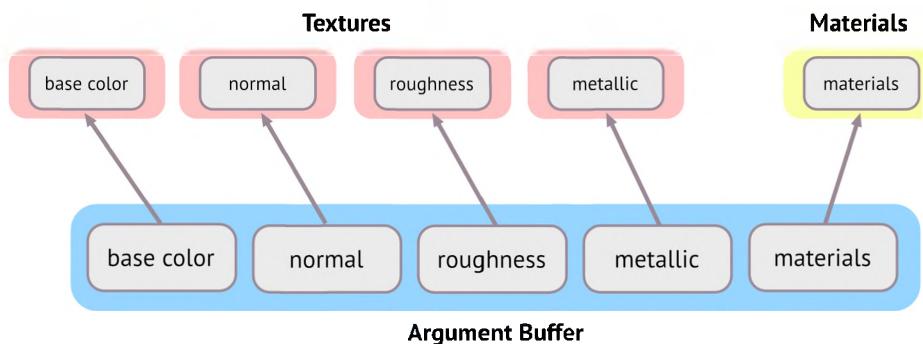
This saves on duplication of textures, and stores all the app textures in one central array, making it easier to process into a heap later.

Secondly, when setting up character joint animation, you used function constants when you defined the pipeline state for the vertex shader. This project also uses function constants for defining the shadow pipeline state.

In the **Render Passes** group, `ShadowRenderPass` and `ForwardRenderPass` sets a render pass state when rendering each model. The model then sets the correct mesh pipeline state depending on this render pass state, whether it is shadow or main,

## Argument Buffers

When rendering a submesh, you currently send up to six textures individually to the GPU for the fragment shader: Base color, normal, roughness, metalness, ambient occlusion and opacity textures. During the frame render loop, each of these incurs a `renderEncoder.setFragmentTexture(texture:at:)` command. Using **argument buffers**, you can group these six textures into one buffer, and set this buffer on the render command encoder with just one command. This argument buffer doesn't only have to point to textures, it can point to any other data necessary to render the frame.



When you come to draw time, instead of setting the textures on the render command encoder, you set the single argument buffer. You then perform `renderEncoder.useResource(_:usage:)` for each texture so that you can access all six textures on the GPU as reusable **indirect resources**.

Once you set up an argument buffer, you can refer to it in a shader, using one structure that matches the buffer data as a parameter to the shader function.

## Creating the Shader Structure

- In the **Shaders** group, open `PBR.metal`.

The `fragment_PBR` function has six parameters for material textures. You're going to combine all of these into one structure, and use the structure as the parameter.

- Create a new Header file in the **Shaders** group named `Material.h`.

- Before `#endif`, create a new structure:

```
struct ShaderMaterial {
    texture2d<float> baseColorTexture [[id(BaseColor)]];
    texture2d<float> normalTexture [[id(NormalTexture)]];
    texture2d<float> roughnessTexture [[id(RoughnessTexture)]];
    texture2d<float> metallicTexture [[id(MetallicTexture)]];
    texture2d<float> aoTexture [[id(A0Texture)]];
    texture2d<float> opacityTexture [[id(OpacityTexture)]];
    Material material [[id(OpacityTexture + 1)]];
};
```

Each argument buffer structure element has an implicit ID. For example, `baseColorTexture` has an implicit ID of `0`. If you want to use an out of order ID index, you can assign an explicit ID with an attribute, for example: `[[id(BaseColor)]]`.

Soon, you'll create an argument buffer that matches these IDs. You'll pass `in material` as a constant value. If you were to create an `MTLBuffer` containing `material`, you can define it in `ShaderMaterial` as: `constant Material &material;`.

- Open `PBR.metal`, and add this with the other file imports:

```
#import "Material.h"
```

- Change the header for `fragment_PBR` to:

```
fragment float4 fragment_PBR(
    FragmentIn in [[stage_in]],
    constant Params &params [[buffer(ParamsBuffer)]],
    constant Light *lights [[buffer(LightBuffer)]],
    constant ShaderMaterial &shaderMaterial
    [[buffer(MaterialBuffer)]],
    depth2d<float> shadowTexture [[texture(ShadowTexture)]])
```

Instead of receiving the textures directly, `fragment_PBR` will receive just one structure containing all the textures and material needed for the model.

- Replace:

```
Material material = _material;
```

- With:

```
Material material = shaderMaterial.material;
texture2d<float> baseColorTexture =
```

```
shaderMaterial.baseColorTexture;
texture2d<float> normalTexture = shaderMaterial.normalTexture;
texture2d<float> metallicTexture =
    shaderMaterial.metallicTexture;
texture2d<float> roughnessTexture =
    shaderMaterial.roughnessTexture;
texture2d<float> aoTexture = shaderMaterial.aoTexture;
texture2d<float> opacityTexture = shaderMaterial.opacityTexture;
```

You remove all the compile errors with these assignments.

For the moment you've finished setting up the GPU shader.

## Creating the Argument Buffer

To pass these textures, you create an **argument buffer** that matches the shader structure.

- In the **Geometry** group, open **Submesh.swift**, and add the new argument buffer property to Submesh:

```
var materialsBuffer: MTLBuffer!
```

`materialsBuffer` will contain pointers to the textures and the material.

- Create a new method in Submesh:

```
mutating func initializeMaterials() {
    guard let fragment =
        Renderer.library.makeFunction(name: "fragment_PBR") else {
            fatalError("Fragment function does not exist")
    }
    let materialEncoder = fragment.makeArgumentEncoder(
        bufferIndex: MaterialBuffer.index)
    materialsBuffer = Renderer.device.makeBuffer(
        length: materialEncoder.encodedLength,
        options: [])
}
```

You create an argument encoder from an existing fragment function.

`MTLFunction.makeArgumentEncoder(bufferIndex:)` looks at the given buffer index and identifies the structure that the fragment function requires. In this case, the `fragment_PBR` function requires `ShaderMaterial` at index `MaterialBuffer.index` (14). The encoder can work out how many elements there are in the structure, and therefore what size `materialsBuffer` should be.



- Add this after the previous code:

```
// 1
materialEncoder.setArgumentBuffer(materialsBuffer, offset: 0)
// 2
let range = Range(BaseColor.index...OpacityTexture.index)
materialEncoder.setTextures(allTextures, range: range)
// 3
let index = OpacityTexture.index + 1
let address = materialEncoder.constantData(at: index)
address.copyMemory(
    from: &material,
    byteCount: MemoryLayout<Material>.stride)
```

Going through this code:

1. The encoder will write the argument data into `materialsBuffer`.
2. The encoder sets all the submesh textures into the argument buffer. Each texture and buffer you set into the argument buffer should have its own unique index that matches the implicit or explicit ID in the shader file.
3. To set the constant material, you first identify the address in the argument buffer. You then copy `material` to that address in the buffer. If you were using a material MTLBuffer, you could use `MTLArgumentEncoder.setBuffer(_:_:offset:)` instead.

During the render loop, setting textures and buffers on the render command encoder incurs some internal verification. This verifying process will now take place here, when the textures and buffers are initially set into the argument buffer. Anything you can move outside of the render loop is a gain.

- Add the following code to the end of `init(mdlSubmesh:mtkSubmesh:)`:

```
initializeMaterials()
```

You've now set up your argument buffer. Instead of sending the textures and material to the fragment shader during the render loop, you'll send this single argument buffer.

## Updating the Draw Call

- In the `Geometry` group, open `Model.swift`.
- In `render(encoder:uniforms:params:renderState:)`, in the `for submesh in mesh.submeshes` loop, locate `updateFragmentMaterials(encoder:submesh:)`.



`updateFragmentMaterials(encoder:submesh:)` encodes all the textures and materials to the fragment function. As you're now using one buffer for all these textures and materials, this method is no longer necessary.

► Replace:

```
updateFragmentMaterials(  
    encoder: encoder,  
    submesh: submesh)
```

► With:

```
encoder.setFragmentBuffer(  
    submesh.materialsBuffer,  
    offset: 0,  
    index: MaterialBuffer.index)
```

Instead of encoding all the textures, you simply send the single argument buffer to the GPU.

If you were to build and run now, you may get a lot of GPU errors. Even if the render appears correct, if you capture the GPU workload, you may still get errors. When you have GPU memory errors, weird things can happen on the display. Debugging these errors can be frustrating as your display may lock up because you have accessed memory that you're not supposed to.

You've set up a level of indirection with the argument buffer pointing to the textures, but you still have to tell the GPU to *load* these textures. When dealing with indirection and buffer data, it's often easy to omit this vital step, so if you have errors at any time, check in the GPU debugger that the resource is available in the indirect resource list, but also check that you are **using** the resource in the render command encoder command list.

► Still inside the conditional `if renderState != .shadowPass`, but after the previous code, add this:

```
submesh.allTextures.forEach { texture in  
    if let texture = texture {  
        encoder.useResource(texture, usage: .read)  
    }  
}
```

Here, you tell the GPU that you're going to read from these textures, and they should be resident on the GPU.



- Build and run the app, and the scene will render as before.



- Capture the GPU workload.
- In the Debug navigator, open **Command Buffer** and **Forward Render Pass**.
- Under **large\_plane.obj**, select **drawIndexedPrimitives**.
- Select **Bound Resources** using the navigator icon at the top left of the pane and examine the resources.

Screenshot of the Xcode Debug Navigator showing the GPU workload for the "large\_plane.obj" drawIndexedPrimitives command.

Label	Type	Size	Details	Parameter Name	Resource
MDL_OBJ-Indices	Index	24 bytes	Offset: 0x0		
Buffer 0x141f4cc70	Buffer 0	224 bytes	Offset: 0x0	vertexBuffer.0	Read
Buffer 0x141f4cd00	Buffer 1	32 bytes	Offset: 0x0	vertexBuffer.1	Read
Buffer 0x141f4cf30	Buffer 2	64 bytes	Offset: 0x0	vertexBuffer.2	Read
Buffer 0x141f4d090	Buffer 3	64 bytes	Offset: 0x0	vertexBuffer.3	Read
Buffer 0x141f4d110	Buffer 4	64 bytes	Offset: 0x0	vertexBuffer.4	Read
Vertex Bytes	Buffer 11 (Bytes)	368 bytes		uniforms	Read
Geometry	Post Vertex Tran...				
Vertex Attributes	Vertex Attributes				
vertex_main	Vertex Function		Library 0x600001288740...		

**Fragment**

Shadow Depth Texture	Texture 10	4098 x 4096	Depth32Float	shadowTexture	Read
Fragment Bytes	Buffer 12 (Bytes)	48 bytes		params	Read
Buffer 0x141f4a660	Buffer 13	224 bytes	Offset: 0x0	lights	Read
Buffer 0x141f4af80	Buffer 14	112 bytes	Offset: 0x0	shaderMaterial	Read
fragment_PBR	Fragment Function		Library 0x600001288740...		

**Attachments**

CAMetalLayer Display Dra...	Color 0	1190 x 812	BGRA8Unorm		Write
MTKView Depth	Depth	1190 x 812	Depth32Float		Write

**Indirect Resources**

plane -color	Texture 0	512 x 512	RGBA8Unorm		Read
--------------	-----------	-----------	------------	--	------

For the ground plane, the Indirect Resources section lists the color texture. `MTLRenderCommandEncoder.useResource(_:_usage:)` explicitly makes the texture accessible to the GPU as an indirect resource.

- Under **Fragment**, double-click the `shaderMaterial` to examine it.

Row	Offset	Texture	Texture
		baseColorTexture	normalTexture
0	0x0	 plane-color	 Not a valid texture

Here, you can examine the textures and material properties in `shaderMaterial`.

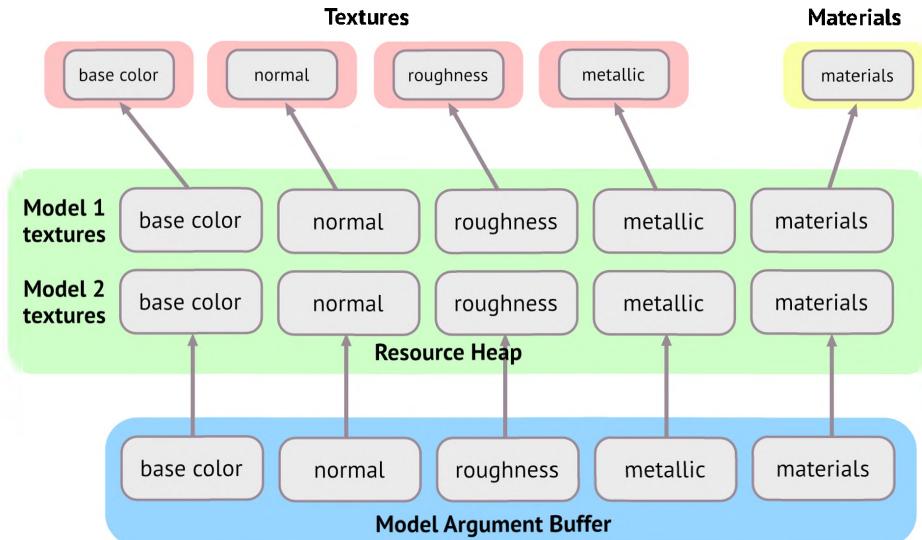
The arrow next to `plane-color` indicates the indirection. If you click the arrow, you'll see the plane's color texture.

All the textures and material correspond to how you encoded them in the argument buffer when you set up `Submesh`. As you can tell, it's very important to ensure that the argument buffer corresponds to the structure that you set up in the fragment shader.

You've now set up your app to use argument buffers for textures and the material instead of sending them individually. This may not feel like a win yet, and you've increased overhead by adding a new buffer. But you've reduced overhead on the render command encoder. Instead of having to validate the textures each frame, the textures are validated when they are first placed into the argument buffer, while you're still initializing your app data. In addition to this, you're grouping your materials together into the one structure, and only using one argument table entry in the fragment function. If you have many parameters that you can group together, this will save resources.

# Resource Heaps

You've grouped textures into an argument buffer for each submesh, but you can also combine all your app's textures into a resource **heap**.



A resource heap is simply an area of memory where you bundle resources. These can be textures or data buffers. To make your textures available on the GPU, instead of having to perform `renderEncoder.useResource(_:_usage:)` for every single texture, you can perform `renderEncoder.useHeap(_:_)` once per frame instead. That's one step further in the quest for reducing render commands.

- In the **Textures** group, open **TextureController.swift**.

**TextureController** stores all your app's textures in one central array: `textures`. From this array, you'll gather all the textures into a heap and move the whole heap at one time to the GPU.

- In **TextureController**, create a new property:

```
static var heap: MTLHeap?
```

- Create a new type method to build the heap:

```
static func buildHeap() -> MTLHeap? {
    let heapDescriptor = MTLHeapDescriptor()

    // add code here

    guard let heap =
        Renderer.device.makeHeap(descriptor: heapDescriptor)
        else { return nil }
    return heap
}
```

`MTLDevice.makeHeap(descriptor:)` is a time-consuming operation, so make sure that you execute it at loading time, rather than when your app is in full swing. Once you've created the heap, it's fast to add Metal buffers and textures to it.

You build a heap from a heap descriptor. This descriptor will need to know the size of all the textures combined. Unfortunately `MTLTexture` doesn't hold that information, but you can retrieve the size of a texture from a texture descriptor.

In the **Utility** group, in **Extensions.swift**, there's an extension on `MTLTexture` that will provide a descriptor from the texture.

- In **TextureController.swift**, in `buildHeap()`, replace `// add code here` with:

```
let descriptors = textures.map { texture in
    texture.descriptor
}
```

Here, you create an array of texture descriptors to match the array of textures. Now you can add up the size of all these descriptors.

- Following on from the previous code, add this:

```
let sizeAndAligns = descriptors.map { descriptor in
    Renderer.device.heapTextureSizeAndAlign(descriptor:
descriptor)
}
heapDescriptor.size = sizeAndAligns.reduce(0) { total,
sizeAndAlign in
    let size = sizeAndAlign.size
    let align = sizeAndAlign.align
    return total + size - (size & (align - 1)) + align
}
if heapDescriptor.size == 0 {
    return nil
}
```



You calculate the size of the heap using size and correct alignment within the heap. As long as align is a power of two, `(size & (align - 1))` will give you the remainder when size is divided by alignment. For example, if you have a size of 129 bytes, and you want to align it to memory blocks of 128 bytes, this is the result of `size - (size & (align - 1)) + align`:

$$129 - (129 \& (128 - 1)) + 128 = 256$$

This result shows that if you want to align blocks to 128, you'll need a 256 byte block to fit 129 bytes.

You have an empty heap, but you need to populate it with textures. Each texture must match the heap's CPU cache mode and also the heap's storage mode.

- At the end of `buildHeap()`, but before `return`, add this:

```
let heapTextures = descriptors.map { descriptor -> MTLTexture in
    descriptor.storageMode = heapDescriptor.storageMode
    descriptor.cpuCacheMode = heapDescriptor.cpuCacheMode
    guard let texture = heap.makeTexture(descriptor: descriptor)
    else {
        fatalError("Failed to create heap textures")
    }
    return texture
}
```

You iterate through the descriptors array and create a texture for each descriptor. You store this new texture in `heapTextures`.

`heapTextures` now contains a bunch of empty texture resources. To copy the submesh texture information to the heap texture resources, you'll need a **blit command encoder**.

## The Blit Command Encoder

To **blit** means to copy from one part of memory to another, and is typically an extremely fast operation. You create a blit command encoder using a command buffer, just as you did the render and compute command encoders. You then use this encoder when you want to copy a resource such as a texture or Metal buffer.

- Add this after the previous code:

```
guard
    let commandBuffer = Renderer.commandQueue.makeCommandBuffer(),
    let blitEncoder = commandBuffer.makeBlitCommandEncoder()
else { return nil }
```



```
zip(textures, heapTextures)
    .forEach { texture, heapTexture in
        heapTexture.label = texture.label
        // blit here
    }
```

You create the blit command encoder using a command buffer. You then set up a `forEach` loop that will process all the textures and match them with the heap textures.

► Replace `// blit here` with:

```
var region =
    MTLRegionMake2D(0, 0, texture.width, texture.height)
for level in 0..texture.mipmapLevelCount {
    for slice in 0..texture.arrayLength {
        blitEncoder.copy(
            from: texture,
            sourceSlice: slice,
            sourceLevel: level,
            sourceOrigin: region.origin,
            sourceSize: region.size,
            to: heapTexture,
            destinationSlice: slice,
            destinationLevel: level,
            destinationOrigin: region.origin)
    }
    region.size.width /= 2
    region.size.height /= 2
}
```

When copying textures, you specify a region. Initially the region will be the entire texture's width and height. You'll then blit mip levels where the region will get progressively smaller.

You copy each texture to a heap texture. Within each texture, you copy each level and slice. Levels contain the texture mipmaps, which is why you halve the region each loop. A slice is either the index into a texture array, or, for a cube texture, one of six cube faces.

Even though there are a lot of parameters to the blit encoder `copy` method, they are simply for deciding which area of the texture is to be copied. You can copy part of a texture by setting the origin and source size of the region. You can also copy part of a texture to a different region in the destination texture.

► Before return, add the following:

```
blitEncoder.endEncoding()
```

```
commandBuffer.commit()
Self.textures = heapTextures
```

This ends the encoding, commits the command buffer and replaces the original textures with the heap textures. `TextureController.buildHeap()` will now create a heap from all the textures gathered during scene loading.

► Open `Submesh.swift`.

In `initializeMaterials()`, you create an argument buffer from the submesh textures when you load the submesh. Unfortunately, as you've now copied all the old textures to new heap textures, your submesh argument buffers point to the wrong textures now.

► At the end of `init(mdlSubmesh:mtkSubmesh:)`, remove:

```
initializeMaterials()
```

You'll need to create the argument buffers after you've created the heap.

► In the `Game` group, open `Renderer.swift`, and add a new method to `Renderer`:

```
func initialize(_ scene: GameScene) {
    TextureController.heap = TextureController.buildHeap()
    for model in scene.models {
        model.meshes = model.meshes.map { mesh in
            var mesh = mesh
            mesh.submeshes = mesh.submeshes.map { submesh in
                var submesh = submesh
                submesh.initializeMaterials()
                return submesh
            }
            return mesh
        }
    }
}
```

`initialize(_:_)` will ensure that the heap gets built before the main render loop. You then process all the submeshes and initialize the materials with the correct textures.

► Open `GameController.swift`, and in `init(metalView:options:)`, after `scene = GameScene()`, add this:

```
renderer.initialize(scene)
```



- Build and run the app to ensure that everything still works.



*Skeletons on parade*

You've now placed all your textures in a heap, and are using those individual textures, but aren't yet taking full advantage of the heap. Before rendering any models, you can send the textures to the GPU at the start of a render pass to be all ready and waiting for processing.

- In the **Render Passes** group, open **ForwardRenderPass.swift**.

- In `draw(commandBuffer:scene:uniforms:params:)`, add this after creating `renderEncoder`:

```
if let heap = TextureController.heap {  
    renderEncoder.useHeap(heap)  
}
```

- Open **Model.swift**, and in `render(encoder:uniforms:params:renderState:)`, remove:

```
submesh.allTextures.forEach { texture in  
    if let texture = texture {  
        encoder.useResource(texture, usage: .read)  
    }  
}
```

Instead of having a `useResource` command for every texture, you perform one `useHeap` every render pass. This could be a huge saving on the number of commands in a render command encoder, and so a reduction of the number of commands that a GPU has to process each frame.

- Build and run the app, and your render should be exactly the same as it was.



- Capture the GPU workload.

- Open Command Buffer > Forward Render Pass and select the useHeap command that you set at the start of the render pass.

In the bound resources, all the scene textures are listed under indirect resources, and are available for use in any shader during this render pass.

 A screenshot of the Xcode Instruments tool showing captured GPU workload. The left sidebar shows a tree view with nodes like "Resources Captured GPU Workload", "Summary", "Performance", "Memory" (422.6 MB), and "Command Buffer". The "Command Buffer" node has a sub-node "Forward Render Pass" which is highlighted with a red box. The main pane shows a table of attachments. A section titled "Indirect Resources" is highlighted with a red box and contains 14 entries, each with a texture name, type, dimensions, and usage status (ReadSample or ReadWrite).
 

Label	Type	Size	Details	Par...	Resource Usag...
CAMetalLayer Display Drawab...	Color 0	1190 x 812	BGRAB8Unorm		Write
MTKView Depth	Depth	1190 x 812	Depth32Float		Write
790FDD37-E1B9-448A-...	Texture 0	1024 x 1024	RGBAB8Unorm		ReadWrite
6398B134-5B1A-4682-...	Texture 1	1024 x 1024	RGBAB8Unorm		ReadWrite
7E0D7506-E2A6-424D-...	Texture 2	1024 x 1024	RGBAB8Unorm		ReadWrite
03CDDAA8-6DF2-41D3-...	Texture 3	1024 x 1024	RGBAB8Unorm		ReadWrite
33B4586A-4791-41CF-8...	Texture 4	1024 x 1024	RGBAB8Unorm		ReadWrite
234CAA07-E2F6-46FB-8...	Texture 5	1024 x 1024	RGBAB8Unorm		ReadWrite
3F80F1B5-01DD-42CC-...	Texture 6	1024 x 1024	RGBAB8Unorm		ReadWrite
5E2F9197-68D5-4D11-A...	Texture 7	1024 x 1024	RGBAB8Unorm		ReadWrite
0E39380F-956A-49F2-B...	Texture 8	1024 x 1024	RGBAB8Unorm		ReadWrite
A21F7E4-D190-47B7-9...	Texture 9	1024 x 1024	RGBAB8Unorm		ReadWrite
2E108F4D-8564-4072-A...	Texture 10	1024 x 1024	RGBAB8Unorm		ReadWrite
3EB784A0-9EB1-4F61-A...	Texture 11	1024 x 1024	RGBAB8Unorm		ReadWrite
455D24FA-0E76-4F2A-9...	Texture 12	1024 x 1024	RGBAB8Unorm		ReadWrite
ECA2CA4-E434-43E8-...	Texture 13	1024 x 1024	RGBAB8Unorm		ReadWrite
688D191E-09A1-4407-B...	Texture 14	1024 x 1024	RGBAB8Unorm		ReadWrite

You've now separated out your textures from your rendering code, with a level of indirection via the argument buffer. But have you seen any performance improvement? In this example, on a recent device, probably not. But the more complicated your render passes get, the better the improvement, as there will be fewer render commands.

## Key Points

- An argument buffer is a collection of pointers to resources that you can pass to shaders.
- A resource heap is a collection of textures or Metal buffers. A heap can be static, as in this chapter's example, but you can also reuse space on the heap where you use different textures at different times.

# Chapter 26: GPU-Driven Rendering

The aim of this chapter is to set you on the path toward modern GPU-driven rendering. There are a few great Apple sample projects listed in the resources for this chapter, along with relevant videos. However, the samples can be quite intimidating. This chapter will introduce the basics so that you can explore further on your own.

The GPU requires a lot of information to be able to render a model. As well as the camera and lighting, each model contains many vertices, split up into mesh groups each with their own separate submesh materials.



*A house model with submeshes expanded*

The scene you'll render, in contrast, will only render two static models, each with one mesh and one submesh. Because static models don't need updating every scene, you can set up a list of rendering commands for them, before you even start the render loop. Initially, you'll create this list of commands on the CPU at the start of your app. Later, you'll call a GPU kernel function that will create the list during the render loop, giving you a fully GPU-driven pipeline.

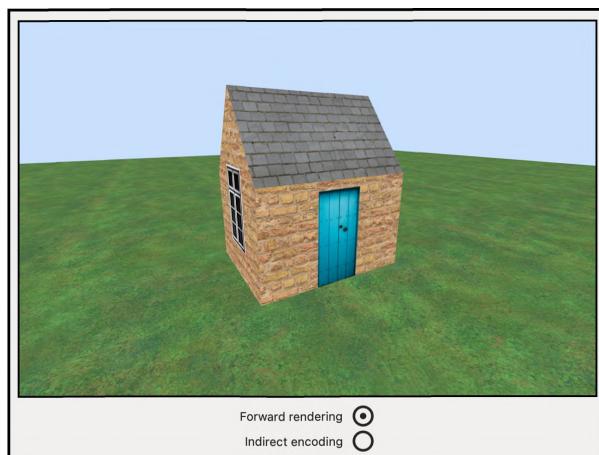
With this simple project, you may not see the immediate gains. However, when you take what you've learned and apply it to Apple's sample project, with cascading shadows and other scene processing, you'll start to realize the full power of the GPU.

You'll need recent hardware to run the code in this chapter. Techniques involved include:

- **Non-uniform threadgroups:** Supported on Apple Family GPU 4 and later (A11).
- **Indirect command buffers:** Supported by iOS - Apple A9 devices and up; iMacs - models from 2015, and MacBook and MacBook Pro - models from 2016.
- **Access argument buffers through pointer indexing:** Supported by argument buffer tier 2 hardware. This includes Apple GPU Family 6 and up (A13 and Silicon). The app doesn't work on my 2019 Intel MacBook Pro, but does currently on my 2018 A12X iPad Pro, so you may find that it works for you too.

## The Starter Project

- In Xcode, open the starter project, and build and run the app.



*The starter app*

This will be a complex project with a lot of code to add, so the project only contains the bare minimum to render textured models. All shadows, transparency and lighting has been removed.

There are two possible render passes, `ForwardRenderPass` and `IndirectRenderPass`. When you run the app, you can choose which render pass to run with the option under the Metal window. Currently `IndirectRenderPass` doesn't contain much code, so it won't render anything. `IndirectRenderPass.swift` is where you'll add most of the CPU code in this chapter. You'll change the GPU shader functions in `Shaders/Indirect.metal`.

- Open `ForwardRenderPass.swift`, and examine  
`draw(commandBuffer:scene:uniforms:params:)`.

Instead of rendering the model in `Model`, the rendering code is all here. You can see each render encoder command listed in this one method. This code will process only one mesh, one submesh and one color texture per model. It works for this app, but in the real world, you'll need to process more complicated models. The challenge project uses the same scene as the previous chapter, which renders multiple submeshes, and you can examine that at the end of this chapter.

## Indirect Command Buffers

In the previous chapter, you created argument buffers for your textures. These argument buffers point to textures in a texture heap.

Your rendering process currently looks like this:



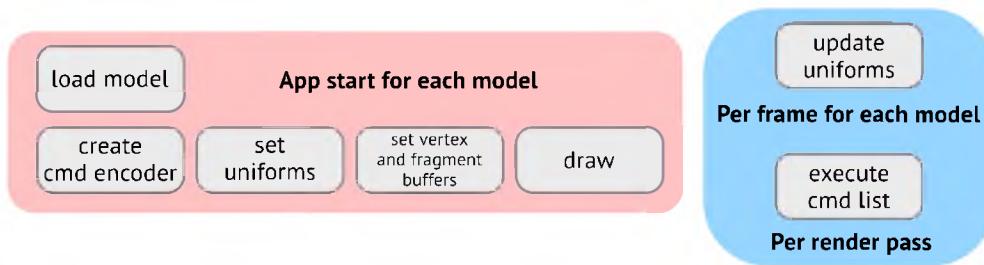
*Your render loop*

You load all the model data, materials and pipeline states at the start of the app. For each render pass, you create a render command encoder and issue commands one after another to that encoder, ending with a draw call. You repeat the drawing process for each model.



Instead of creating these commands per render pass, you can create them all at the start of the app using an **indirect command buffer** with a list of commands. You'll set up each command with pointers to the relevant uniform, material and vertex buffers and specify how to do the draw. During the render loop, you can just issue one execute command to the render command encoder, and the encoder will send the list of commands, all at once, off to the GPU.

Your rendering process will then look like this:



### *Indirect rendering*

Remember that your aim is to do as much as you can when your app first loads, and as little as you have to per frame. To achieve this, you'll:

1. Place all your uniform data in buffers. Because the indirect commands need to point to buffers at the start of the app, you can't send ad hoc bytes to the GPU. You can still update the buffers each frame. You'll set up a model matrix for each model in an array and then place this array into a Metal buffer. The models are static, so in this case, you won't need to update the buffer each frame.
2. Set up an indirect command buffer. This buffer will hold all the draw commands.
3. Loop through the models, setting up the indirect commands in the indirect command buffer.
4. Clean up the render loop and use the resources you referred to in the indirect commands to send them to the GPU.
5. Change the shader functions to use the array of model constants.
6. Execute the command list.



# 1. Initializing the Uniform Buffers

- In the **Render Passes** group, open **IndirectRenderPass.swift**.

**IndirectRenderPass** contains the minimum code to conform to **RenderPass**. It also contains a pipeline state that references the shader functions `vertex_indirect` and `fragment_indirect`. At the moment, these functions are duplicates of `vertex_main` and `fragment_main`.

- Add these new properties to **IndirectRenderPass**:

```
var uniformsBuffer: MTLBuffer!
var modelParamsBuffer: MTLBuffer!
```

You create a buffer that will hold the camera uniform data. `modelParamsBuffer` will hold the array of model matrices and tiling values.

- In the **Shaders** group, open **Common.h**, and create a new structure:

```
typedef struct {
    matrix_float4x4 modelMatrix;
    matrix_float3x3 normalMatrix;
    uint tiling;
} ModelParams;
```

You separate out the model information from the uniform data. The reason for doing this will become clearer as you progress through the chapter.

- Open **IndirectRenderPass.swift**, and add a new method to **IndirectRenderPass**:

```
mutating func initializeUniforms(_ models: [Model]) {
    let bufferLength = MemoryLayout<Uniforms>.stride
    uniformsBuffer =
        Renderer.device.makeBuffer(length: bufferLength, options: [])
    uniformsBuffer.label = "Uniforms"

    var modelParams: [ModelParams] = models.map { model in
        var modelParams = ModelParams()
        modelParams.modelMatrix = model.transform.modelMatrix
        modelParams.normalMatrix = modelParams.modelMatrix.upperLeft
        modelParams.tiling = model.tiling
        return modelParams
    }
    modelParamsBuffer = Renderer.device.makeBuffer(
        bytes: &modelParams,
        length: MemoryLayout<ModelParams>.stride * models.count,
        options: []
    )
}
```



```
    modelParamsBuffer.label = "Model Transforms Array"  
}
```

You set up the camera uniform and model transform data buffers.

Even though the models are static, you'll still have to update the camera uniforms every frame in case the user has changed the camera position.

► Add this new method to `IndirectRenderPass`:

```
func updateUniforms(scene: GameScene, uniforms: Uniforms) {  
    var uniforms = uniforms  
    uniformsBuffer.contents().copyMemory(  
        from: &uniforms,  
        byteCount: MemoryLayout<Uniforms>.stride)  
}
```

You load up the uniforms buffer with the current data for each frame.

► Call this method at the top of `draw(commandBuffer:scene:uniforms:params:)`:

```
updateUniforms(scene: scene, uniforms: uniforms)
```

► Create a new method in `IndirectRenderPass` to initialize the uniform buffers:

```
mutating func initialize(models: [Model]) {  
    initializeUniforms(models)  
}
```

Soon, you'll be initializing a few more buffers in this method.

Next, you need to call `initialize(models:)` from `Renderer`.

► In the `Game` group, open `Renderer.swift`, and add this to the end of `initialize(_:)`:

```
indirectRenderPass.initialize(models: scene.models)
```

## 2. Setting up an Indirect Command Buffer

You're now ready to create some indirect commands.

► Open `ForwardRenderPass.swift`, and look at `draw(commandBuffer:scene:uniforms:params:)`. Refresh your memory on all the render commands necessary to render the scene. You're going to move all these commands to an indirect command list.



- Open **IndirectRenderPass.swift**, and add a new property to **IndirectRenderPass**.

```
var icb: MTLIndirectCommandBuffer!
```

This buffer will hold the render command list.

- Create a new method in **IndirectRenderPass**:

```
mutating func initializeICBCommands(_ models: [Model]) {
    let icbDescriptor = MTLIndirectCommandBufferDescriptor()
    icbDescriptor.commandTypes = [.drawIndexed]
    icbDescriptor.inheritBuffers = false
    icbDescriptor.maxVertexBufferBindCount = 25
    icbDescriptor.maxFragmentBufferBindCount = 25
    icbDescriptor.inheritPipelineState = true
}
```

You create an Indirect Command Buffer descriptor. You specify that (eventually) the GPU should expect an indexed draw call. That's a draw call that uses an index buffer for indexing into the vertices. You set the maximum number of buffers that the ICB can bind to in the vertex and fragment shader parameters to 25. This is far too many, but you can renumber the buffer indices when your app is complete.

You set `inheritPipelineState` to `true`. Because this app contains such simple models, you can set the render pipeline state at the start of the render pass, and all encoder commands will inherit the current pipeline state. If you require a different pipeline for different submeshes, you'd set `inheritPipelineState` to `false` and add setting the render pipeline state to the list of indirect encoder commands.

- Following on from that code, create the indirect command buffer:

```
guard let icb = Renderer.device.makeIndirectCommandBuffer(
    descriptor: icbDescriptor,
    maxCommandCount: models.count,
    options: []) else { fatalError("Failed to create ICB") }
self.icb = icb
```

The ICB will need one command per draw call. In this app, you're only performing one draw call per model, but in a more complex app where you're doing a draw call for every submesh, you'd have to iterate through the models prior to setting up the ICB to find out how many draw calls you'll do.

## 3. Setting up the Indirect Commands

Now that you've set up an indirect command buffer, you'll add the list of commands to it.

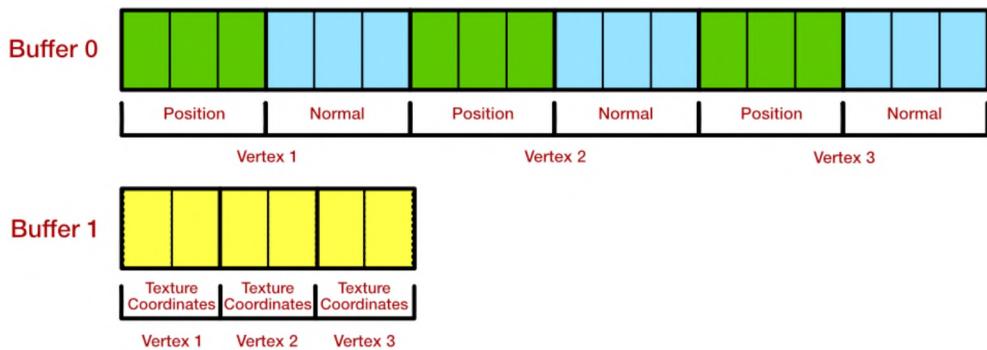
- Add the following code to the end of `initializeICBCommands(_:)`:

```
for (modelIndex, model) in models.enumerated() {  
    let mesh = model.meshes[0]  
    let submesh = mesh.submeshes[0]  
    let icbCommand = icb.indirectRenderCommandAt(modelIndex)  
    icbCommand.setVertexBuffer(  
        uniformsBuffer, offset: 0, at: UniformsBuffer.index)  
    icbCommand.setVertexBuffer(  
        modelParamsBuffer, offset: 0, at: ModelParamsBuffer.index)  
    icbCommand.setFragmentBuffer(  
        modelParamsBuffer, offset: 0, at: ModelParamsBuffer.index)  
    icbCommand.setVertexBuffer(  
        mesh.vertexBuffers[VertexBuffer.index],  
        offset: 0,  
        at: VertexBuffer.index)  
    icbCommand.setVertexBuffer(  
        mesh.vertexBuffers[UVBuffer.index],  
        offset: 0,  
        at: UVBuffer.index)  
    icbCommand.setFragmentBuffer(  
        submesh.argumentBuffer!, offset: 0, at:  
        MaterialBuffer.index)  
}
```

This code may look familiar to you from the render loop in `ForwardRenderPass.draw(commandBuffer:scene:uniforms:params:)`. You use the model index to keep track of the command list, and you set all the necessary data for each draw call.

It's important to remember how you formatted your vertex buffers when loading the models.

- In the **Geometry** group, open **VertexDescriptor.swift**, and examine how the vertex buffers are configured.



### Vertex buffer layouts

The vertex descriptor is simplified from the previous chapter, as there are no animated models to process.

- Open **IndirectRenderPass.swift**, add the draw call to the end of the `for` loop in `initializeICBCommands(_:)`:

```
icbCommand.drawIndexedPrimitives(
    .triangle,
    indexCount: submesh.indexCount,
    indexType: submesh.indexType,
    indexBuffer: submesh.indexBuffer,
    indexBufferOffset: submesh.indexBufferOffset,
    instanceCount: 1,
    baseVertex: 0,
    baseInstance: modelIndex)
```

This draw command is very similar to the one that you've already been using. There are a couple of extra arguments:

- **baseVertex:** The vertex in the vertex buffer to start rendering from.
- **baseInstance:** The instance to start rendering from. You have set up an array of `modelParams`, one for each model. Using `baseInstance`, in the vertex shader, you can index into the array to get the correct element.

The command list is now complete.

- Call this method at the end of `initialize(models:)`:

```
initializeICBCommands(models)
```

## 4. Updating the Render Loop

Currently none of your resources are making their way to the GPU.

- Create a new method in `IndirectRenderPass`:

```
func useResources(  
    encoder: MTLRenderCommandEncoder, models: [Model]  
) {  
    encoder.pushDebugGroup("Using resources")  
    encoder.useResource(uniformsBuffer, usage: .read)  
    encoder.useResource(modelParamsBuffer, usage: .read)  
    if let heap = TextureController.heap {  
        encoder.useHeap(heap)  
    }  
    for model in models {  
        let mesh = model.meshes[0]  
        let submesh = mesh.submeshes[0]  
        encoder.useResource(  
            mesh.vertexBuffers[VertexBuffer.index], usage: .read)  
        encoder.useResource(  
            mesh.vertexBuffers[UVBuffer.index], usage: .read)  
        encoder.useResource(  
            submesh.indexBuffer, usage: .read)  
        encoder.useResource(  
            submesh.argumentBuffer!, usage: .read)  
    }  
    encoder.popDebugGroup()  
}
```

When you `use` a resource, it's available to the GPU as indirect resource ready for the GPU to access.

- Add this to `draw(commandBuffer:scene:uniforms:params:)`, before `renderEncoder.endEncoding()`:

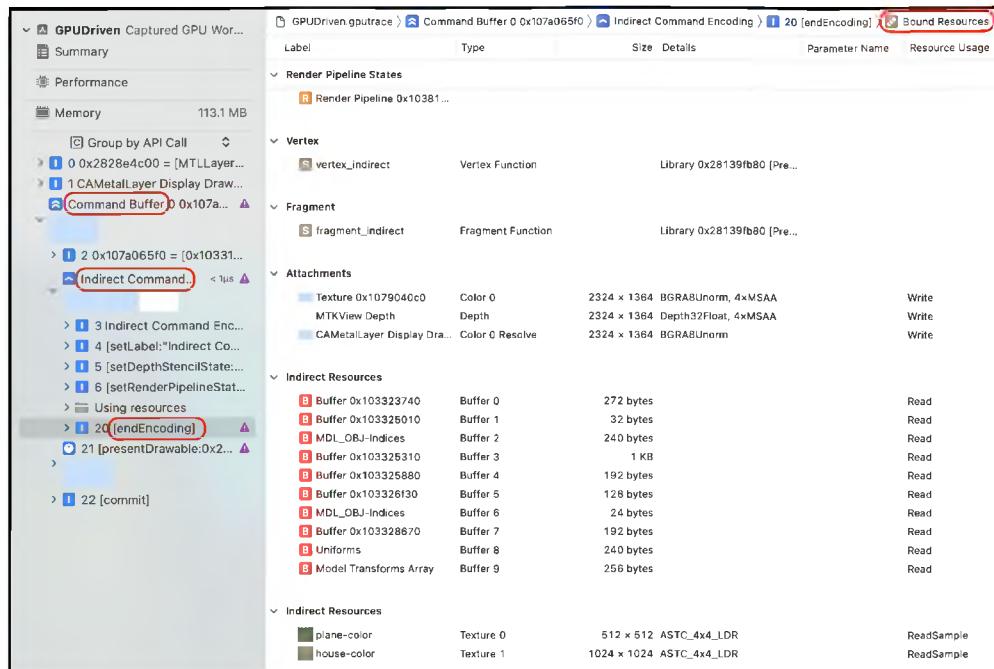
```
useResources(encoder: renderEncoder, models: scene.models)
```

- To ensure everything is working so far, build and run the app, switch to **Indirect encoding** and capture the GPU workload.

You're not yet executing any draw calls, so you'll have a clear blue screen.



- Expand the **Indirect Command Encoding** render pass, and select **[endEncoding]**.



### *Loaded indirect resources*

All of your resources are listed in the Indirect Resources bound to the GPU and are available to your vertex and fragment shaders.

## 5. Updating the Shader Functions

- In the **Shaders** group, open **Indirect.metal**.

This file is currently a duplicate of **Shaders.metal**. However, you set up your indirect commands to use an array of model transforms instead of holding each model's transform in **Uniforms**, so you'll change the vertex function to reflect that.

- Add new parameters to **vertex\_indirect**:

```
constant ModelParams *modelParams [[buffer(ModelParamsBuffer)]],  
uint modelIndex [[base_instance]]
```

**modelParams** is the array of model transforms. You'll extract the correct instance from the array using **modelIndex**, which is the same as **baseInstance** that you already set in the draw call.

- Add the following code to the top of `vertex_indirect`:

```
ModelParams model = modelParams[modelIndex];
```

- Where you calculate `.position`, replace `uniforms.modelMatrix` with:

```
model.modelMatrix
```

You use the appropriate matrix from the array of model transforms rather than the single `modelMatrix` value in `uniforms`.

`fragment_indirect` will require the model's tiling value, so you'll pass the correct instance to the fragment function.

- Add this property to `VertexOut`:

```
uint modelIndex [[flat]];
```

The `[[flat]]` attribute ensures that the value won't be interpolated between the vertex and fragment function.

- In `vertex_indirect`, add the model index to the `VertexOut` `out` assignment:

```
.modelIndex = modelIndex
```

- Just as you did in the vertex function, add the new parameter to `fragment_indirect`:

```
constant ModelParams *modelParams [[buffer(ModelParamsBuffer)]]
```

- Add this to the top of `fragment_indirect`:

```
ModelParams model = modelParams[in.modelIndex];
```

- In the conditional where you sample `baseColorTexture`, replace `params.tiling` with:

```
model.tiling
```



- Remove this parameter from `fragment_indirect`'s header:

```
constant Params &params [[buffer(ParamsBuffer)]],
```

That's fixed up the shader functions so that they'll read an array of parameters instead of a single instance.

## 6. Execute the Command List

All the code you have written in this chapter so far has been building up to one command.

Drum roll....

- Open `IndirectRenderPass.swift`, and add the following code to `draw(commandBuffer:scene:uniforms:params:)`, before `renderEncoder.endEncoding()`:

```
renderEncoder.executeCommandsInBuffer(  
    icb, range: 0..
```

This code will execute all the commands in the indirect command buffer's list within the range specified here. If you specify a range of `0..<1`, then only the first draw call would be performed.

- Build and run the app, and switch to **Indirect encoding**.

And... you get an error:

```
The indirect command buffer inherits pipelines  
( inheritPipelineState = YES) but the render pipeline set on  
this encoder does not support indirect command buffers  
( supportIndirectCommandBuffers = NO )
```

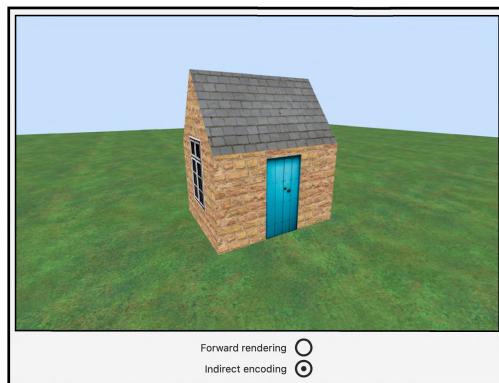
When you use a pipeline state in an indirect command list, you have to tell it that it should support indirect command buffers.

- Open `Pipelines.swift`, and add this to `createIndirectPSO()` before return:

```
pipelineDescriptor.supportIndirectCommandBuffers = true
```



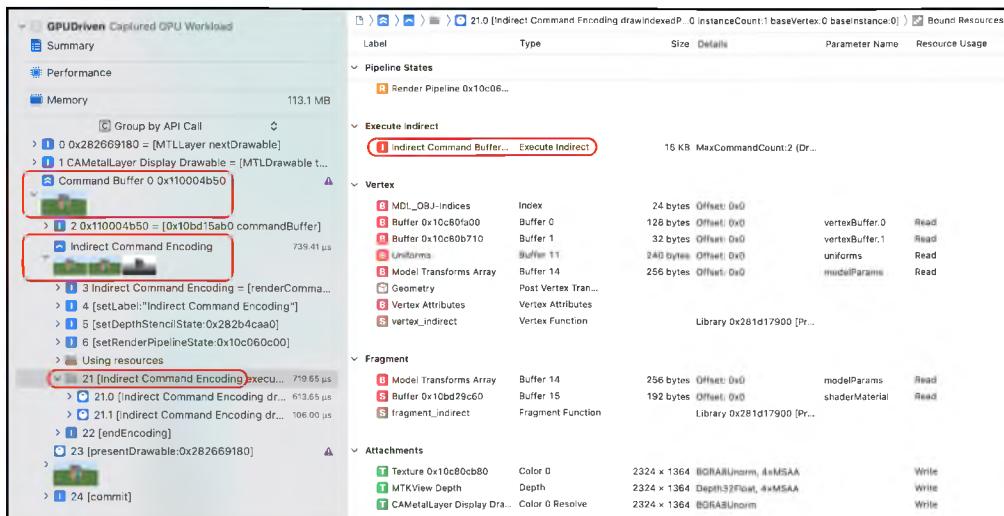
- Build and run the app, and switch to **Indirect encoding**.



*Indirect encoding*

This may not be the most exciting result — as both forward rendering and indirect encoding renders look exactly the same — but behind the scenes, it's a different story. Very little is happening in your render loop, and all the heavy lifting is done at the very start of the app. Success. :)

- Capture the GPU workload, and expand both the **Indirect Command Encoding** render pass and **Indirect Command Encoding**.



*Execute indirect commands*

- In the bound resources, double-click **Indirect Command Buffer**.

0 [drawIndexedPrimitives:Triangle indexCount:6 indexType:UInt32 indexBuffer:"MDL_OBJ-Indices" (0x14ae)]
B MDL_OBJ-Indices                  IndexBuffer                  24 bytes Offset: 0x0
B Buffer 0x14ae76790              Vertex 0                  128 bytes Offset: 0x0
B Buffer 0x14ae799f0              Vertex 1                  32 bytes Offset: 0x0
B Uniforms                         Vertex 11                240 bytes Offset: 0x0
B Model Transforms Array        Vertex 13                256 bytes Offset: 0x0
B Model Transforms Array        Fragment 13            256 bytes Offset: 0x0
B Buffer 0x14af3d360              Fragment 14            80 bytes Offset: 0x0
1 [drawIndexedPrimitives:Triangle indexCount:60 indexType:UInt32 indexBuffer:"MDL_OBJ-Indices" (0x14ae)]
B MDL_OBJ-Indices                  IndexBuffer                  240 bytes Offset: 0x0
B Buffer 0x14ae76bd0              Vertex 0                  1 KB Offset: 0x0
B Buffer 0x14ae74ef0              Vertex 1                  272 bytes Offset: 0x0
B Uniforms                         Vertex 11                240 bytes Offset: 0x0
B Model Transforms Array        Vertex 13                256 bytes Offset: 0x0
B Model Transforms Array        Fragment 13            256 bytes Offset: 0x0
B Buffer 0x14af3d4c0              Fragment 14            80 bytes Offset: 0x0

*The indirect command list*

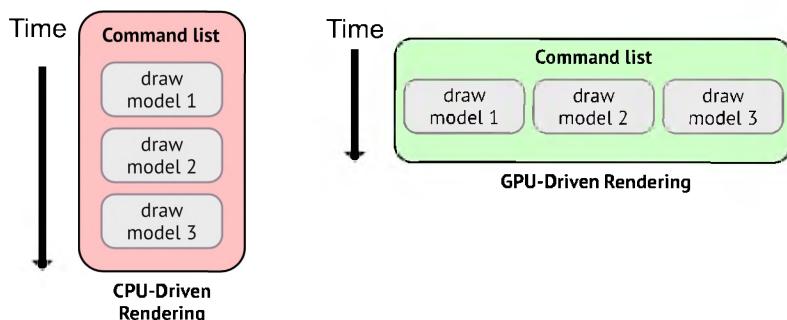
You can see both your draw call commands listed with their encoded resources.

## GPU-Driven Rendering

You've achieved indirect CPU rendering, by setting up a command list and rendering it. However, you can go one better and get the GPU to create this command list.

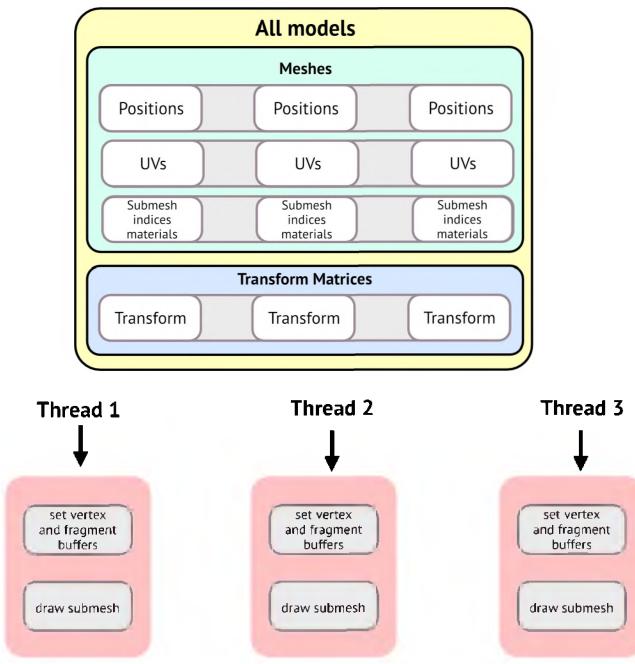
- Open **IndirectRenderPass.swift**, and look at the `for` loop in `initializeICBCommands(_:)`.

This `for` loop executes serially on the CPU, but is one that you can easily parallelize. Each ICB command executes one after another, but by moving this loop to the GPU, you can create each command at the same time over multiple GPU cores.



*GPU command creation*

When you come to write real-world apps, setting up the render loop at the very start of the app is impractical. In each frame, you'll be determining which models to render. Are the models in front of the camera? Is the model occluded by another model? Should you render a model with lower level of detail? By creating the command list every frame, you have complete flexibility in which models you should render, and which you should ignore. As you'll see, the GPU is amazingly fast at creating these render command lists, so you can include this process each frame.



### *Creating commands per thread*

You'll create a compute shader and pass it all the buffers that you used during the `initializeICBCommands(_:) for` loop:

- uniform and model parameter buffers
- the indirect command buffer

For the models' vertex buffers and materials, you'll create an array of argument buffers containing for each model:

- the vertex buffers
- the index buffer
- the submesh material argument buffer

There's one more array you'll need to send: the draw arguments for each model. Each model's draw call is different from every other. You have to specify, for example, what the index buffer is and what is the index count. Fortunately Apple have created a format that you can use for this, called `MTLDrawIndexedPrimitivesIndirectArguments`. That's some mouthful!

There's quite a lot of setup code, and you have to be careful when matching buffers with compute shader parameters. If you make an error, it's difficult to debug it, and your computer may lock up. Running the app on an external device, such as iPhone or iPad is preferable, if slightly slower.

These are the steps you'll take:

1. Create the kernel function.
2. Set up the compute pipeline state object.
3. Set up the argument buffers for the kernel function.
4. Set up the draw arguments.
5. Complete the compute command encoder.

## 1. Creating the Kernel Function

You'll start by creating the kernel function compute shader so that you can see what data you have to pass. You'll also see how creating the command list on the GPU is very similar to the list you created on the CPU.

► In the **Shaders** group, create a new Metal file named **ICB.metal**, and add the following:

```
#import "Common.h"

struct ICBContainer {
    command_buffer icb [[id(0)]];
};

struct Model {
    constant float *vertexBuffer;
    constant float *uvBuffer;
    constant uint *indexBuffer;
    constant float *materialBuffer;
};
```

These are the argument buffer structures you'll need:

1. The indirect command buffer container. On the Swift side, you'll create an argument buffer to hold the indirect command buffer. In the kernel function, you'll encode commands to this command buffer. `ICBContainer`, as suggested by its name, simply contains this command buffer.
2. You'll collect an array of model data that you'll send to the kernel function. For vertices, each element will hold the positions and normals in the vertex buffer, the UVs in the UV buffer and also the index buffer that indexes into the vertex buffers. For the fragment, you'll hold the submesh's material argument buffer.

You can add an explicit `[[id((n))]` attribute to each of the structure parameters. If you don't, the ID number is implicit, starting at zero. When you encode the argument buffers, you'll add each element in order, so in the `Model` structure, you don't need to specify the ID.

► Add a new kernel function to `ICB.metal`:

```
kernel void encodeCommands(
    // 1
    uint modelIndex [[thread_position_in_grid]],
    // 2
    device ICBContainer *icbContainer [[buffer(ICBBuffer}}],
    constant Uniforms &uniforms [[buffer(UniformsBuffer}}],
    // 3
    constant Model *models [[buffer(ModelsBuffer}}],
    constant ModelParams *modelParams
    [[buffer(ModelParamsBuffer}}],
    constant MTLDrawIndexedPrimitivesIndirectArguments
        *drawArgumentsBuffer [[buffer(DrawArgumentsBuffer)]])
{
}
```

`Common.h` contains the index numbers for these new buffer indices.

Going through these arguments:

1. You'll dispatch a thread for each model. `modelIndex` gives the thread position in the grid, and is the index into the arrays of `models` and `modelParams`.
2. The kernel receives the indirect command buffer container, and the uniforms. You'll encode the uniform data and send it to the vertex and fragment functions.

3. The `models` buffer will contain arrays of argument buffers. You'll retrieve the current model using `modelIndex`.
4. You'll formulate draw call arguments using the submesh data. You'll retrieve these arguments from `drawArgumentsBuffer` to create the draw call.

Now that you've set up all the data, it's an easy task to encode the draw call.

► Add this to `encodeCommands`:

```
// 1
Model model = models[modelIndex];
MTLDrawIndexedPrimitivesIndirectArguments drawArguments
    = drawArgumentsBuffer[modelIndex];
// 2
render_command cmd(icbContainer->icb, modelIndex);
// 3
cmd.set_vertex_buffer(&uniforms, UniformsBuffer);
cmd.set_vertex_buffer(model.vertexBuffer, VertexBuffer);
cmd.set_vertex_buffer(model.uvBuffer, UVBuffer);
cmd.set_vertex_buffer(modelParams, ModelParamsBuffer);
cmd.set_fragment_buffer(modelParams, ModelParamsBuffer);
cmd.set_fragment_buffer(model.materialBuffer, MaterialBuffer);
```

Going through the code:

1. You retrieve the model and draw arguments using the thread position in grid.
2. On the Swift side, when you set up the indirect command buffer, you'll indicate how many commands it should expect. You use `modelIndex` to point to the appropriate command.
3. Just as you would in the render loop, or as you did in the indirect command buffer earlier, you encode the data needed for the draw call.

Finally, you'll encode the draw call.

► Add this code to the end of `encodeCommands`:

```
cmd.draw_indexed_primitives(
    primitive_type::triangle,
    drawArguments.indexCount,
    model.indexBuffer + drawArguments.indexStart,
    drawArguments.instanceCount,
    drawArguments.baseVertex,
    drawArguments.baseInstance);
```

This looks very similar to the draw call you're accustomed to, with the primitive type and the index buffer details.



You've now encoded a complete draw call, and that's all that's required for the compute function. Your next task is to set up the compute function on the CPU side, with a compute pipeline state and pass all the data to the compute function.

**Note:** You're not performing any extra logic here to see whether the model should be rendered this frame. But if you determine that the model shouldn't be rendered, instead of doing a draw call, you'd create an empty command with `cmd.reset()`.

## 2. The Compute Pipeline State

- Open `IndirectRenderPass.swift`, and create these new properties in `IndirectRenderPass`:

```
let icbPipelineState: MTLComputePipelineState  
let icbComputeFunction: MTLFunction
```

You'll need a new compute pipeline state which uses the compute function you just created.

- Add the following code to the end of `init()`:

```
icbComputeFunction =  
    Renderer.library.makeFunction(name: "encodeCommands")!  
icbPipelineState = PipelineStates.createComputePSO(  
    function: "encodeCommands")
```

This code creates the compute function in the Metal library, and also the compute pipeline state.

## 3. Setting Up the Argument Buffers

The `encodeCommands` kernel function requires two structures as input: one for the ICB, and one for the model.

- In `IndirectRenderPass`, add two buffer properties for the argument buffers to match these structures:

```
var icbBuffer: MTLBuffer!  
var modelsBuffer: MTLBuffer!
```



- In `initializeICBCommands(_:)`, remove the entire `for` loop, so that the last command in the method is `self.icb = icb`.

You're going to be creating the commands on the GPU each frame from now on.

- Add this code to the end of `initializeICBCommands(_:)`:

```
let icbEncoder = icbComputeFunction.makeArgumentEncoder(
    bufferIndex: ICBBuffer.index)
icbBuffer = Renderer.device.makeBuffer(
    length: icbEncoder.encodedLength,
    options: [])
icbEncoder.setArgumentBuffer(icbBuffer, offset: 0)
icbEncoder.setIndirectCommandBuffer(icb, index: 0)
```

Just as you did in the previous chapter, you create an argument encoder to match the compute function parameter and assign an argument buffer that will contain the command list to the encoder. You also set the indirect command buffer in the argument buffer.

- Create a new method in `IndirectRenderPass` to fill the model array buffer:

```
mutating func initializeModels(_ models: [Model]) {
    // 1
    let encoder = icbComputeFunction.makeArgumentEncoder(
        bufferIndex: ModelsBuffer.index)
    // 2
    modelsBuffer = Renderer.device.makeBuffer(
        length: encoder.encodedLength * models.count, options: [])
    // 3
    for (index, model) in models.enumerated() {
        let mesh = model.meshes[0]
        let submesh = mesh.submeshes[0]
        encoder.setArgumentBuffer(
            modelsBuffer, startOffset: 0, arrayElement: index)
        encoder.setBuffer(
            mesh.vertexBuffers[VertexBuffer.index], offset: 0, index:
            0)
        encoder.setBuffer(
            mesh.vertexBuffers[UVBuffer.index],
            offset: 0,
            index: 1)
        encoder.setBuffer(
            submesh.indexBuffer,
            offset: submesh.indexBufferOffset,
            index: 2)
        encoder.setBuffer(submesh.argumentBuffer!, offset: 0, index:
            3)
    }
}
```

Going through this code:

1. You create an argument buffer encoder. You stored the `encodeCommands` function in `icbComputeFunction` so that the encoder can reference the function arguments and see how much space the argument buffer will need.
2. You create the argument buffer using the required length provided by the `encodeCommands` function, multiplied by the number of models you'll encode.
3. You iterate through the models and set the buffers on the argument encoder, specifying the index number to use for the element in the argument buffer array. These buffers match the properties in the compute shader for the `Model` structure.

► Add this to the end of `initialize(models:)`:

```
initializeModels(models)
```

## 4. Setting Up the Draw Arguments

The `encodeCommands` kernel function takes in an array of draw arguments that it uses for each draw call. You'll now set these up into a buffer.

► Create a new buffer property in `IndirectRenderPass` for the draw arguments:

```
var drawArgumentsBuffer: MTLBuffer!
```

► Add a new method to `IndirectRenderPass`:

```
mutating func initializeDrawArguments(models: [Model]) {  
    let drawLength = models.count *  
  
    MemoryLayout<MTLDrawIndexedPrimitivesIndirectArguments>.stride  
    drawArgumentsBuffer = Renderer.device.makeBuffer(  
        length: drawLength, options: [])  
    drawArgumentsBuffer.label = "Draw Arguments"  
    var drawPointer =  
        drawArgumentsBuffer.contents().bindMemory(  
            to: MTLDrawIndexedPrimitivesIndirectArguments.self,  
            capacity: models.count)  
}
```

You set up the buffer with the appropriate length. You also initialize a pointer so you can store the draw arguments in the buffer.



- Add the following code after the code you just added:

```
for (modelIndex, model) in models.enumerated() {
    let mesh = model.meshes[0]
    let submesh = mesh.submeshes[0]
    var drawArgument = MTLDrawIndexedPrimitivesIndirectArguments()
    drawArgument.indexCount = UInt32(submesh.indexCount)
    drawArgument.indexStart = UInt32(submesh.indexBufferOffset)
    drawArgument.instanceCount = 1
    drawArgument.baseVertex = 0
    drawArgument.baseInstance = UInt32(modelIndex)
    drawPointer.pointee = drawArgument
    drawPointer = drawPointer.advanced(by: 1)
}
```

Here, you iterate through the models adding a draw argument into the buffer for each model. Each property in `drawArgument` corresponds to a parameter in the final draw call.

- Call this method at the end of `initialize(models:)`:

```
initializeDrawArguments(models: models)
```

## 5. Completing the Compute Command Encoder

You've done all the preamble and setup code. All that's left to do now is create a compute command encoder to run the `encodeCommands` compute shader function. The function will create a render command to render every model.

- Still in `IndirectRenderPass.swift`, add the following code to `draw(commandBuffer:scene:uniforms:params:)`, after `updateUniforms(...)` but before creating `renderEncoder`:

```
guard
    let computeEncoder = commandBuffer.makeComputeCommandEncoder()
    else { return }
encodeDraw(encoder: computeEncoder)
useResources(encoder: computeEncoder, models: scene.models)
dispatchThreads(
    encoder: computeEncoder, drawCount: scene.models.count)
computeEncoder.endEncoding()
```

You'll write a method to encode the draw and ready all the buffers to send to the GPU. You'll then ensure that the buffers are loaded to the GPU by using the resources. Finally, you'll dispatch the threads to the compute kernel.

- Add the first method to `IndirectRenderPass`:

```
func encodeDraw(encoder: MTLComputeCommandEncoder) {
    encoder.setComputePipelineState(icbPipelineState)
    encoder.setBuffer(
        icbBuffer, offset: 0, index: ICBBuffer.index)
    encoder.setBuffer(
        uniformsBuffer, offset: 0, index: UniformsBuffer.index)
    encoder.setBuffer(
        modelsBuffer, offset: 0, index: ModelsBuffer.index)
    encoder.setBuffer(
        modelParamsBuffer, offset: 0, index:
        ModelParamsBuffer.index)
    encoder.setBuffer(
        drawArgumentsBuffer, offset: 0, index:
        DrawArgumentsBuffer.index)
}
```

Here, you set all the buffers in one go. Generally you loop through the models and set each vertex buffer individually. As these are mostly argument buffers, the resources have already been verified that they're suitable for sending to the GPU.

- Change the header for `useResources(encoder:models:)` to:

```
func useResources(
    encoder: MTLComputeCommandEncoder, models: [Model]
) {
```

Now that you're encoding the resources on the GPU, you encode them with the compute command encoder.

- Add this to `useResources(encoder:models:)` after `encoder.pushDebugGroup("...")`:

```
encoder.useResource(icb, usage: .write)
```

Just as in the previous chapter, you must **use** all the resources that argument buffers point to, to ensure they are installed to the GPU. You can spend a lot of time looking at pink or black renders when you forget to transfer a resource, so ensure that you're using all the resources that the GPU looks for.

You set the usage of the indirect command buffer to `write`, as this is where the `encodeCommands` kernel function will write the commands.

- Because you no longer need to use the resources in the render loop, remove the following line from the end of `draw(commandBuffer:scene:uniforms:params:)`:

```
useResources(encoder: renderEncoder, models: scene.models)
```

- To remove the last compiler error, add the following method to `IndirectRenderPass`:

```
func dispatchThreads(  
    encoder: MTLComputeCommandEncoder,  
    drawCount: Int  
) {  
    let threadExecutionWidth =  
        icbPipelineState.threadExecutionWidth  
    let threads = MTLSIZE(width: drawCount, height: 1, depth: 1)  
    let threadsPerThreadgroup = MTLSIZE(  
        width: threadExecutionWidth, height: 1, depth: 1)  
    encoder.dispatchThreads(  
        threads,  
        threadsPerThreadgroup: threadsPerThreadgroup)  
}
```

You tell the compute command encoder how many threads to create, and dispatch them off to the GPU. The `encodeCommands` kernel function will set up all the render commands in parallel.

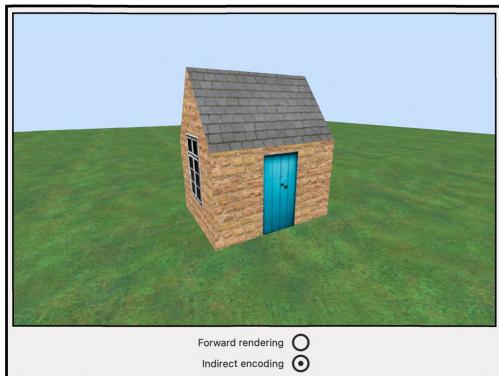
- Before you build and run the app, save all of the documents you have open.

When you're programming GPUs and moving around blocks of memory, sometimes you can accidentally set memory blocks in areas where you're not supposed to. When this happens, your display may go crazy with flickering and drawing weirdness, and you'll have to restart your computer. Hopefully, you have followed this chapter correctly, and this won't happen to you. Not until you start experimenting, anyway. :]

**Note:** For example, I coded `encoder.useResource(icbBuffer, usage: .write)` instead of `encoder.useResource(icb, usage: .write)`, and my computer locked up. I was testing this before setting up the app render options, and the app would automatically restart when the computer did, so eventually I booted the computer into safe mode. The GPU can be a treacherous area.

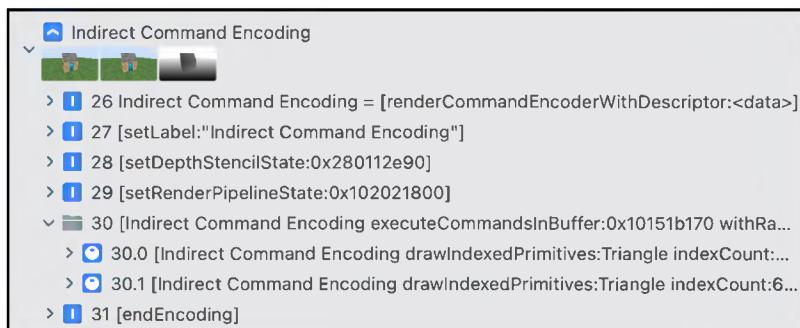
► Build and run the app.

If you've fulfilled your mission and your code is all correct, the renders will be exactly the same on both options.



*The final render*

Capture the GPU workload and examine both the compute command list and the render command list — take note where exactly the resources load. Notice how few commands there are on the render command coder.



*The down-sized render pass*

## Challenge

In the **challenge** folder for this chapter, you'll find an app similar to the one in the previous chapter that includes rendering multiple submeshes. Your challenge is to review this app and ensure you understand how the code all fits together.

The app separates out static models (buildings) and dynamic models (skeletons). `IndirectRenderPass` renders the static models, and `ForwardRenderPass` renders the dynamic models. There is nothing conceptually new in the project, but adding multiple submeshes increases the complexity.

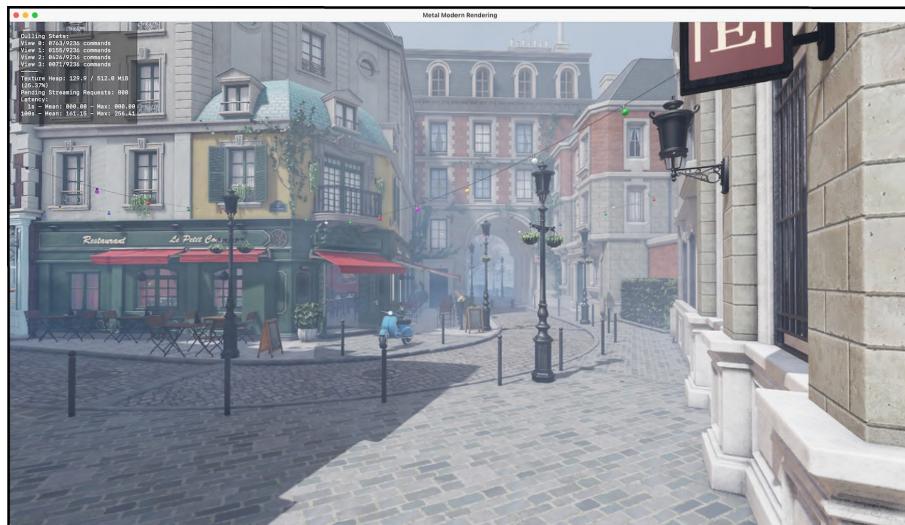
## Key Points

- Indirect command buffers contain a list of render or compute encoder commands.
- You can create the list of commands on the CPU at the start of your app. For simple static rendering work, this will be fine.
- Argument buffers should match your shader function parameters. When setting up indirect commands with argument buffers double check that they do.
- Argument buffers point to other resources. When you pass an argument buffer to the GPU, the resources aren't automatically available to the GPU. You must also `useResource`. If you don't you'll get unexpected rendering results.
- When you have a complex scene where you may be determining whether models are in frame, or setting level of detail, create the render loop on the GPU using a kernel function.

# Where to Go From Here?

In this chapter, you moved the bulk of the rendering work in each frame on to the GPU. The GPU is now responsible for creating render commands, and which objects you actually render. Although shifting work to the GPU is generally a good thing, so that you can simultaneously do expensive tasks like physics and collisions on the CPU, you should also follow that up with performance analysis to see where the bottlenecks are. You can read more about this in Chapter 31, “Performance Optimization”.

GPU-driven rendering is a fairly recent concept, and the best resources are Apple’s WWDC sessions listed in **references.markdown** in the resources folder for this chapter.



*Apple sample: Modern Rendering With Metal*

Apple’s sample Modern Rendering With Metal renders Amazon’s huge Bistro scene, with heavy mesh resources and many lights. The sample uses several advanced techniques including indirect command buffers for GPU-driven rendering, and is the best project to tear apart and analyze.

# Section IV: Ray Tracing

In this section, you'll trace rays to render objects with more realism than the rasterization techniques you've used up to now. As a bonus, you'll also do some post-processing with Metal Performance Shaders. To wrap up, you'll consider how best to profile and optimize your game.



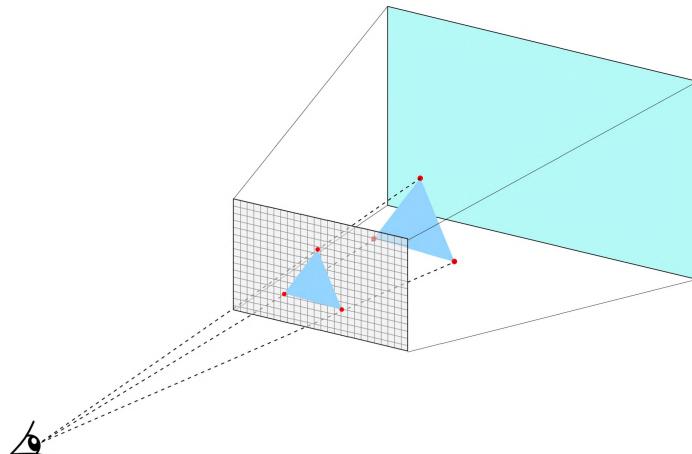
# Chapter 27: Rendering With Rays

In previous chapters, you worked with a traditional pipeline model — a **raster-model**, which uses a *rasterizer* to color the pixels on the screen. In this section, you'll learn about another, somewhat different rendering technique: a **ray-model**.



# Getting Started

In the world of computer graphics, there are two main approaches to rendering graphics. The first approach is **geometry -> pixels**. This approach transforms geometry into pixels using the raster-model. The raster-model assumes you know all of the models and their geometry (triangles) beforehand.



*The raster-model*

A pseudo-algorithm for the raster-model might look something like this:

```
for each triangle in the scene:  
    if visible:  
        mark triangle location  
        apply triangle color  
    if not visible:  
        discard triangle
```

The second approach is **pixels -> geometry**. This approach involves shooting rays from the camera *out of the screen and into the scene* using the ray-model.

A pseudo-algorithm for the ray-model may look something like this:

```
for each pixel on the screen:  
    if there's an intersection (hit):  
        identify the object hit  
        change pixel color  
        optionally bounce the ray  
    if there's no intersection (miss):  
        discard ray  
        leave pixel color unchanged
```

You'll be using the ray-model for the remainder of this section.

In ideal conditions, light travels through the air as a ray following a straight line until it hits a surface. Once the ray hits something, any combination of the following events may happen to the light ray:

- Light gets absorbed into the surface.
- Light gets reflected by the surface.
- Light gets refracted through the surface.
- Light gets scattered from another point under the surface.

When comparing the two models, the raster-model is a faster rendering technique, highly optimized for GPUs. This model scales well for larger scenes and implements antialiasing with ease. If you're creating highly interactive rendered content, such as 1st- and 3rd-person games, the raster-model might be the better choice since pixel accuracy is not paramount.

In contrast, the ray-model is more parallelizable and handles shadows, reflection and refractions more easily. When you're rendering static, far away scenes, the ray-model approach might be the better choice.

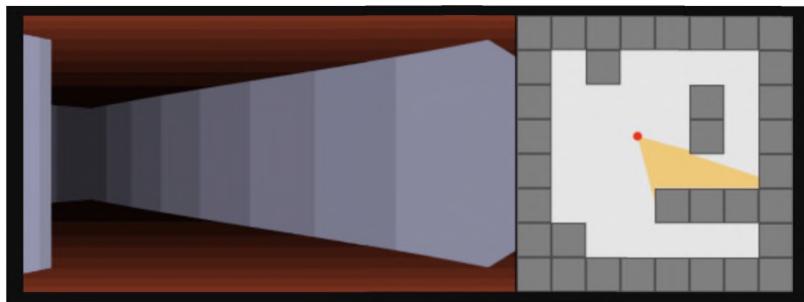
The ray-model has a few variants. Among the most popular are ray casting, ray tracing, path tracing and raymarching. Before you get started, it's important to understand each.

## Ray Casting

In 1968 *Arthur Appel* introduced ray casting, making it one of the oldest ray-model variants. However, it wasn't until 1992 that it became popular in the world of gaming — that's when *Id Software*'s programmer, *John Carmack*, used it for their *Wolfenstein 3D* game. With ray casting, the main idea is to cast rays from the camera into the scene looking for surfaces the ray can hit. In *Wolfenstein 3D*, they used a floor map to describe all of the surfaces in the scene.



Because the object height was usually the same across all objects, scanning was fast and simple — only one ray needs to be cast for each vertical line of pixels (column).



*Ray casting*

But there's more to the simplicity and fast performance of this algorithm than objects having the same height.

The walls, ceiling and floor each had one specific color and texture, and the light source was known ahead of time. With that information, the algorithm could quickly calculate the shading of an object, especially since it assumes that when a surface faces the light, it will be lit (and not hiding in the shadows).

In its rawest form, a ray casting algorithm states that “*For each cell in the floor map, shoot a ray from the camera, and find the closest object blocking the ray path.*”

The number of rays cast tends to equal the screen width:

```
For each pixel from 0 to width:  
    Cast ray from the camera  
    If there's an intersection (hit):  
        Color the pixel in object's color  
        Stop ray and go to the next pixel  
    If there's no intersection (miss):  
        Color the pixel in the background color
```

Ray casting has a few advantages:

- It's a high-speed rendering algorithm because it works with scene constraints such as the number of rays being equal to the width of the screen — about a thousand rays.
- It's suitable for real-time, highly interactive scenes where pixel accuracy is not essential.
- The size occupied on disk is small because scenes don't need to be saved since they're rendered so fast.

There are also disadvantages to this algorithm:

- The quality of the rendered image looks blocky because the algorithm uses only primary rays that stop when they hit an object.
- The scene is limited to basic geometric shapes that can be easily intersected by rays.
- The calculations used for intersections are not always precise.

To overcome some of these disadvantages, you'll learn about another variant of the ray-model known as **ray tracing**.

## Ray Tracing

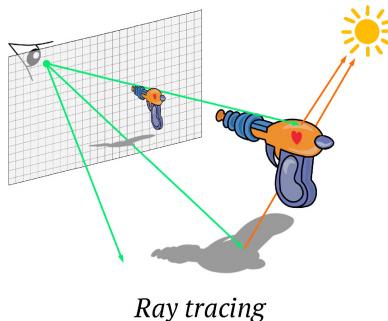
Ray tracing was introduced in 1979 by Turner Whitted. In contrast to ray casting, which shoots about a thousand rays into the scene, ray tracing shoots a ray for each pixel (width \* height), which can easily amount to a million rays!

Advantages of using ray tracing:

- The quality of images rendered is much better than those rendered with ray casting.
- The calculations used for intersections are precise.
- The scene can contain any type of geometric shape, and there are no constraints at all.

Of course, there are also disadvantages to using ray tracing:

- The algorithm is way slower than ray casting.
- The rendered images need to be stored on disk because they take a long time to render again.



Whitted's approach changes what happens when a ray hits a surface.

Here's his algorithm:

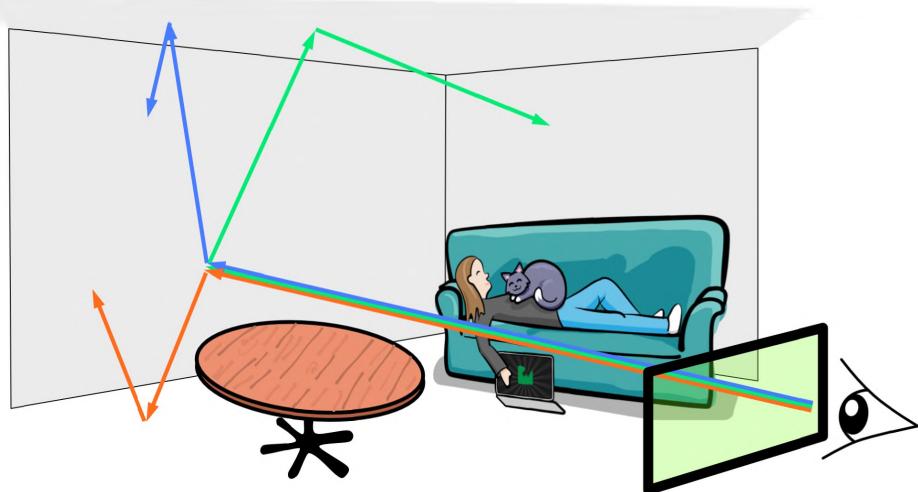
```
For each pixel on the screen:  
  For each object in the scene:  
    If there's an intersection (hit):  
      Select the closest hit object  
      Recursively trace reflection/refraction rays  
      Color the pixel in the selected object's color
```

The recursive step of the ray tracing algorithm is what adds more realism and quality to ray-traced images. However, the holy grail of realistic rendering is path tracing.

## Path Tracing

Path Tracing was introduced as a Monte Carlo algorithm to find a numerical solution to an integral part of the rendering equation. James Kajiya presented the rendering equation in 1986. You'll learn more about the rendering equation in Chapter 29, "Advanced Lighting".

The main idea of the Monte Carlo integration – also known as the Russian Roulette method – is to shoot multiple primary rays for each pixel, and when there's a hit in the scene, shoot just  $K$  more secondary rays (usually just one more) in a random direction for each of the primary rays shot:

*Path Tracing*

The path tracing algorithm looks like this:

```

For each pixel on the screen:
    Reset the pixel color C.
    For each sample (random direction):
        Shoot a ray and trace its path.
        C += incoming radiance from ray.
    C /= number of samples
  
```

Path tracing has a few advantages over other ray-model techniques:

- It's a predictive simulation, so it can be used for engineering or other areas that need precision.
- It's photo-realistic if a large enough number of rays are used.

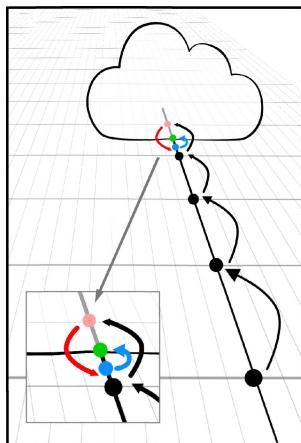
There are some disadvantages too:

- It's slow compared to other techniques, so it is mostly used in *off-line rendering* such as for animated movies.
- It needs precisely defined lights, materials and geometry.

Ah, wouldn't it be great to have a sweet spot where rendering is still close to real-time, and the image quality is more than acceptable? Enter raymarching!

## Raymarching

Raymarching is one of the newer approaches to the ray-model. It attempts to make rendering faster than ray tracing by jumping (or marching) in fixed steps along the ray, making the time until an intersection occurs shorter.



*Raymarching*

The last jump might end up missing the target. When that happens, you can make another jump back but of a smaller size. If you miss the target again, make yet another jump even smaller than the previous one, and so on until you are getting close enough to the target.

The algorithm is straightforward:

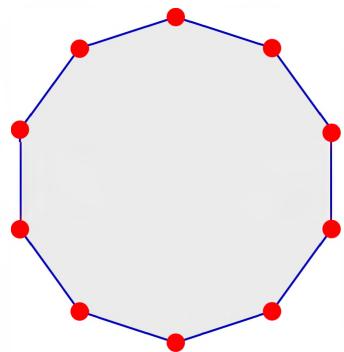
```
For each step up to a maximum number of steps:  
    Travel along the ray and check for intersections.  
    If there's an intersection (hit):  
        Color the pixel in object's color  
    If there's no intersection (miss):  
        Color the pixel in the background color  
        Add the step size to the distance traveled so far.
```

In case the last step is missing the target often, you can retry with a smaller-sized step in the algorithm above. A smaller step improves the accuracy for hitting the target, but it slows down the total marching time.

In 1996, John Hart introduced **sphere tracing** which is a faster raymarching technique used for rendering **implicit surfaces**; it uses geometric distance. Sphere tracing marches along the ray toward the first intersection in steps guaranteed not to go past an implicit surface.

To make a distinction between explicit and implicit surfaces, you need to remember that the raster-model works with geometry stored explicitly as a list of vertices and indices before rendering.

As you can see in the following image, the rasterized circle is made of line segments between vertices:

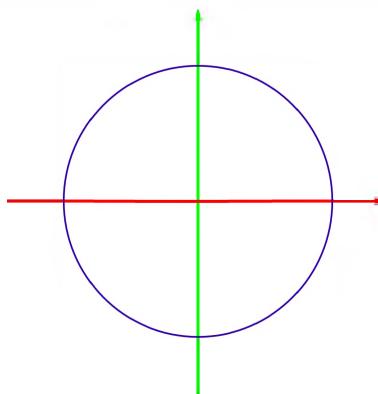


*An explicit surface*

Implicit surfaces, on the other hand, are shapes described by functions rather than by geometry stored before rendering.

So, in this case, the traced circle is perfectly round as each point on the circle is precisely defined by the circle equation:

$$F(X, Y) = X^2 + Y^2 - R^2$$

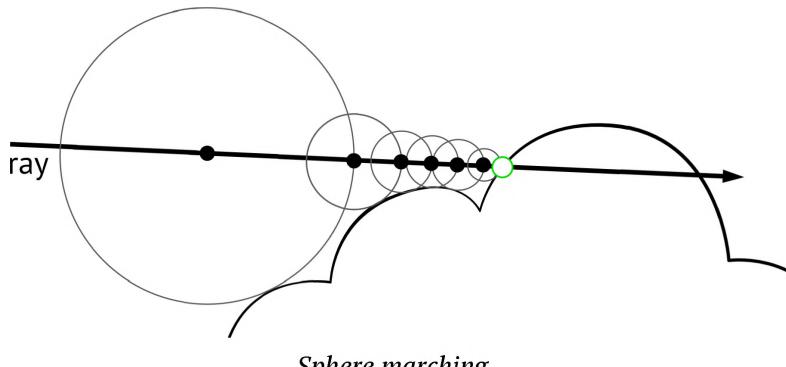


*An implicit surface*

The function that describes a sphere of radius R is straightforward as well:

$$F(X, Y, Z) = X^2 + Y^2 + Z^2 - R^2$$

This type of function can help you estimate the biggest possible sphere that can fit the current marching step. With this technique, you can have variable marching steps which help speed up the marching time.



*Sphere marching*

You'll look at this new, modified algorithm a bit later because you need first to learn how to measure the distance from the current step (ray position) to the nearest surface in the scene.

## Signed Distance Functions

**Signed Distance Functions (SDF)** describe the distance between any given point and the surface of an object in the scene. An SDF returns a negative number if the point is inside that object or positive otherwise.

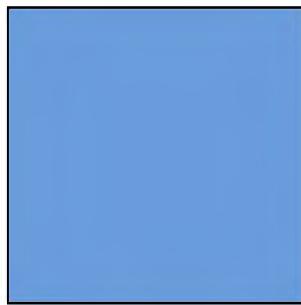
SDFs are useful because they allow for reducing the number of samples used by ray tracing. The difference between the two techniques is that in ray tracing, the intersection is determined by a strict set of equations, while in raymarching the intersection is approximated. Using SDFs, you can march along the ray until you get close enough to an object. Close enough is inexpensive to compute compared to precisely determining intersections.

All right, time to finally write some code!

## The Starter Playground

- In Xcode, open the starter playground included with this chapter.

The initial playground produces a light-blue background.



*The starter playground*

You can see all four playground pages contained in this playground by opening the **Project navigator**, or pressing **Cmd-0**. You can also navigate to next and previous playgrounds by choosing **Next** and **Previous** in the playground page.

All of the playground pages consist of an **MTKView** and a **Renderer**. In the page's **Sources** group, **Renderer.swift** contains the Metal rendering code, which simply consists of setting up a compute command encoder which dispatches threads to a compute function on every frame.

In the page's **Resources** group, **Shaders.metal** contains the compute function which draws to the view's drawable texture.

## Using a Signed Distance Function

- Open the **SDF** page, and inside the **Resources** folder, open **Shaders.metal**. Within the kernel function, add the following code below // SDF:

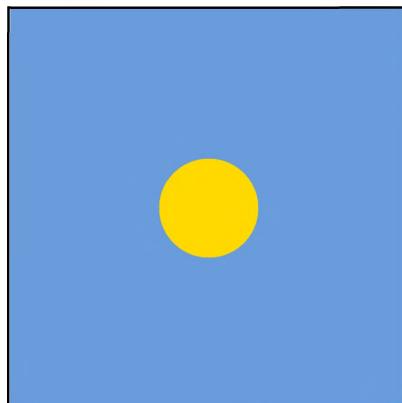
```
// 1
float radius = 0.25;
float2 center = float2(0.0);
// 2
float distance = length(uv - center) - radius;
// 3
if (distance < 0.0) {
    color = float4(1.0, 0.85, 0.0, 1.0);
}
```

Going through the code:

1. Define the radius and center of the circle.
2. Create an SDF that can find the distance to this circle from any point on the screen.
3. Check the sign of the distance variable. If it's negative, it means the point is inside the circle, so change the color to yellow.

**Note:** You learned earlier that the function that describes a circle is  $F(X, Y) = X^2 + Y^2 - R^2$  which is what you used for the SDF. Since the center is at  $(0, 0)$  you can more easily recognize the function in this reduced form instead: `dist = length(uv) - radius`.

► Run the playground, and you'll see a yellow circle in the middle of the screen:



A yellow circle drawn with an SDF

Now that you know how to calculate the distance to a circle from any point on the screen, you can apply the same principle and calculate the distance to a sphere too.

## The Raymarching Algorithm

- Open the **Raymarching** playground page.
- Inside the **Resources** folder, open **Shaders.metal**.

- Start by adding the following above the kernel function:

```
struct Sphere {
    float3 center;
    float radius;
    Sphere(float3 c, float r) {
        center = c;
        radius = r;
    }
};
```

A `Sphere` has a center, a radius and a constructor, so you can build a sphere with the provided arguments.

- Next, create a structure for the ray you'll march along in the scene:

```
struct Ray {
    float3 origin;
    float3 direction;
    Ray(float3 o, float3 d) {
        origin = o;
        direction = d;
    }
};
```

A `Ray` has the ray origin, direction and a constructor.

- Add the following code below the code you just added:

```
float distanceToSphere(Ray r, Sphere s) {
    return length(r.origin - s.center) - s.radius;
}
```

You create an SDF for calculating the distance from a given point to the sphere. The difference from the old function is that your point is now marching along the ray, so you use the ray position instead.

Remember the raymarching algorithm you saw earlier?

```
For each step up to a maximum number of steps:
    Travel along the ray and check for intersections.
    If there's an intersection (hit):
        Color the pixel in object's color
    If there's no intersection (miss):
        Color the pixel in the background color
    Add the step size to the distance traveled so far.
```

You can now turn that into code. The first thing you need is a ray to march along with the sphere.

► Inside the kernel function, add the following code below // raymarching:

```
// 1
Sphere s = Sphere(float3(0.0), 1.0);
Ray ray = Ray(float3(0.0, 0.0, -3.0),
              normalize(float3(uv, 1.0)));
// 2
for (int i = 0.0; i < 100.0; i++) {
    float distance = distanceToSphere(ray, s);
    if (distance < 0.001) {
        color = float3(1.0);
        break;
    }
    ray.origin += ray.direction * distance;
}
```

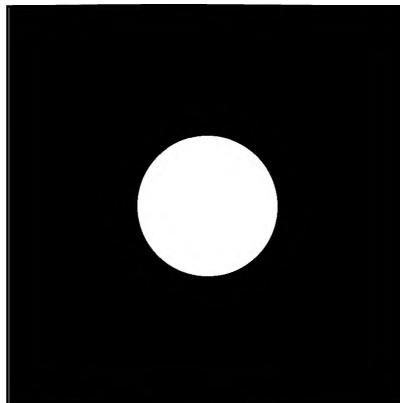
Going through the code:

1. Create a sphere object and a ray. You need to normalize the direction of the ray to make sure its length will always be 1 thus making sure the ray will never miss the object by overshooting the ray beyond the intersection point.
2. Loop enough times to get acceptable precision. On each iteration, calculate the distance from the current position along the ray to the sphere while also checking the distance against **0.001**, a number small enough that's still not zero to make sure you're not yet touching the sphere. If you did, color it white. Otherwise, update the ray position by moving it closer to the sphere.

**Note:** You use **100** in this case, but you can try with an increased number of steps to see how the quality of the rendered image improves — being at the expense of more GPU time used, of course.

That's it! That loop is the essence of raymarching.

- Run the playground now:



*A raymarched sphere*

What if you want other objects or even more spheres in the scene? You can do that with a neat instancing trick.

- First, set up the sphere and ray. Replace these lines inside the kernel function:

```
Sphere s = Sphere(float3(0.0), 1.0);
Ray ray = Ray(float3(0.0, 0.0, -3.0),
              normalize(float3(uv, 1.0)));
```

- With:

```
Sphere s = Sphere(float3(1.0), 0.5);
Ray ray = Ray(float3(1000.0), normalize(float3(uv, 1.0)));
```

Here, you position the sphere at  $(1, 1, 1)$  and set its radius to  $0.5$ . This means the sphere is now contained in the  $[0.5 - 1.5]$  range. The ray origin is now much farther away to give you plenty of range for the number of steps you'll use for marching.

Create a function that takes in a ray as the only argument. All you care about now is to find the shortest distance to a complex scene that contains multiple objects.

► Place this function below `distanceToSphere`:

```
float distanceToScene(Ray r, Sphere s, float range) {
    // 1
    Ray repeatRay = r;
    repeatRay.origin = fmod(r.origin, range);
    // 2
    return distanceToSphere(repeatRay, s);
}
```

1. You make a local copy of the ray (a.k.a. that neat trick to do instancing). By using the `fmod` or modulo function on the ray's origin, you effectively repeat the space throughout the entire screen. This creates an infinite number of spheres, each with its own (repeated) ray. You'll see an example next that should help you understand this better.

2. `distanceToSphere` returns the value of the repeated ray.

Now, call the function from inside the loop.

► Replace the following line:

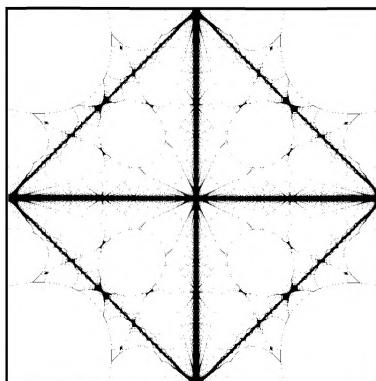
```
float distance = distanceToSphere(ray, s);
```

► With this:

```
float distance = distanceToScene(ray, s, 2.0);
```

You send a range of `2.0`, so the sphere fits safely inside this range.

► Run the playground:



*Interesting, but not spheres*

Um, it's interesting, but where is the promised infinite number of spheres?

► In compute, replace this:

```
output.write(float4(color, 1.0), gid);
```

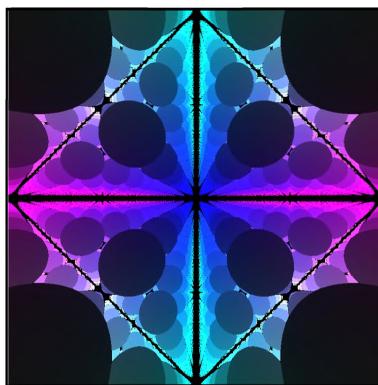
► With:

```
output.write(float4(color * abs((ray.origin - 1000.) / 10.0),  
1.0), gid);
```

That line is complex. You multiply the color with the ray's current position, which is conveniently offset by 1000 to match its initial origin. Next, you divide by 10.0 to scale down the result, which would be bigger than 1.0 and give you a solid white. Then, you guard against negative values by using the abs function because the left side of the screen x is lower than 0 which would give you a solid black.

So at this moment, your X and Y values are between [0, 1], which is what you need to draw colors, and then these colors are also mirrored top/bottom and left/right.

► Run the playground now, and you'll see the following:



*Infinite spheres*

There's one more exciting thing you can do to this scene: animate it!

The kernel function already provides you with a convenient timer variable which `Renderer` updates on the CPU:

```
constant float &time [[buffer(0)]]
```

To create the sensation of movement, you can plug it into a ray's coordinates. You can even use math functions such as `sin` and `cos` to make the movement look like a spiral.

But first, you need to create a camera and make that move along the ray instead.

- Inside `compute`, replace this:

```
Ray ray = Ray(float3(1000.0), normalize(float3(uv, 1.0)));
```

- With:

```
float3 cameraPosition = float3(
    1000.0 + sin(time) + 1.0,
    1000.0 + cos(time) + 1.0,
    time);
Ray ray = Ray(cameraPosition, normalize(float3(uv, 1.0)));
```

You replaced the static ray origin with one that changes over time. The X and Y coordinates move the sphere in a circular pattern while the Z coordinate moves it more into the screen.

The `1.0` that you added to both X and Y coordinates is there to prevent the camera from crashing into the nearest sphere.

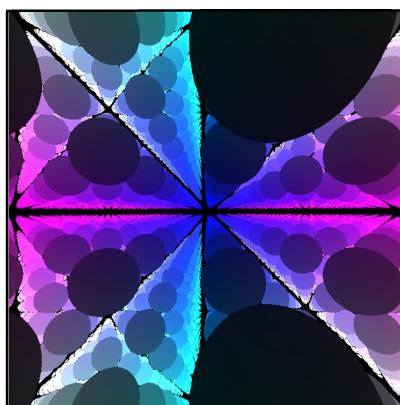
- Now, replace this:

```
output.write(float4(color * abs((ray.origin - 1000.0)/10.0),
    1.0), gid);
```

- With:

```
float3 positionToCamera = ray.origin - cameraPosition;
output.write(float4(color * abs(positionToCamera / 10.0),
    1.0), gid);
```

- Run the playground, and let yourself mesmerized by the trippy animation. But not for too long. There is still work you need to do.



*Animated spheres*



All right, you need to master one more skill before you can create beautifully animated clouds: random noise.

## Creating Random Noise

Noise, in the context of computer graphics, represents perturbations in the expected pattern of a signal. In other words, noise is everything the output contains but was not expected to be there. For example, pixels with different colors that make them seem misplaced among neighboring pixels.

Noise is useful in creating random procedural content such as fog, fire or clouds. You'll work on creating clouds later, but you first need to learn how to handle noise. Noise has many variants such as **Value** noise and **Perlin** noise. However, for the sake of simplicity, you'll only work with value noise in this chapter.

Value noise uses a method that creates a lattice of points which are assigned random values. The noise function returns a number based on the interpolation of values of the surrounding lattice points.

**Octaves** are used in calculating noise to express the multiple irregularities around us. For each octave, you run the noise functions with a different frequency (the period at which data is sampled) and amplitude (the range at which the result can be in). Multiple octaves of this noise can be generated and then summed together to create a form of fractal noise.

The most apparent characteristic of noise is randomness. Since Metal Shading Language does not provide a random function, you'll need to create one yourself. You need a random number between  $[0, 1]$ , which you can get by using the **fract** function. This returns the fractional component of a number.

You use a **pseudorandom number generator** technique that creates sequences of numbers whose properties approximate the properties of sequences of random numbers. This sequence is not truly random because it's determined by an initial seed value which is the same every time the program runs.

- Open the **Random Noise** playground page.
- Inside the **Resources** folder, open **Shaders.metal**, and add the following code above **compute**:

```
float randomNoise(float2 p) {
    return fract(6791.0 * sin(47.0 * p.x + 9973.0 * p.y));
```

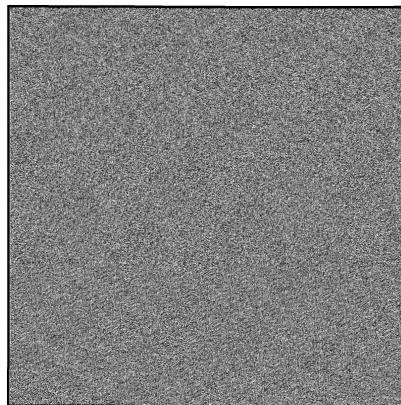


The values used in this function are all prime numbers because they're guaranteed not to return the same fractional part for a different number that would otherwise divide it — one of its factors.

- Inside `compute`, replace the last line with this:

```
float noise = randomNoise(uv);
output.write(float4(float3(noise), 1.0), gid);
```

- Run the playground, and you'll see a pretty decent noise pattern like this one:



*Noise*

You can simulate a zooming-in effect by implementing **tiling**. Tiling splits the view into many tiles of equal size, each with its own solid color.

In `Shaders.metal`, right above this line:

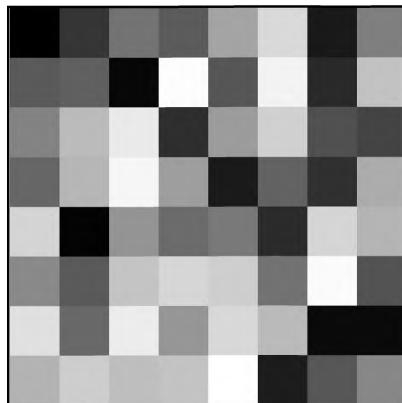
```
float noise = randomNoise(uv);
```

- Add this:

```
float tiles = 8.0;
uv = floor(uv * tiles);
```



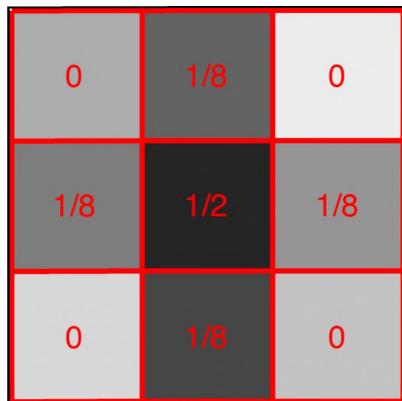
- Run the playground again, and you'll see the tiles:



*Tiled noise*

This pattern, however, is far too heterogeneous. What you need is a smoother noise pattern where colors are not so distinctive from the adjacent ones.

To smooth out the noise pattern, you'll make use of **pixel neighborhoods**, also known as convolution kernels in the world of image processing. One such famous convolution grid is the **Von Neumann neighborhood**:



*Convolution*

Neighborhood averaging produces a blurry result. You can easily express this grid with code.

► In **Shaders.metal**, create a new function above `compute`:

```
float smoothNoise(float2 p) {
    // 1
    float2 north = float2(p.x, p.y + 1.0);
    float2 east = float2(p.x + 1.0, p.y);
    float2 south = float2(p.x, p.y - 1.0);
    float2 west = float2(p.x - 1.0, p.y);
    float2 center = float2(p.x, p.y);
    // 2
    float sum = 0.0;
    sum += randomNoise(north) / 8.0;
    sum += randomNoise(east) / 8.0;
    sum += randomNoise(south) / 8.0;
    sum += randomNoise(west) / 8.0;
    sum += randomNoise(center) / 2.0;
    return sum;
}
```

Going through the code:

1. Store the coordinates for the central pixel and the neighbors located at the four cardinal points relative to the central pixel.
2. Calculate the value noise for each of the stored coordinates and divide it by its convolution weight. Add each of these values to the total noise value sum.

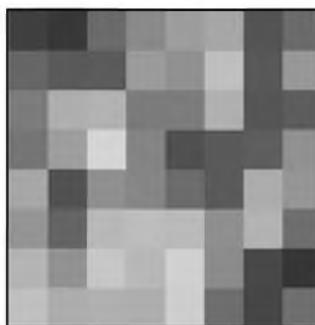
► Now replace this line:

```
float noise = randomNoise(uv);
```

► With this one:

```
float noise = smoothNoise(uv);
```

► Run the playground, and you'll see a smoother noise pattern:

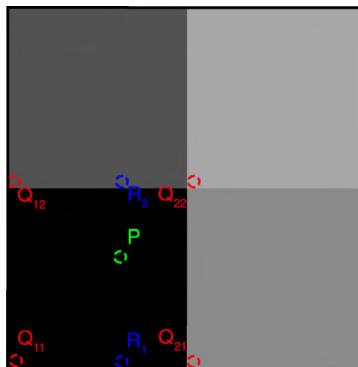


*More homogeneous noise*

The tiles are still distinctive from one another, but notice how the colors are now more homogeneous.

The next step is to smooth the edges between tiles, making them look borderless. This is done with bilinear interpolation. In other words, you mix the colors at the endpoints of a line to get the color at the middle of the line.

In the following image you have the Q values located in the tile corners. Q<sub>11</sub> has a black color. Q<sub>21</sub> has a gray color from the tile to its right. The R<sub>1</sub> value can be computed by interpolating these two values. Similarly, for Q<sub>12</sub> and Q<sub>22</sub>, the value for R<sub>2</sub> can be obtained. Finally, the value of P will be obtained by interpolating the value of R<sub>1</sub> and R<sub>2</sub>:



*Interpolation*

► In **Shaders.metal**, add this function before compute:

```
float interpolatedNoise(float2 p) {
    // 1
    float q11 = smoothNoise(float2(floor(p.x), floor(p.y)));
    float q12 = smoothNoise(float2(floor(p.x), ceil(p.y)));
    float q21 = smoothNoise(float2(ceil(p.x), floor(p.y)));
    float q22 = smoothNoise(float2(ceil(p.x), ceil(p.y)));
    // 2
    float2 ss = smoothstep(0.0, 1.0, fract(p));
    float r1 = mix(q11, q21, ss.x);
    float r2 = mix(q12, q22, ss.x);
    return mix(r1, r2, ss.y);
}
```

Going through the code:

1. Sample the value noise in each of the four corners of the tile.
2. Use `smoothstep` for cubic interpolation. Mix the corner colors to get the R and P colors.

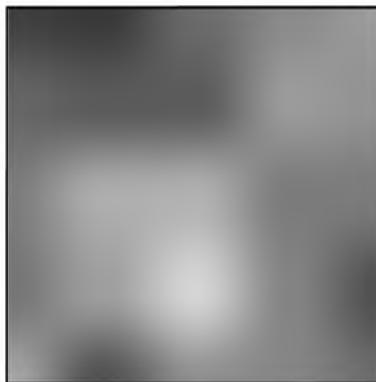
- In `compute`, replace these lines:

```
float tiles = 8.0;
uv = floor(uv * tiles);
float noise = smoothNoise(uv);
```

- With:

```
float tiles = 4.0;
uv *= tiles;
float noise = interpolatedNoise(uv);
```

- Run the playground, and you'll see a smoother noise pattern:



*Smoother noise*

## FBm Noise

The noise pattern looks good, but you can still improve it with the help of another technique called **fractional Brownian motion (fBm)**. By applying an amplitude factor to octaves of noise, the noise becomes more focused and sharper. What's unique about fBm is that when you zoom in on any part of the function, you'll see a similar result in the zoomed-in part.

- Open `Shaders.metal`, and add the following code above `compute`:

```
float fbm(float2 uv, float steps) {
    // 1
    float sum = 0;
    float amplitude = 0.8;
    for(int i = 0; i < steps; ++i) {
        // 2
        sum += interpolatedNoise(uv) * amplitude;
        // 3
```

```
    uv += uv * 1.2;
    amplitude *= 0.4;
}
return sum;
}
```

Going through the code:

1. Initialize the accumulation variable `sum` to `0` and the `amplitude` factor to a value that satisfies your need for the noise quality (try out different values).
2. At each step, compute the value noise attenuated by amplitude and add it to the sum.
3. Update both the location and amplitude values (again, try out different values).

► In `compute`, replace this line:

```
float noise = interpolatedNoise(uv);
```

► With:

```
float noise = fbm(uv, tiles);
```

Here, you use the value of `tiles` in place of `steps` just because they coincidentally have the same value, but you could have created a new variable named `steps` if you wanted it to be a different value than four. By adding four octaves of noise at different amplitudes, you generate a simple gaseous-like pattern.

► Run the playground:



*fBm noise*

Fantastic! You're almost done, but there's one more stop: marching clouds.

## Marching Clouds

All right, it's time to apply what you've learned about signed distance fields, random noise and raymarching by making some marching clouds.

- Open the **Clouds** playground page.

This is a copy of your completed **Random Noise** playground page.

- In the **Resources** folder, open **Shaders.metal**.
- Add the following code after the Ray structure:

```
struct Plane {  
    float yCoord;  
    Plane(float y) {  
        yCoord = y;  
    }  
  
    float distanceToPlane(Ray ray, Plane plane) {  
        return ray.origin.y - plane.yCoord;  
    }  
  
    float distanceToScene(Ray r, Plane p) {  
        return distanceToPlane(r, p);  
    }  
}
```

This is similar to what you used in the raymarching section. Instead of a **Sphere**, however, you create a **Plane** for the ground, an **SDF** for the plane and another one for the scene. You're only returning the distance to the plane in the scene at the moment.

Everything else from now on happens inside **compute**.

- Replace these lines:

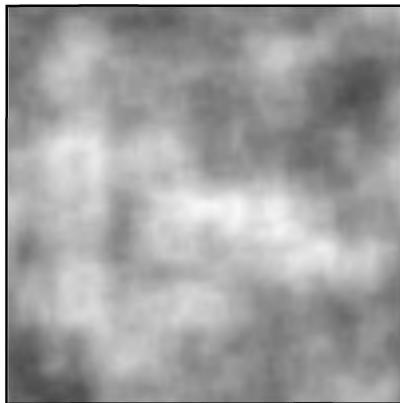
```
uv *= tiles;  
float3 clouds = float3(fbm(uv));
```

- With:

```
float2 noise = uv;  
noise.x += time * 0.1;  
noise *= tiles;  
float3 clouds = float3(fbm(noise));
```

You add `time` to the `X` coordinate, attenuated by `0.1` to slow the movement down a bit.

- Run the playground now, and you'll see the noise pattern gently moving to the left side.



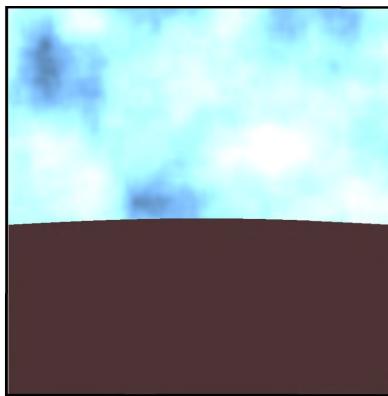
*Animated noise*

- Open **Shaders.metal**, and add this below the previous code:

```
// 1
float3 land = float3(0.3, 0.2, 0.2);
float3 sky = float3(0.4, 0.6, 0.8);
clouds *= sky * 3.0;
// 2
uv.y = -uv.y;
Ray ray = Ray(float3(0.0, 4.0, -12.0),
              normalize(float3(uv, 1.0)));
Plane plane = Plane(0.0);
// 3
for (int i = 0.0; i < 100.0; i++) {
    float distance = distanceToScene(ray, plane);
    if (distance < 0.001) {
        clouds = land;
        break;
    }
    ray.origin += ray.direction * distance;
}
```

Going through the code:

1. Set the colors for the clouds and the sky; then, add the sky color to the noise for a bluish effect.
  2. Since the image is upside down, you reverse the Y coordinate. Create a ray and a plane object.
  3. Apply the raymarching algorithm you learned in the previous section.
- Run the playground one last time, and you'll see marching clouds above the ground plane:



*Marching clouds*

So. Much. Beautiful.

## Key Points

- Ray casting, ray tracing, path tracing and raymarching are all ray-model rendering algorithms that you create in kernel functions, rather than using the rasterizing pipeline with vertex and fragment functions.
- Signed distance functions (SDF) describe the distance between points and object surfaces. The function returns a negative value when the point is inside the object.
- You can't generate random numbers directly on the GPU. You can use fractions of prime numbers to produce a pseudo random number.
- Fractional Brownian motion (fBm) filters octaves of noise with frequency and amplitude settings to produce a finer noise granularity (with more detail in the noise).



# Chapter 28: Advanced Shadows

Shadows and lighting are important topics in Computer Graphics. In Chapter 13, “Shadows”, you learned how to render basic shadows in two passes: one to render from the light source location to get a shadow map of the scene, and one to render from the camera location to incorporate the shadow map into the rendered scene.

Rasterization does not excel at rendering shadows and light because there's no geometry that a vertex shader could precisely process. So now you'll learn how to do it differently.

Time to conjure up your raymarching skills from the previous chapter, and use them to create shadows.

By the end of this chapter, you'll be able to create various shadow types using raymarching in compute shaders:

- Hard shadows.
- Soft shadows.
- Ambient Occlusion.



# The Starter Playground

- In Xcode, open the starter playground included with this chapter.

As in the previous chapter, you can see all three playground pages contained in this playground by opening the **Project navigator**, or pressing **Cmd-0**. Also, all of the playground pages consist of an **MTKView** and a **Renderer**. You'll work solely in the **Resources** group for each page, in **Shaders.metal**.

## Hard Shadows

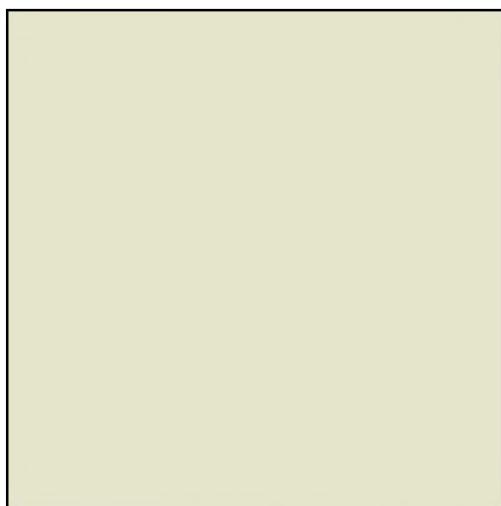
When creating shadows in a rasterized render pass, you create a shadow map, which requires you to bake the shadows.

With raymarching, you make use of **signed distance fields (SDF)**. An SDF is a real-time tool that provides you with the precise distance to a boundary. This makes calculating shadows easy as they come for “free”, meaning that all of the information you need to compute shadows already exists and is available because of the SDF.

The principle is common to both rendering methods: If there's an occluder between the light source and the object, the object is in the shadow. Otherwise, it's lit.

Great! Time to put that wisdom down in code.

- Open the **Hard Shadows** page, and run the playground.



*The starter playground*

You start with an empty view.

- In the **Resources** group, open **Shaders.metal**, and add this before `compute`:

```
struct Rectangle {  
    float2 center;  
    float2 size;  
};
```

This structure will hold rectangle objects with a center and a size.

Next, you'll add a function that gets the distance from any point on the screen to a given rectangle boundary. If its return value is positive, a given point is outside the rectangle; all other values are inside the rectangle.

- Add the following code below the previous code:

```
float distanceToRectangle(float2 point, Rectangle rectangle) {  
    // 1  
    float2 distances =  
        abs(point - rectangle.center) - rectangle.size / 2;  
    return  
    // 2  
    all(sign(distances) > 0)  
    ? length(distances)  
    // 3  
    : max(distances.x, distances.y);  
}
```

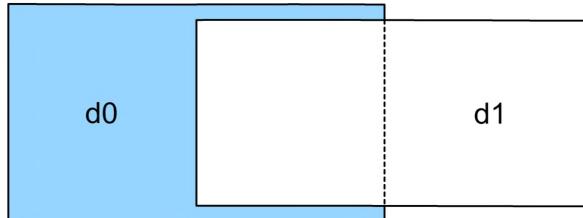
Going through the code:

1. Offset the current point coordinates by the given rectangle center. Then, get the symmetrical coordinates of the given point by using the `abs()` function, and calculate the signed distance to each of the two edges.
2. If those two distances are positive, then you'll need to calculate the distance to the corner.
3. Otherwise, return the distance to the closer edge.

**Note:** In this case, `rectangle.size / 2` is the distance from the rectangle center to an edge, similar to what a radius is for a circle.



Next, is a handy function that lets you subtract one shape from another. Think about Set Theory from back in your school days.



*Shape subtraction*

**Note:** You can find out more about Set Theory here: [https://en.wikipedia.org/wiki/Complement\\_\(set\\_theory\)#Relative\\_complement](https://en.wikipedia.org/wiki/Complement_(set_theory)#Relative_complement)

- Add this function to **Shaders.metal** before **compute**:

```
float differenceOperator(float d0, float d1) {
    return max(d0, -d1);
}
```

This code yields a value that can be used to calculate the difference result from the previous image, where the second shape is subtracted from the first. The result of this function is a signed distance to a compound shape boundary. It'll only be negative when inside the first shape, but outside the second.

- Continue by adding this code to design a basic scene:

```
float distanceToScene(float2 point) {
    // 1
    Rectangle r1 = Rectangle{float2(0.0), float2(0.3)};
    float d2r1 = distanceToRectangle(point, r1);
    // 2
    Rectangle r2 = Rectangle{float2(0.05), float2(0.04)};
    float2 mod = point - 0.1 * floor(point / 0.1);
    float d2r2 = distanceToRectangle(mod, r2);
    // 3
    float diff = differenceOperator(d2r1, d2r2);
    return diff;
}
```

Going through the code:

1. Create a rectangle, and get the distance to it.
2. Create a second, smaller rectangle, and get the distance to it. The difference here is that the area is repeated every 0.1 points — which is a 10th of the size of the scene — using a modulo operation. See the note below.
3. Subtract the second repeated rectangle from the first rectangle, and return the resulting distance.

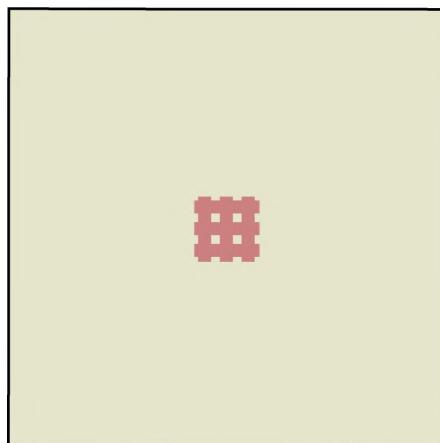
**Note:** The `fmod` function in MSL uses `trunc` instead of `floor`, so you create a custom mod operator because you also want to use the negative values. You use the GLSL specification for `mod` which is  $x - y * \text{floor}(x/y)$ . You need the modulus operator to draw many small rectangles mirrored with a distance of 0.1 from each other.

Finally, use these functions to generate a shape that looks a bit like a fence or a trellis.

- At the end of `compute`, replace the `color` assignment with:

```
float d2scene = distanceToScene(uv);
bool inside = d2scene < 0.0;
float4 color = inside ? float4(0.8, 0.5, 0.5, 1.0) :
    float4(0.9, 0.9, 0.8, 1.0);
```

- Run the playground:



*The initial scene*

For shadows to work, you need to:

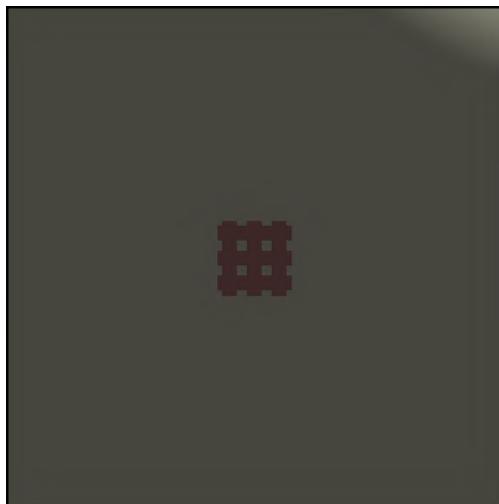
1. Get the distance to the light.
2. Know the light direction.
3. Step in that direction until you either reach the light or hit an object.

► Open **Shaders.metal**, and above the last line in `compute`, add this:

```
float2 lightPos = 2.8 * float2(sin(time), cos(time));
float dist2light = length(lightPos - uv);
color *= max(0.3, 2.0 - dist2light);
```

First, you create a light at position `lightPos`, which you'll animate just for fun using the timer uniform that you passed from the host code. Then, you get the distance from any given point to `lightPos`, and you color the pixel based on the distance from the light — but only if it's not inside an object. You make the color lighter when closer to the light, and darker when further away with the `max()` function to avoid negative values for the brightness of the light.

► Run the playground.



*A moving light*

Notice the moving light that appears at the corners as it circuits the scene.

You just took care of the first two steps: light position and direction. Now it's time to handle the third one: the shadow function.

► In **Shaders.metal**, add this above `compute`:

```
float getShadow(float2 point, float2 lightPos) {
    // 1
    float2 lightDir = lightPos - point;
    // 2
    for (float lerp = 0; lerp < 1; lerp += 1 / 300.0) {
        // 3
        float2 currentPoint = point + lightDir * lerp;
        // 4
        float d2scene = distanceToScene(currentPoint);
        if (d2scene <= 0.0) { return 0.0; }
    }
    return 1.0;
}
```

Going through the code:

1. Get a vector from the point to the light.
2. Use a loop to divide the vector into many smaller steps. If you don't use enough steps, you might jump past the object, leaving holes in the shadow.
3. Calculate how far along the ray you are currently, and move along the ray by this `lerp` distance to find the point in space you are sampling.
4. See how far you are from the surface at that point, and then test if you're inside an object. If yes, return 0, because you're in the shadow. Otherwise, return 1, because the ray didn't hit an object.

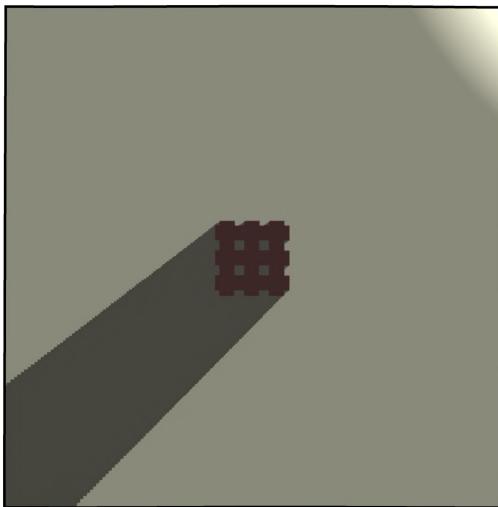
It's finally time to see some shadows.

► Above the last line in `compute`, add this:

```
float shadow = getShadow(uv, lightPos);
color *= 2;
color *= shadow * .5 + .5;
```

A value of 2 is used here to enhance the light brightness and the effect of the shadow. Feel free to play with various values and notice how changes affect it.

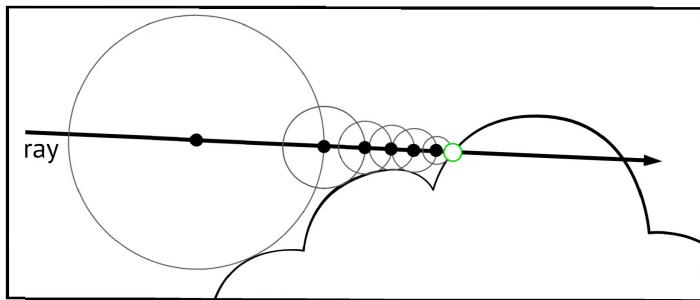
► Run the playground.



*A moving shadow*

The shadow loop goes in 1-pixel steps, which is not good performance-wise. You can improve that a little by moving along in big steps, provided you don't step past the object. You can safely step in any direction by the distance to the scene instead of a fixed step size, and this way you skip over empty areas fast.

When finding the distance to the nearest surface, you don't know what direction the surface is in, but you have the radius of a circle that intersects with the nearest part of the scene. You can trace along the ray, always stepping to the edge of the circle until the circle radius becomes  $0$ , which means it intersected a surface.

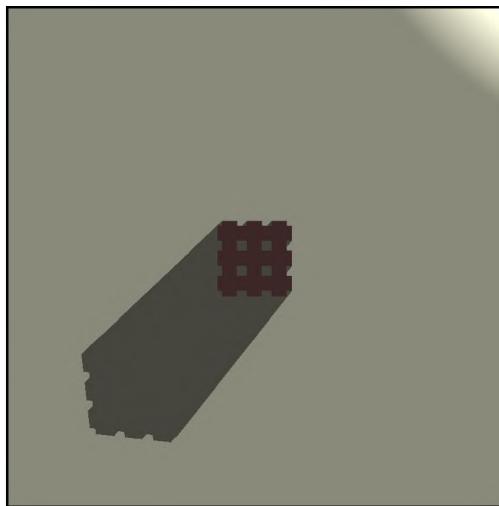


*Sphere intersection*

- In **Shaders.metal**, replace the contents of the `getShadow` function with this:

```
float2 lightDir = normalize(lightPos - point);
float shadowDistance = 0.75;
float distAlongRay = 0.0;
for (float i = 0; i < 80; i++) {
    float2 currentPoint = point + lightDir * distAlongRay;
    float d2scene = distanceToScene(currentPoint);
    if (d2scene <= 0.001) { return 0.0; }
    distAlongRay += d2scene;
    if (distAlongRay > shadowDistance) { break; }
}
return 1.0;
```

- Run the playground again, and the shadow is now faster and looks more accurate.



*A more accurate shadow*

In raymarching, the size of the step depends on the distance from the surface. In empty areas, it jumps big distances, and it can travel a long way. However, if it's parallel to the object and close to it, the distance is always small, so the jump size is also small. That means the ray travels very slowly. With a fixed number of steps, it doesn't travel far. With eighty or more steps you should be safe from getting holes in the shadow.

Congratulations, you made your first hard shadow. Next, you'll be looking into soft shadows. Soft shadows tend to be more realistic and thus, better looking.

## Soft Shadows

Shadows are not only black or white, and objects aren't just in shadow or not. Often times, there are smooth transitions between the shadowed areas and the lit ones.

- In the playground, select the **Soft Shadows** playground page.

If you run the playground now, the view will be entirely black.

- In the **Resources** folder, open **Shaders.metal**.

- First, before `compute`, add structures to hold a ray, a sphere, a plane and a light object:

```
struct Ray {
    float3 origin;
    float3 direction;
};

struct Sphere {
    float3 center;
    float radius;
};

struct Plane {
    float yCoord;
};

struct Light {
    float3 position;
};
```

Nothing new or worth noting here, except that for a plane, all you need to know is its Y-coordinate because it's a horizontal plane.

- Next, create a few distance operation functions to help you determine distances between elements of the scene:

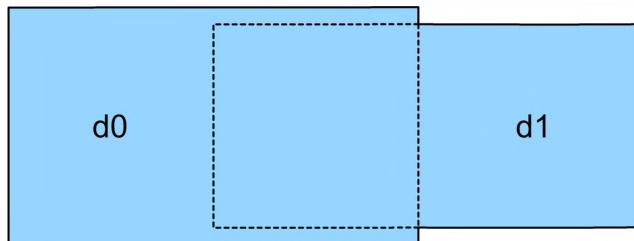
```
float distToSphere(Ray ray, Sphere s) {
    return length(ray.origin - s.center) - s.radius;
}

float distToPlane(Ray ray, Plane plane) {
    return ray.origin.y - plane.yCoord;
}

float differenceOp(float d0, float d1) {
    return max(d0, -d1);
}
```

```
float unionOp(float d0, float d1) {
    return min(d0, d1);
}
```

Only the union function is new here, which lets you join two areas together.



*The union operation*

You'll create a function that gives you the closest distance to any object in the scene. You'll be able to use this function to generate a shape that looks like a hollow sphere with holes in it.

► Before `compute`, add the new function:

```
float distToScene(Ray r) {
    // 1
    Plane p = Plane{0.0};
    float d2p = distToPlane(r, p);
    // 2
    Sphere s1 = Sphere{float3(2.0), 2.0};
    Sphere s2 = Sphere{float3(0.0, 4.0, 0.0), 4.0};
    Sphere s3 = Sphere{float3(0.0, 4.0, 0.0), 3.9};
    // 3
    Ray repeatRay = r;
    repeatRay.origin = fract(r.origin / 4.0) * 4.0;
    // 4
    float d2s1 = distToSphere(repeatRay, s1);
    float d2s2 = distToSphere(r, s2);
    float d2s3 = distToSphere(r, s3);
    // 5
    float dist = differenceOp(d2s2, d2s3);
    dist = differenceOp(dist, d2s1);
    dist = unionOp(d2p, dist);
    return dist;
}
```

Going through the code:

1. Create a plane and calculate the distance to it from the current ray.
2. Create three spheres: one small one, and two larger ones that are concentric.

3. Create a repeated ray, just as you did in the previous chapter, that mirrors the small sphere located between `float3(0)` and `4.0` on each of the three axes. The `fract` function returns the fractional part of a value.
4. Calculate the distance to the three spheres. The small sphere is created repeatedly every `4.0` units in all directions.
5. Calculate the difference between the two large spheres first, which results in a large hollow sphere. Then, subtract the small one from them, resulting in the large sphere having holes in it. Finally, join the result with the plane to complete the scene.

In Chapter 7, “The Fragment Function”, you learned about normals and why they’re needed. Next, you’ll create a function that finds the normal on any surface. As an example, on your plane, the normal is always pointing up, so its vector is `(0, 1, 0)`; the normal in 3D space is a `float3`, and you need to know its precise position on the ray. A plane, however, is a trivial case.

Assume the ray touches the left side of a sphere situated at the origin. The normal vector is `(-1, 0, 0)` at that contact point that’s pointing to the left, and away from the sphere. If the ray moves slightly to the right of that point, it’s inside the sphere (e.g., `-0.001`). If the ray moves slightly to the left, it’s outside the sphere (e.g., `0.001`).

If you subtract left from right, you get  $(-0.001 - 0.001) = -0.002$ , which still points to the left, so this is your X-coordinate of the normal. Repeat this for Y and Z.

► Add this before `compute`:

```
float3 getNormal(Ray ray) {
    float2 eps = float2(0.001, 0.0);
    float3 n = float3(
        distToScene(Ray{ray.origin + eps.xyy, ray.direction}) -
        distToScene(Ray{ray.origin - eps.xyy, ray.direction}),
        distToScene(Ray{ray.origin + eps.yxy, ray.direction}) -
        distToScene(Ray{ray.origin - eps.yxy, ray.direction}),
        distToScene(Ray{ray.origin + eps.yyx, ray.direction}) -
        distToScene(Ray{ray.origin - eps.yyx, ray.direction}));
    return normalize(n);
}
```

`eps` is a 2D vector, so you can easily do vector swizzling using the chosen value 0.001 for one coordinate, and 0 for the other two coordinates, as needed in each case.

You covered all of the cases and checked that the ray is either inside or outside on all three axes. Finally, you're ready to see some visuals. You'll be writing a raymarching loop again.

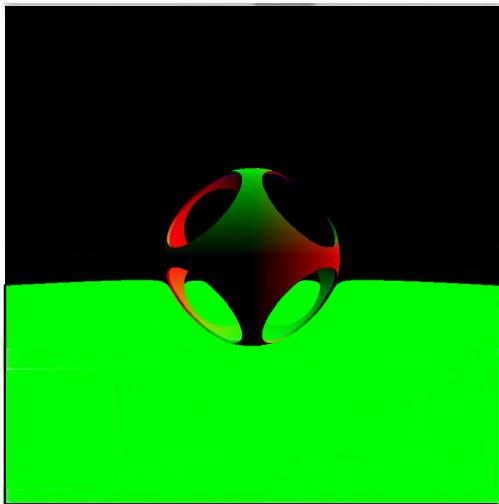
- Replace the last line of `compute`, with this:

```
// 1
Ray ray = Ray{float3(0., 4., -12), normalize(float3(uv, 1.))};
// 2
for (int i = 0; i < 100; i++) {
    // 3
    float dist = distToScene(ray);
    // 4
    if (dist < 0.001) {
        col = float3(1.0);
        break;
    }
    // 5
    ray.origin += ray.direction * dist;
}
// 6
float3 n = getNormal(ray);
output.write(float4(col * n, 1.0), gid);
```

Going through the code:

1. Create a ray to travel with inside the scene.
2. Use the loop to divide the ray into many smaller steps. If you don't use enough steps, you might jump past the object.
3. Calculate the new distance to the scene.
4. See how far you are from the surface at that point, and then test if you're inside an object. If yes, break out of the loop.
5. Move along the ray by the distance to the scene to find the point in space you are sampling at.
6. Get the normal so that you can calculate the color of every pixel.

- Run the **Soft Shadows** playground, and you'll see the colors representing the normal values.



*Colors representing normals*

Now that you have normals, you can calculate lighting for each pixel in the scene.

- In **Shaders.metal**, create a new function above **compute**:

```
float lighting(Ray ray, float3 normal, Light light) {
    // 1
    float3 lightRay = normalize(light.position - ray.origin);
    // 2
    float diffuse = max(0.0, dot(normal, lightRay));
    // 3
    float3 reflectedRay = reflect(ray.direction, normal);
    float specular = max(0.0, dot(reflectedRay, lightRay));
    // 4
    specular = pow(specular, 200.0);
    return diffuse + specular;
}
```

Going through the code:

1. Find the direction to the light ray by normalizing the distance between the light position and the current ray origin.
2. For diffuse lighting, you need the angle between the normal and **lightRay**, that is, the dot product of the two. Also, make sure you're never using negative values by making **0** the minimum value possible.

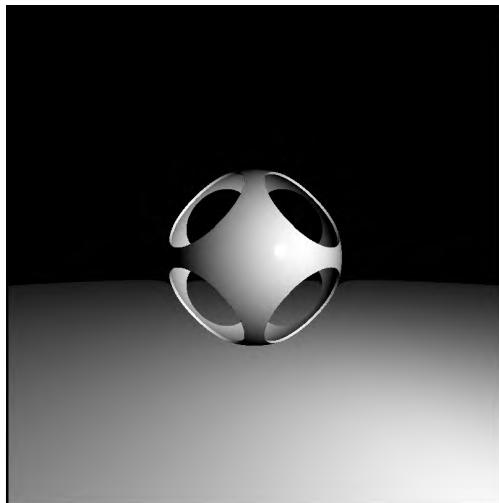
3. For specular lighting, you need reflections on surfaces, and they depend on the angle you're looking at. You first cast a ray into the scene, reflect it from the surface, and then measure the angle between the reflected ray and `lightRay`.
4. Finally, take a high power of that value to make it much sharper and return the combined light.

► Replace the last line of `compute`, with this:

```
Light light = Light{float3(sin(time) * 10.0, 5.0,
                           cos(time) * 10.0)};
float l = lighting(ray, n, light);
output.write(float4(col * l, 1.0), gid);
```

You create a light that circles around in time and use it to calculate the lighting in the scene.

► Run the playground, and you'll see the light circling your central sphere:



*A light circling the sphere*

Next, shadows!

► In `Shaders.metal`, add this function before `compute`:

```
float shadow(Ray ray, Light light) {
    float3 lightDir = light.position - ray.origin;
    float lightDist = length(lightDir);
    lightDir = normalize(lightDir);
    float distAlongRay = 0.01;
    for (int i = 0; i < 100; i++) {
```

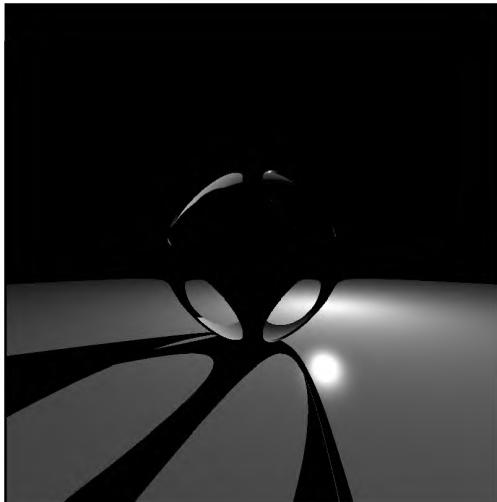
```
Ray lightRay = Ray{ray.origin + lightDir * distAlongRay,
                    lightDir};
float dist = distToScene(lightRay);
if (dist < 0.001) { return 0.0; }
distAlongRay += dist;
if (distAlongRay > lightDist) { break; }
}
return 1.0;
}
```

The shadow function is quite similar to that of hard shadows with a few modifications. You normalize the direction of the light, and then you keep updating the distance along the ray as you march along with it. You also reduce the number of steps to only 100.

- Replace the last line in `compute` with this:

```
float s = shadow(ray, light);
output.write(float4(col * l * s, 1.0), gid);
```

- Run the playground, and you'll see the light casting shadows.



*Light casting shadows*

Time to finally get some soft shadows in the scene.

In real life, a shadow spreads out the farther it gets from an object. For example, where an object touches the floor, you get a sharp shadow; but farther away from the object, the shadow is more blurred. In other words, you start at some point on the floor, march toward the light, and have either a hit or a miss.

Hard shadows are straightforward: you hit something, it's in the shadow. Soft shadows have in-between stages.

► In **Shaders.metal**, replace the **shadow** function with this:

```
// 1
float shadow(Ray ray, float k, Light l) {
    float3 lightDir = l.position - ray.origin;
    float lightDist = length(lightDir);
    lightDir = normalize(lightDir);
// 2
    float light = 1.0;
    float eps = 0.1;
// 3
    float distAlongRay = eps * 2.0;
    for (int i=0; i<100; i++) {
        Ray lightRay = Ray{ray.origin + lightDir * distAlongRay,
                           lightDir};
        float dist = distToScene(lightRay);
// 4
        light = min(light, 1.0 - (eps - dist) / eps);
// 5
        distAlongRay += dist * 0.5;
        eps += dist * k;
// 6
        if (distAlongRay > lightDist) { break; }
    }
    return max(light, 0.0);
}
```

Going through the code, here are the differences from the previous shadow function:

1. Add an attenuator **k** as a function argument, which you'll use to get intermediate values of light.
2. Start with a white light and a small value for **eps**. This is a variable that tells you how much wider the beam is as you go out into the scene. A thin beam means a sharp shadow while a wide beam means a soft shadow.
3. Start with a small **distAlongRay**, because otherwise, the surface at this point would shadow itself.
4. Compute the light by subtracting the distance from the beam width **eps** and then dividing by it. This gives you the percentage of beam covered. If you invert it ( $1 - \text{beam width}$ ) you get the percentage of beam that's in the light. Then, take the minimum of this new value and light to preserve the darkest shadow as you march along the ray.

5. Move along the ray, and increase the beam width in proportion to the distance traveled and scaled by the attenuator  $k$ .
6. If you're past the light, break out of the loop. Avoid negative values by returning the maximum between  $0.0$  and the value of light.

Next, adapt the compute kernel code to work with the new shadow function.

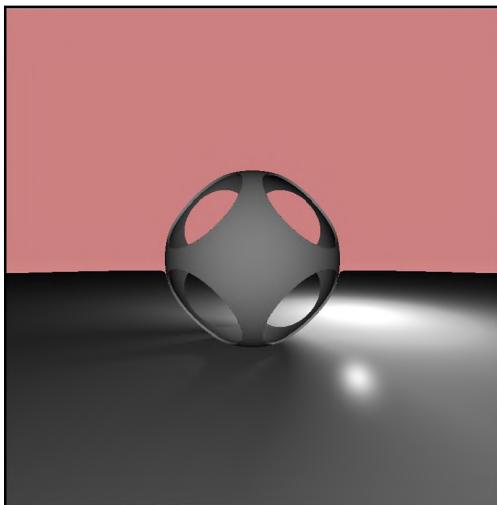
► In `compute`, replace all of the lines after the one where you created the `Ray` object, with this:

```
// 1
bool hit = false;
for (int i = 0; i < 200; i++) {
    float dist = distToScene(ray);
    if (dist < 0.001) {
        hit = true;
        break;
    }
    ray.origin += ray.direction * dist;
}
// 2
col = float3(1.0);
// 3
if (!hit) {
    col = float3(0.8, 0.5, 0.5);
} else {
    float3 n = getNormal(ray);
    Light light = Light{float3(sin(time) * 10.0, 5.0,
                                cos(time) * 10.0)};
    float l = lighting(ray, n, light);
    float s = shadow(ray, 0.3, light);
    col = col * l * s;
}
// 4
Light light2 = Light{float3(0.0, 5.0, -15.0)};
float3 lightRay = normalize(light2.position - ray.origin);
float fl = max(0.0, dot(getNormal(ray), lightRay) / 2.0);
col = col + fl;
output.write(float4(col, 1.0), gid);
```

Going through the code:

1. Add a Boolean that tells you whether or not you hit the object. If the distance to the scene is within  $0.001$ , you have a hit.
2. Start with a default white color. This is important, because when you later multiply this color with the value of shadow and that of the light; white will never influence the result because of multiplying by 1.

3. If there's no hit, color everything in a nice sky color, otherwise determine the shadow value.
  4. Add another fixed light source in front of the scene to see the shadows in greater detail.
- Run the playground, and you'll see a beautiful combination of shadow tones.



*Shadow Tones*

## Ambient Occlusion

**Ambient occlusion (AO)** is a global shading technique, unlike the Phong local shading technique you learned about in Chapter 10, “Lighting Fundamentals”. AO is used to calculate how exposed each point in a scene is to ambient lighting which is determined by the neighboring geometry in the scene.

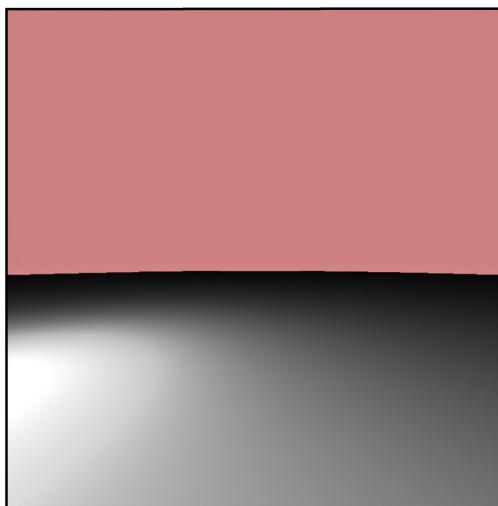
AO is, however, a weak variant of global illumination. It looks like a scene on a rainy day and feels like a non-directional, diffuse shading effect. For hollow objects, AO makes the interior look darker because the light is even more occluded inside. As you move towards the edges of the object, it looks lighter and lighter.

Only large objects are taken into consideration when computing the amount of ambient light, such as the sky, walls or any other objects that would normally be big enough to cast a shadow if they were lit. AO is usually a fragment post-processing technique. However, you are looking into it in this chapter because AO is a type of shadow.

In the following top-down image, you can see how the base of the curved wall is darker, as well as the base of the box.



- In the playground, open the **Ambient Occlusion** playground page.
- Run the playground, and you'll see an scene with a ground plane and a light.



*Ambient occlusion starter scene*

- In the **Resources** folder, open **Shaders.metal**.

- Add a new box object type below the other types:

```
struct Box {
    float3 center;
    float size;
};
```

- Next, add a new distance function for Box before distToScene:

```
float distToBox(Ray r, Box b) {
    // 1
    float3 d = abs(r.origin - b.center) - float3(b.size);
    // 2
    return min(max(d.x, max(d.y, d.z)), 0.0)
        + length(max(d, 0.0));
}
```

Going through the code:

1. Offset the current ray origin by the center of the box. Then, get the symmetrical coordinates of the ray position by using the `abs` function. Offset the resulting distance `d` by the length of the box edge.
2. Get the distance to the farthest edge by using the `max` function, and then get the smaller value between `0` and the distance you just calculated. If the ray is inside the box, this value will be negative, so you need to add the larger length between `0` and `d`.

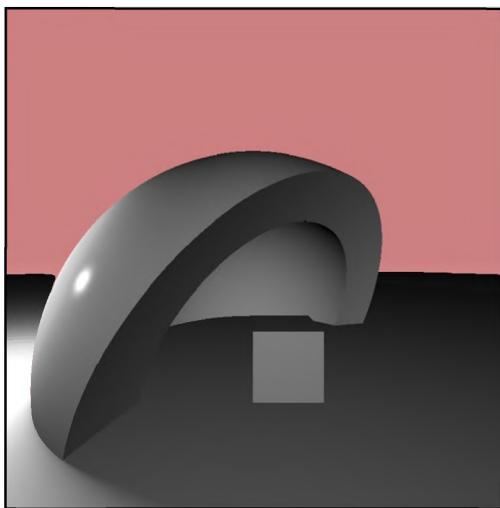
- Replace the return line in `distToScene` with this:

```
// 1
Sphere s1 = Sphere{float3(0.0, 0.5, 0.0), 8.0};
Sphere s2 = Sphere{float3(0.0, 0.5, 0.0), 6.0};
Sphere s3 = Sphere{float3(10., -5., -10.), 15.0};
float d2s1 = distToSphere(r, s1);
float d2s2 = distToSphere(r, s2);
float d2s3 = distToSphere(r, s3);
// 2
float dist = differenceOp(d2s1, d2s2);
dist = differenceOp(dist, d2s3);
// 3
Box b = Box{float3(1., 1., -4.), 1.};
float dtb = distToBox(r, b);
dist = unionOp(dist, dtb);
dist = unionOp(d2p, dist);
return dist;
```

Going through the code:

1. Draw two spheres with the same center: one with a radius of 8, and one with a radius of 6. Draw a third, larger sphere at a different location.
2. Subtract the second sphere from the first, resulting in a hollow, thicker sphere. Subtract the third sphere from the hollow sphere to make a cross-section through it.
3. Add a box and a plane to complete the scene.

► Run the **Ambient Occlusion** playground, and you'll see the hollow sphere, box and plane.



*The ambient occlusion scene*

Time to work on the ambient occlusion code.

► In **Shaders.metal**, create a skeleton function above **compute**:

```
float ao(float3 pos, float3 n) {  
    return n.y * 0.5 + 0.5;  
}
```

This function uses the normal's Y component for light and adds **0.5** to it. This makes it look like there's light directly above.

► Inside **compute**, since there are no shadows anymore, replace this line:

```
col = col * l * s;
```



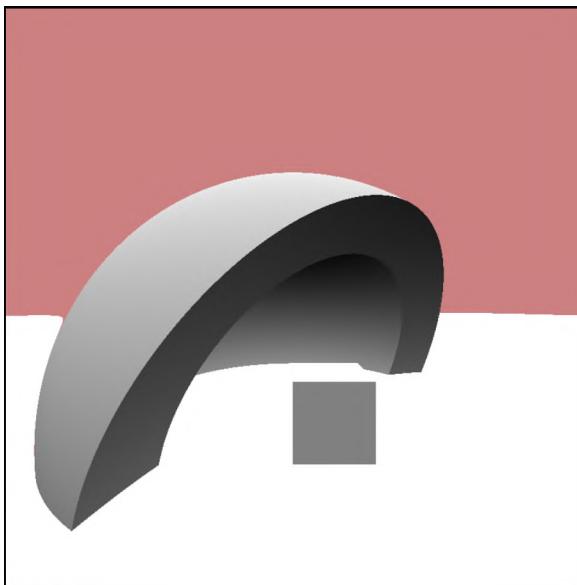
- With this:

```
float o = ao(ray.origin, n);
col = col * o;
```

- At the end of compute, remove this:

```
col = col + fl;
```

- Remove the unused variable definitions light, l, s, light2, lightRay and f1.
- Run the playground, and you'll see the same scene as before — this time without the shadows, and the surfaces pointing upward are brighter.



This is a good start, but it's not how ambient occlusion should look — at least not yet.

**Ambient** means that the light does not come from a well-defined light source, but is rather lighting coming from other objects in the scene, all contributing to the general scene light. **Occlusion** means how much of the ambient light is blocked.

The main idea about ambient occlusion is to use the point where the ray hits the surface and look at what's around it. If there's an object anywhere around it that will block most of the light nearby, that area will be dark. If there's nothing around it, then the area is well lit. For in-between situations, you need more precision about how much light was occluded.

**Cone tracing** is a technique that uses a cone instead of a ray. If the cone intersects an object, you don't just have a simple true/false result. You can find out how much of the cone the object covers at that point. Tracing a cone might be a challenge though. You could make a cone using spheres aligned along a line, small at one end and big at the other end. This would be a good cone approximation to use. Since you're doubling the sphere size at each step, that means you travel out from the surface very fast, so you need fewer iterations. That also gives you a nice wide cone.

► In `Shaders.metal`, replace the contents of the `ao` function with this:

```
// 1
float eps = 0.01;
// 2
pos += n * eps * 2.0;
// 3
float occlusion = 0.0;
for (float i = 1.0; i < 10.0; i++) {
    // 4
    float d = distToScene(Ray{pos, float3(0)}));
    float coneWidth = 2.0 * eps;
    // 5
    float occlusionAmount = max(coneWidth - d, 0.);
    // 6
    float occlusionFactor = occlusionAmount / coneWidth;
    // 7
    occlusionFactor *= 1.0 - (i / 10.0);
    // 8
    occlusion = max(occlusion, occlusionFactor);
    // 9
    eps *= 2.0;
    pos += n * eps;
}
// 10
return max(0.0, 1.0 - occlusion);
```

Going through the code:

1. `eps` is both the cone radius and the distance from the surface.
2. Move away a bit to prevent hitting surfaces you're moving away from.
3. `occlusion` is initially zero (the scene is white).

4. Get the scene distance, and double the cone radius so you know how much of the cone is occluded.
  5. Eliminate negative values for the light by using the `max` function.
  6. Get the amount, or ratio, of occlusion scaled by the cone width.
  7. Set a lower impact for more distant occluders; the iteration count provides this.
  8. Preserve the highest occlusion value so far.
  9. Double `eps`, and then move along the normal by that distance.
10. Return a value that represents how much light reaches this point.
- Run the playground, and you'll see ambient occlusion in all of its splendor.



*Ambient occlusion*

It would be useful to have a camera that moves around the scene. All it needs is a position, a ray that can be used as the camera's direction and a divergence factor which shows how much the ray spreads.

► In **Shaders.metal**, add a new structure with the other types:

```
struct Camera {
    float3 position;
    Ray ray{float3(0), float3(0)};
    float rayDivergence;
};
```

Here, you're setting up a camera using the **look-at** technique. This requires the camera to have a forward direction, an up direction and a left vector. If you're using a right-handed coordinate system, it's a right vector instead.

► Add this function before `compute`:

```
Camera setupCam(float3 pos, float3 target,
                float fov, float2 uv, int x) {
    // 1
    uv *= fov;
    // 2
    float3 cw = normalize(target - pos);
    // 3
    float3 cp = float3(0.0, 1.0, 0.0);
    // 4
    float3 cu = normalize(cross(cw, cp));
    // 5
    float3 cv = normalize(cross(cu, cw));
    // 6
    Ray ray = Ray{pos,
                  normalize(uv.x * cu + uv.y * cv + 0.5 * cw)};
    // 7
    Camera cam = Camera{pos, ray, fov / float(x)};
    return cam;
}
```

Going through the code:

1. Multiply the `uv` coordinates by the field of view.
2. Calculate a unit direction vector `cw` for the camera's forward direction.
3. The left vector will point orthogonally from an up and forward vector. `cp` is a temporary up vector.
4. The cross product gives you an orthogonal direction, so calculate the left vector `cu` using the forward and up vectors.

5. Calculate the correct up vector `cv` using the left and forward vectors.
6. Create a ray at the given origin with the direction determined by the left vector `cu` for the X-axis, by the up vector `cv` for the Y-axis and by the forward vector `cw` for the Z-axis.
7. Create a camera using the ray you created above. The third parameter is the ray divergence and represents the width of the cone. `x` is the number of pixels inside the field of view (e.g., if the view is 60 degrees wide and contains 60 pixels, each pixel is 1 degree). This is useful for speeding up the SDF when far away, and also for antialiasing.

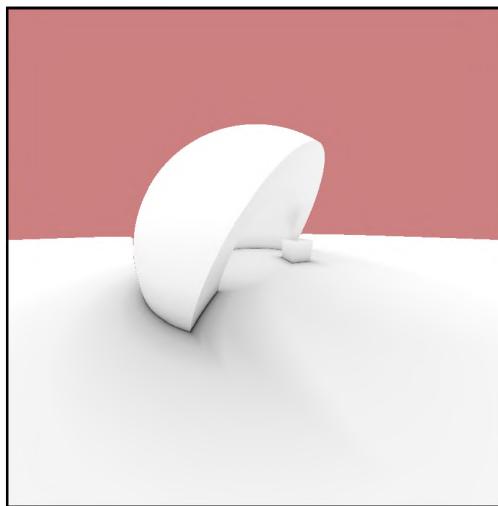
► To initialize the camera, replace this line in `compute`:

```
Ray ray = Ray{float3(0., 4., -12), normalize(float3(uv, 1.))};
```

► With this:

```
float3 camPos = float3(sin(time) * 10., 3., cos(time) * 10.);  
Camera cam = setupCam(camPos, float3(0), 1.25, uv, width);  
Ray ray = cam.ray;
```

Run the playground, and as the camera circles the scene, you can view the ambient occlusion from all directions.



*Camera circling the scene*

## Key Points

- Raymarching produces better quality shadows than rasterized shadows.
- Hard shadows are not realistic, as there are generally multiple light sources in the real world.
- Soft shadows give better transitions between areas in shadow and not.
- Ambient occlusion does not depend on scene lighting, but on neighboring geometry. The closer geometry is to an area, the darker the area is.

## Where to Go From Here?

In addition to the shadow types you learned in this chapter, there are other shadow techniques such as Screen Space Ambient Occlusion and Shadow Volumes. If you're interested in learning about these, review **references.markdown** in the resources folder for this chapter.



# Chapter 29: Advanced Lighting

As you've progressed through this book, you've encountered various lighting and reflection models:

- In Chapter 10, “Lighting Fundamentals” you started with the Phong reflection model which defines light as a sum of three distinct components: ambient light, diffuse light and specular light.
- In Chapter 11, “Maps & Materials” you briefly looked at physically based rendering and the Fresnel effect.
- In Chapter 21, “Image-Based Lighting” you implemented skybox-based reflection and image-based lighting, and you used a Bidirectional Reflectance Distribution Function (BRDF) look-up table.

In this chapter, you'll learn about **global illumination** and the famous **rendering equation** that defines it.



While reflection is possible using the local illumination techniques you've seen so far, advanced effects — like refraction, subsurface scattering, total internal reflection, caustics and color bleeding — are only possible with global illumination.



*A real-life example of global illumination and caustics*

You'll start by examining the rendering equation. From there, you'll move on to raymarched reflection and refraction.

## The Rendering Equation

Two academic papers — one by *James Kajiya*, and the other by *David Immel et al.* — introduced the rendering equation in 1986. In its raw form, this equation might look intimidating:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}) d\omega_i$$

*The rendering equation*

The rendering equation is based on the law of conservation of energy, and in simple terms, it translates to an **equilibrium equation** where the sum of all source lights must equal the sum of all destination lights:

incoming light + emitted light = transmitted light + outgoing light

If you rearrange the terms of the equilibrium equation, you get the most basic form of the rendering equation:

$$\text{outgoing light} = \text{emitted light} + \text{incoming light} - \text{transmitted light}$$

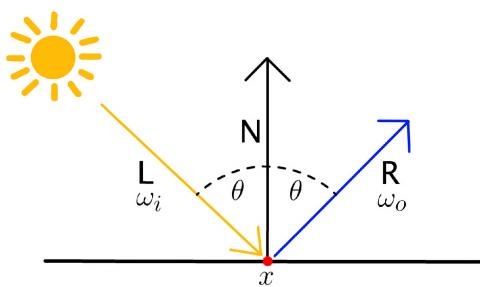
The incoming light – transmitted light part of the equation is subject to recursion because of multiple light bounces at that point. That recursion process translates to an integral over a unit hemisphere that's centered on the normal vector at the point and which contains all the possible values for the negative direction of the incoming light.

Although the rendering equation might be a bit intimidating, think of it like this: *All the light leaving an object is what remains from all the lights coming into the object after some of them were transmitted through the object.*

The transmitted light can be either absorbed by the surface of the object (material), changing its color; or scattered through the object, which leads to a range of interesting optical effects such as refraction, subsurface scattering, total internal reflection, caustics and so on.

## Reflection

Reflection, like any other optical phenomenon, has an equation that depends on three things: the incoming light vector, the incident angle and the normal vector for the surface.



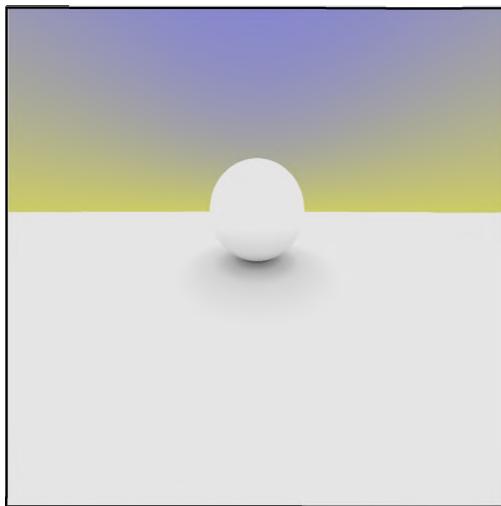
*Reflection*

The law of reflection states that the angle at which an incident light hits the surface of an object will be the same as the angle of the light that's being reflected off the normal.

But enough with the theory for now. Time to have some fun coding!

## Getting Started

- In Xcode, open the starter playground named **AdvancedLighting**, and select the **1. Reflection** playground page.
- Run the playground:



*The starter playground*

The code in this playground should look familiar to you because you've seen it in the two previous chapters. You'll continue as before, writing code in the metal shader for the relevant page.

You'll start by adding a checkerboard pattern to the plane, getting it to reflect onto the sphere.

## Drawing a Checkerboard Pattern

To draw a pattern on the plane, you first need to have a way of identifying objects within the scene by comparing their proximity to the camera based on distance.

- Inside the **Resources** folder for this playground page, open **Shaders.metal**, and create two constants to identify the two objects in the scene:

```
constant float PlaneObj = 0.0;
constant float SphereObj = 1.0;
```

- In `distToScene`, after this line:

```
float dts = distToSphere(r, s);
```

- Add this:

```
float object = (dtp > dts) ? SphereObj : PlaneObj;
```

Here, you check whether the distance to the plane is greater than the distance to the sphere, and you hold the result in `object`.

- Replace `return dist;` with:

```
return float2(dist, object);
```

You include both distance and object information in the function return.

- Run the playground to verify the image hasn't changed.

In `Shaders.metal`, the kernel function `compute` is where you're raymarching the scene. In a `for` loop, you iterate over a considerable number of samples and update the ray color until you attain enough precision. It's in this code block that you'll draw the pattern on the plane.

- In `compute`, inside the `for` loop, locate:

```
float2 dist = distToScene(cam.ray);
```

`distToScene` returns the closest object in `dist.y`.

- Immediately after that line, add this:

```
float closestObject = dist.y;
```

- After `hit = true;`, add this:

```
// 1
if (closestObject == PlaneObj) {
    // 2
    float2 pos = cam.ray.origin.xz;
    pos *= 0.1;
    // 3
    pos = floor(fmod(pos, 2.0));
    float check = mod(pos.x + pos.y, 2.0);
    // 4
    col *= check * 0.5 + 0.5;
}
```

Going through the code:

1. Build the checkerboard if the selected object is the plane.
2. Get the position of the camera ray in the horizontal XZ plane since you're interested in intersecting the floor plane only.
3. Create squares. You first alternate between 0s and 1s on both X and Z axes by applying the modulo operator.

At this point, you have a series of pairs containing either 0s or 1s or both. Next, add the two values together from each pair, and apply the modulo operator again.

If the sum is 2, roll it back to 0; otherwise, it will be 1.

4. Apply color. Initially, it's a solid white color. Multiply by 0.5 to tone it down, and add 0.5 back, so you can have both white and grey squares.

You have a compile error for the missing `mod` function. However, before adding the missing function, take a moment to understand why you need to implement a separate modulo operation.

The `fmod` function, as implemented by the Metal Shading Language, performs a **truncated division** where the remainder will have the same sign as the numerator:

```
fmod = numerator - denominator * trunc(numerator / denominator)
```

A second approach, missing from MSL, is known as **floored division**, where the remainder has the same sign as the denominator:

```
mod = numerator - denominator * floor(numerator / denominator)
```

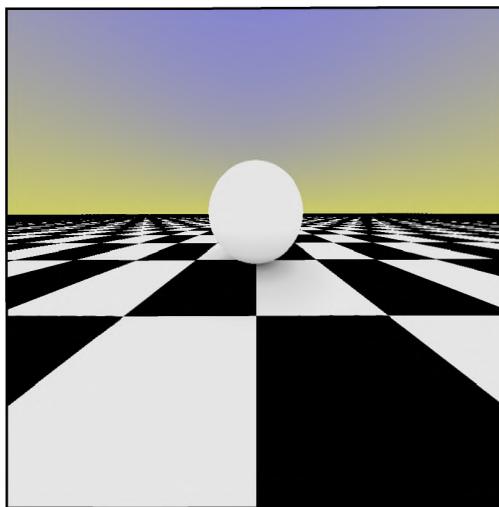
These two approaches could have entirely different results.

When calculating `pos`, the values need to alternate between 0s and 1s, so taking the `floor` of the truncated division is enough. However, when you add the two coordinates to determine the `check` value on the next line, you need to take the `floor` of their sum.

► Add the new floored division function above `compute`:

```
float mod(float x, float y) {  
    return x - y * floor(x / y);  
}
```

- Run the playground, and you'll see your checkerboard pattern.



*The checkerboard pattern*

All you need to do now is reflect the checkerboard onto the sphere.

- In `Shaders.metal`, add a new reflection function above `compute`:

```
Camera reflectRay(Camera cam, float3 n, float eps) {
    cam.ray.origin += n * eps;
    cam.ray.dir = reflect(cam.ray.dir, n);
    return cam;
}
```

The MSL standard library provides a `reflect()` function that takes the incoming ray direction and intersecting surface normal as arguments and returns the outgoing (reflected) ray direction. The `reflectRay` function is a convenience that returns the `Camera` object, not just its ray direction.

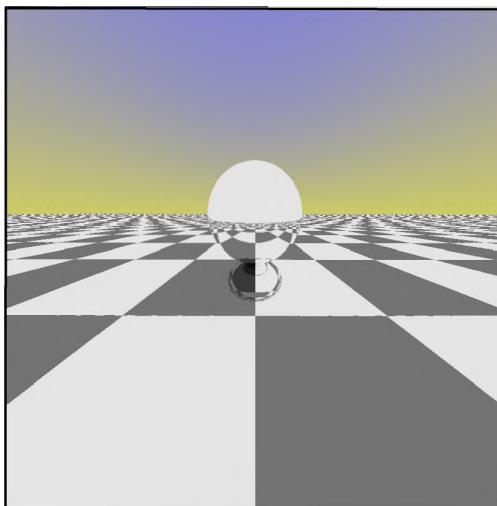
- In `compute`, *after* the `if (closestObject == PlaneObj)` block, but *inside* the `if (dist.x < eps)` block, add this:

```
float3 normal = getNormal(cam.ray);
cam = reflectRay(cam, normal, eps);
```

This code gets the normal where the camera ray intersects an object, and reflects it at that point. You move the ray away from the surface, along the normal and not along the ray direction as you might have expected because that could be almost parallel to the surface. You only move away a small distance `eps` that's precise enough to tell you when there's not a hit anymore.

The bigger `eps` is, the fewer steps you need to hit the surface, so the faster your tracing is – but it's also less accurate. You can play with various values for `eps` until you find a balance between precision and speed that satisfies your needs.

► Run the playground:



*Reflecting the checkerboard*

You're successfully reflecting the checkerboard onto the sphere, but the sky is not reflecting. This is because in the starter code you used the Boolean `hit`, which stops and breaks out of the loop when the ray first hits any object. That's not true anymore, because now you need the ray to keep hitting objects for reflection.

► Open **Shaders.metal**, and in `compute`, replace this code:

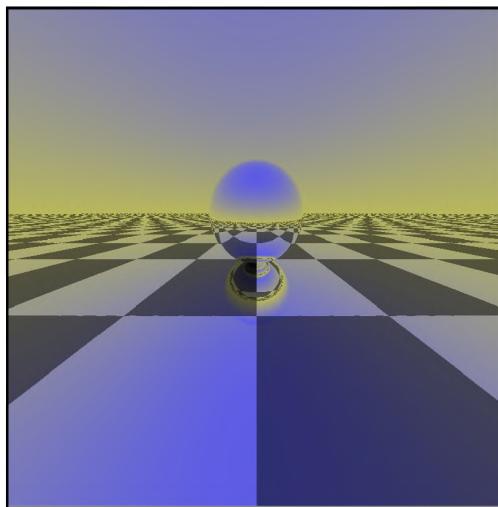
```
if (!hit) {
    col = mix(float3(.8, .8, .4), float3(.4, .4, 1.),
              cam.ray.dir.y);
} else {
    float3 n = getNormal(cam.ray);
    float o = ao(cam.ray.origin, n);
    col = col * o;
}
```

► With:

```
col *= mix(float3(0.8, 0.8, 0.4), float3(0.4, 0.4, 1.0),
          cam.ray.dir.y);
```

You add the sky color to the scene color globally, not just when a ray failed to hit an object in the scene. You can optionally remove the `ao` function and the two lines in `compute where hit` appears since you're not using them anymore.

- Run the playground, and you'll see the sky is now also reflected on the sphere and the floor.



*Reflecting the sky*

You can spin the camera a little bit to make the reflection look more interesting.

- In `Shaders.metal`, add this parameter to `compute`:

```
constant float &time [[buffer(0)]]
```

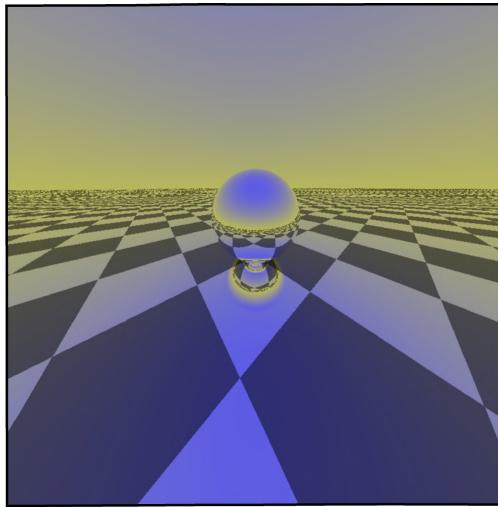
- And replace this line:

```
float3 camPos = float3(15.0, 7.0, 0.0);
```

- With this:

```
float3 camPos = float3(sin(time) * 15.0,  
                        sin(time) * 5.0 + 7.0,  
                        cos(time) * 15.0);
```

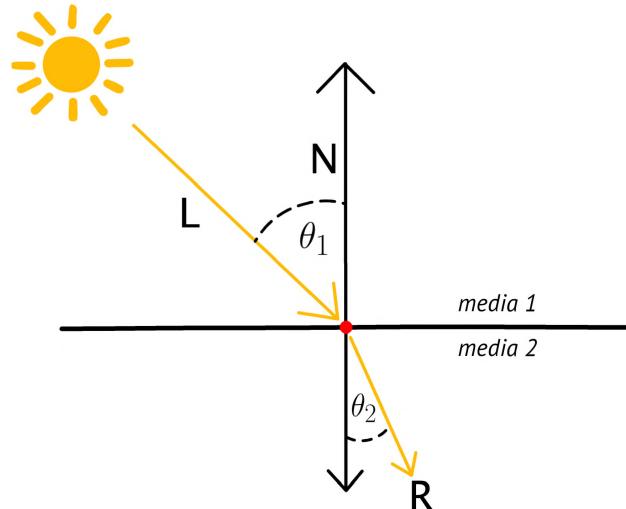
- Run the playground, and you'll see the same image but now nicely animated.



*Animated reflections*

## Refraction

The law of refraction is a little more complicated than simple equality between the incoming and outgoing light vector angles.



*Refraction*

Refraction is dictated by **Snell's law**, which states that the ratio of angles equals the reversed ratio of indices of refraction:

$$\frac{\sin \theta_2}{\sin \theta_1} = \frac{\text{IOR of media 1}}{\text{IOR of media 2}}$$

*Snell's law*

The **index of refraction (IOR)** is a constant that defines how fast light propagates through various media. IOR is defined as the speed of light in a vacuum divided by the phase velocity of light in that particular medium.

**Note:** There are published lists with IOR values for various media but the ones that interest us here are that of air ( $\text{IOR} = 1$ ) and that of water ( $\text{IOR} = 1.33$ ). See [https://en.wikipedia.org/wiki/List\\_of\\_refractive\\_indices](https://en.wikipedia.org/wiki/List_of_refractive_indices) for more details.

To find the angle for the refracted light vector through water, for example, all you need to know is the incoming light vector angle, which you can use from the reflected light vector. Then, you can divide that by the IOR for water since IOR for air is 1 and does not affect the calculation:

```
sin(theta2) = sin(theta1) / 1.33
```

Time for some more coding.

- Open the **2. Refraction** playground page, and run it.

The code is the same as the previous section, so you'll see the same animation.

- Inside the **Resources** folder for this playground page, open **Shaders.metal**.

You first need to have a way of knowing when the ray is inside the sphere, as you only do refraction in that case.

- In **compute**, add the following code before the **for** loop:

```
bool inside = false;
```



In the first part of this chapter, you identified objects, so you know when the ray hits the sphere. This means that you can change the sign of the distance depending on whether the ray enters the sphere, or leaves it. As you know from previous chapters, a negative distance means you're inside the object you are sending your ray towards.

- Locate:

```
float2 dist = distToScene(cam.ray);
```

- And, add this line below it:

```
dist.x *= inside ? -1.0 : 1.0;
```

This adjusts the `x` value to reflect whether you are inside the sphere or not. Next, you need to adjust the normals.

- Delete this line:

```
float3 normal = getNormal(cam.ray);
```

- Then, locate this line:

```
if (dist.x < eps) {
```

- After you find it, add the normal definition back into the code right below it:

```
float3 normal = getNormal(cam.ray) * (inside ? -1.0 : 1.0);
```

You now have a normal that points outward when outside the sphere and inward when you're inside the sphere.

- Move the following line so that it is inside the inner `if` block because you only want the plane to be reflective from now on:

```
cam = reflectRay(cam, normal, eps);
```

- After the inner `if` block, add an `else` block where you make the sphere refractive:

```
// 1
else if (closestObject == SphereObj) {
    inside = !inside;
    // 2
    float ior = inside ? 1.0 / 1.33 : 1.33;
    cam = refractRay(cam, normal, eps, ior);
}
```

Going through the code:

1. Check whether you're inside the sphere. On the first intersection, the ray is now inside the sphere, so turn `inside` to `true` and do the refraction. On the second intersection, the ray now leaves the sphere, so turn `inside` to `false`, and refraction no longer occurs.
2. Set the index of refraction (IOR) based on the ray direction. IOR for water is `1.33`. The ray is first going air-to-water, then it's going water-to-air in which case the IOR becomes `1 / 1.33`.

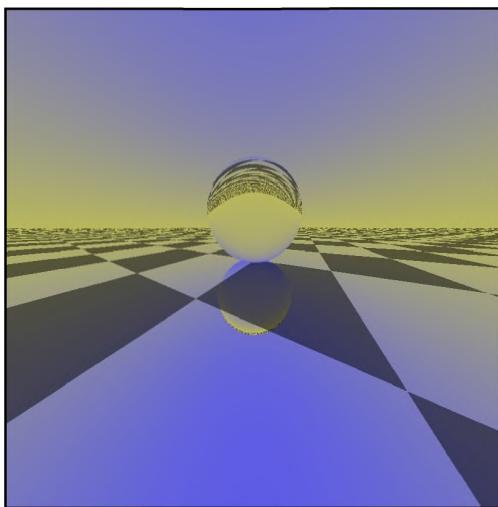
- To fix the compile error currently being shown by Xcode, add this missing function above `compute`:

```
Camera refractRay(Camera cam, float3 n, float eps, float ior) {
    cam.ray.origin -= n * eps * 2.0;
    cam.ray.dir = refract(cam.ray.dir, n, ior);
    return cam;
}
```

The MSL standard library also provides a `refract()` function, so you're just building a convenience function around it. You subtract the distance this time because the ray is inside the sphere.

You also double the `eps` value, which is enough to move far enough inside to avoid another collision. If `eps` were still the old value, the ray might stop and consider it another collision with the object since `eps` was defined precisely for this purpose: precision. Doubling it will make the ray pass just over the point that was already a collision point before.

- Run the playground, and you'll see the sphere now being refractive.



## Raytraced Water

It's relatively straightforward to create a cheap, fake water-like effect on the sphere.

- Open the **3. Water** playground page, and run it. You'll see the same animation from the previous section.
- Inside the **Resources** folder for this playground page, open **Shaders.metal**.
- In the **distToScene** function, locate:

```
float object = (dtp > dts) ? SphereObj : PlaneObj;
```

- And add this code afterward:

```
if (object == SphereObj) {  
    // 1  
    float3 pos = r.origin;  
    pos += float3(sin(pos.y * 5.0),  
                  sin(pos.z * 5.0),  
                  sin(pos.x * 5.0)) * 0.05;  
    // 2  
    Ray ray = Ray{pos, r.dir};  
    dts = distToSphere(ray, s);  
}
```

Going through the code:

1. Get the ray's current position, and apply ripples to the surface of the sphere by altering all three coordinates. Use `0.05` to attenuate the altering. A value of `0.001` is not large enough to make an impact, while `0.01` is too much of an impact.
2. Construct a new ray using the altered position as the new ray origin while preserving the old direction. Calculate the distance to the sphere using this new ray.

► In `compute`, replace:

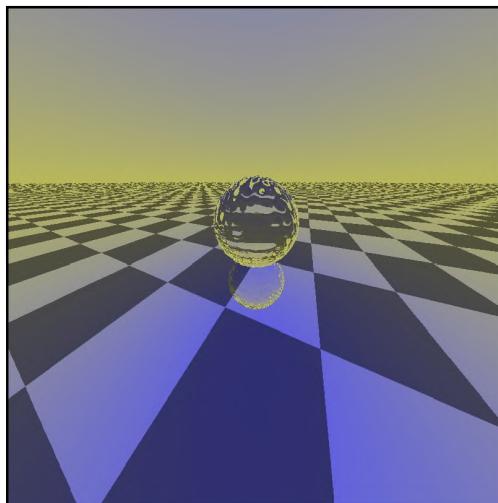
```
cam.ray.origin += cam.ray.dir * dist.x;
```

► With:

```
cam.ray.origin += cam.ray.dir * dist.x * 0.5;
```

You added an attenuation factor of `0.5` to make the animation slower but more precise.

► Run the playground, and you'll see a water-like ball.



## Key Points

- The rendering equation is the gold standard of realistic rendering. It describes conservation of energy where the sum of incoming light must equal outgoing light.
- Reflection depends on the angle of the incoming light and the surface normal.
- Refraction takes into account the medium's index of refraction, which defines the speed at which light travels through the medium.

## Where to Go From Here?

If you want to explore more about water rendering, the **references.markdown** file for this chapter contains links to interesting articles.

This concludes the series of chapters using raymarching. But don't worry, rendering is far from over. In the next chapters, you'll dip your toes into image processing and learn about using Metal for accelerating ray tracing.

# Chapter 30: Metal Performance Shaders

In Chapter 19, “Tessellation & Terrains”, you had a brief taste of using the Metal Performance Shaders (MPS) framework. MPS consists of low-level, fine-tuned, high-performance kernels that run off the shelf with minimal configuration. In this chapter, you’ll dive a bit deeper into the world of MPS.



# Overview

The MPS kernels make use of data-parallel primitives that are written in such a way that they can take advantage of each GPU family's characteristics. The developer doesn't have to care about which GPU the code needs to run on, because the MPS kernels have multiple versions of the same kernel written for every GPU you might use. Think of MPS kernels as convenient black boxes that work efficiently and seamlessly with your command buffer. Simply give it the desired effect, a source and destination resource (buffer or texture), and then encode GPU commands on the fly!

## The Sobel Filter

The Sobel filter is a great way to detect edges in an image.

- In the **starter** folder for this chapter, open and run **sobel.playground**, and you'll see such an effect (left: original image, right: Sobel filter applied):



*The Sobel filter*

Assuming you've already created a device object, a command queue, a command buffer and a texture object for the input image, there are only a few lines of code to apply the Sobel filter to your input image:

```
let shader = MPSImageSobel(device: device)
shader.encode(
    commandBuffer: commandBuffer,
    sourceTexture: inputImage,
    destinationTexture: drawable.texture)
```

MPS kernels are not thread-safe, so it's not recommended to run the same kernel on multiple threads that are all writing to the same command buffer concurrently.

Moreover, you should always allocate your kernel to only one device, because the kernel's `init(device:)` method could allocate resources that are held by the current device and might not be available to another device.

**Note:** MPS kernels provide a `copy(with:device:)` method that allows them to be copied to another device.

The MPS framework serves a variety of purposes:

- Image processing
- Matrix / vector mathematics
- Neural Networks

**Note:** The Metal Performance Shaders for ray tracing has been replaced by a newer ray tracing API.

In this chapter, you'll mainly focus on image processing, with a brief look at matrix mathematics.

## Image Processing

There are a few dozen MPS image filters, among the most common being:

- Morphological (area min, area max, dilate, erode).
- Convolution (median, box, tent, Gaussian blur, Sobel, Laplacian, and so on).
- Histogram (histogram, histogram equalization, histogram specification).
- Threshold (binary, binary inverse, to zero, to zero inverse, and so on).
- Manipulation (conversion, Lanczos scale, bilinear scale, transpose).

**Note:** For a complete list of MPS kernels, consult Apple's official Image Filters page ([https://developer.apple.com/documentation/metalperformanceshaders/image\\_filters](https://developer.apple.com/documentation/metalperformanceshaders/image_filters)). If you want to create your own filters, you can get inspired from Gimp's list of filters (<https://docs.gimp.org/en/filters.html>).

An RGB image is nothing but a matrix with numbers between 0 and 255 (when using 8-bit color channels). A greyscale image only has one such matrix because it only has one channel. For color images, there are three separate RGB channels (red, green, blue), so consequently three matrices, one for each channel.

One of the most important operations in image processing is **convolution**, which is an operation consisting of applying a much smaller matrix, often called the kernel, to the original image and obtaining the desired effect as a result.

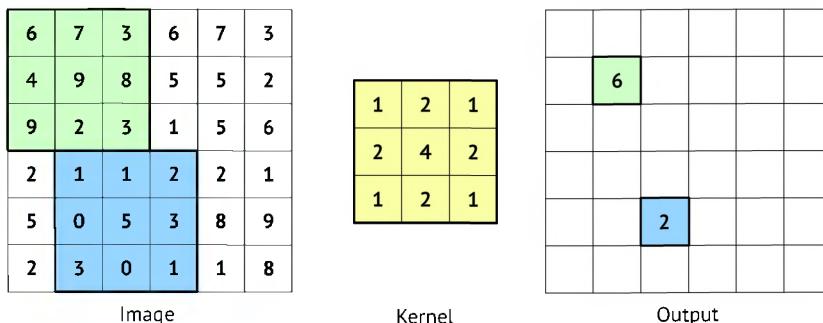
As an example, this matrix is used for obtaining Gaussian blur:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

*A Gaussian blur matrix*

**Note:** You can find a list of common kernels at [https://en.wikipedia.org/wiki/Kernel\\_\(image\\_processing\)](https://en.wikipedia.org/wiki/Kernel_(image_processing))

Here's a diagram showing how the kernel is applied to two pixels:



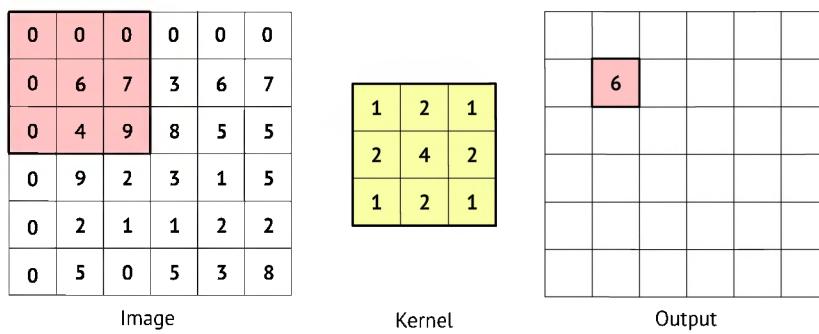
### Convolution

And, here's how the result shown in green was calculated:

```
(6 * 1 + 7 * 2 + 3 * 1 +
 4 * 2 + 9 * 4 + 8 * 2 +
 9 * 1 + 2 * 2 + 3 * 1) / 16 = 6
```

In this example, 16 represents the weight, and it's not a randomly chosen number — it's the sum of the numbers from the convolution kernel matrix.

When you need to apply convolution to the border pixels, you can apply padding to the input matrix. For example, when the center of a  $3 \times 3$  convolution kernel overlaps with the image element at position  $(0, 0)$ , the image matrix needs to be padded with an extra row and extra column of zeros. However, if the bottom rightmost element of the convolution kernel overlaps with the image element at position  $(0, 0)$ , the image matrix needs to be padded with two extra rows and two extra columns of zeros.



*Convolution applied to border pixels*

Applying the convolution kernel to an image matrix padded with an extra row, and extra column of zeros, gives you this calculation for a  $3 \times 3$  kernel:

$$(0 * 1 + 0 * 2 + 0 * 1 + \\ 0 * 2 + 6 * 4 + 7 * 2 + \\ 0 * 1 + 4 * 2 + 9 * 1) / 9 = 6$$

In this case, the weight, 9, is the sum of the numbers from the convolution kernel matrix that are affecting only the non-zero numbers from the image matrix ( $4 + 2 + 2 + 1$ ). Something like this is straightforward, but when you need to work with larger kernels and multiple images that need convolution, this task might become non-trivial.

You know how to calculate by hand and apply convolution to an image — and at the very beginning of the chapter you saw an MPS filter on an image too — but how about using MPS in your engine?

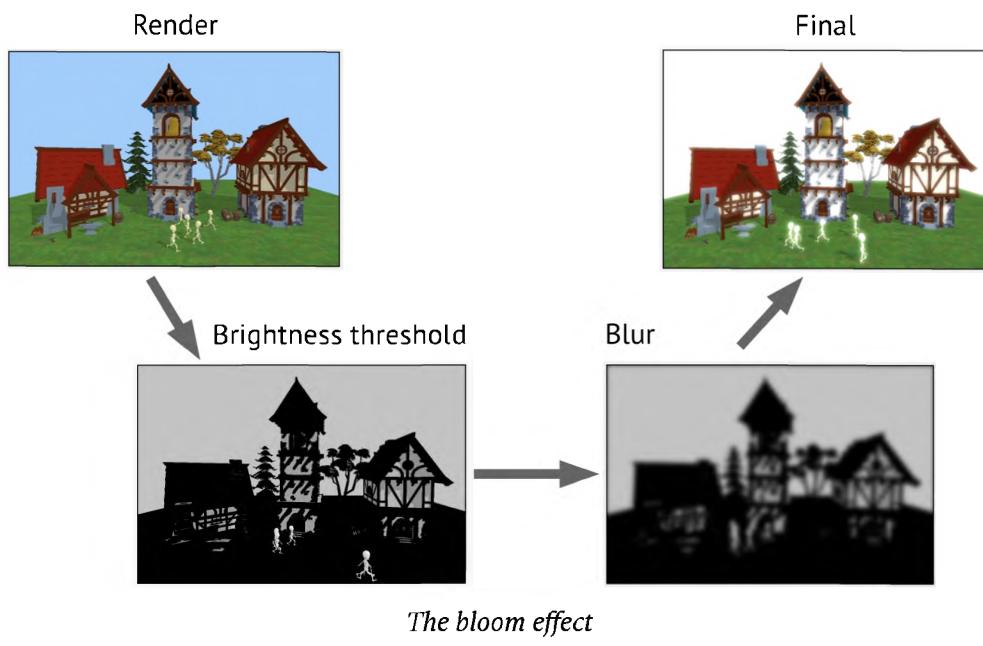
What if you were to implement **bloom** in your engine?

Guess what? You are going to do just that next!

## Bloom

The bloom effect is quite a spectacular one. It amplifies the brightness of objects in the scene and makes them look luminous as if they're emitting light themselves.

Below is a diagram that gives you an overview of how to achieve bloom:



Here are the steps you're going to take:

- Render the entire scene to a texture.
- Apply a threshold filter to this texture. This will amplify the lighter parts of the image, making them brighter.
- Apply a blur filter to the threshold texture from the previous step.
- Combine this texture with the initial scene for the final image.

# The Starter Project

- In Xcode, open the starter project for this chapter and build and run the app.



The scene is the same as Chapter 25, “Managing Resources”, complete with marching skeletons.

The starter project has a new **Post Processing** group, already set up with **Bloom.swift** and **Outline.swift**. In these files, you’ll add some post processing filters to the final image.

In the **Game** group, open **Renderer.swift**, and locate `// Post processing` in `draw(scene:in:)`. **Renderer** initializes **bloom** and **outline**, and depending on the option that the user chooses, runs a post processing effect.

The project currently renders to the view’s drawable texture. Instead of sending this texture straight to the screen, you’ll intercept and use the drawable texture as input to the threshold filter.

Currently, when you choose **Bloom** or **Outline** in the app, a print statement goes to the debug console.

## Setting Up the Textures

- In the **Post Processing** group, open **Bloom.swift**, and import the MPS framework:

```
import MetalPerformanceShaders
```

- Define two textures at the top of **Bloom**:

```
var outputTexture: MTLTexture!
var finalTexture: MTLTexture!
```

`outputTexture` will hold the blurred threshold texture, and `finalTexture` will hold this texture combined with the initial render.

Renderer calls `resize(view:size:)` from `mtkView(_:drawableSizeWillChange:)` whenever the window resizes, so this is where you'll create the textures.

- Add the following code to `resize(view:size:)`:

```
outputTexture = TextureController.makeTexture(
    size: size,
    pixelFormat: view.colorPixelFormat,
    label: "Output Texture",
    usage: [.shaderRead, .shaderWrite])
finalTexture = TextureController.makeTexture(
    size: size,
    pixelFormat: view.colorPixelFormat,
    label: "Final Texture",
    usage: [.shaderRead, .shaderWrite])
```

You create the two textures. Later, you'll use them as the destinations of MPS filters, so you mark them as writeable.

## Image Threshold to Zero

The Metal Performance Shader `MPSImageThresholdToZero` is a filter that returns either the original value for each pixel having a value greater than a specified brightness threshold or `0`. It uses the following test:

```
destinationColor =
sourceColor > thresholdValue ? sourceColor : 0
```

This filter has the effect of making darker areas black, while the lighter areas retain their original color value.

- In `postProcess(view:commandBuffer:)`, replace `print("Post processing: Bloom")` with:

```
guard
    let drawableTexture =
        view.currentDrawable?.texture else { return }
let brightness = MPSImageThresholdToZero(
    device: Renderer.device,
    thresholdValue: 0.5,
    linearGrayColorTransform: nil)
brightness.label = "MPS brightness"
brightness.encode(
    commandBuffer: commandBuffer,
    sourceTexture: drawableTexture,
    destinationTexture: outputTexture)
```

Here, you create an MPS kernel to create a threshold texture with a custom brightness threshold set to `0.5` — where all pixels with less than a color value of `0.5` will be turned to black. The input texture is the view’s drawable texture, which contains the current rendered scene. The result of the filter will go into `outputTexture`. Internally, the MPS kernel samples from `drawableTexture`, so you have to set the view’s drawable to be used for read/write operations.

- Open `Renderer.swift`, and add this to the end of `init(metalView:options:)`:

```
metalView.framebufferOnly = false
```

Metal optimizes `drawable` as much as possible, so setting `framebufferOnly` to `false` will affect performance slightly.

To be able to see the result of this filter, you’ll blit `outputTexture` back into `drawable.texture`. You should be familiar with the blit command encoder from when you copied textures to the heap in Chapter 25, “Managing Resources”

## The Blit Command Encoder

- Open `Bloom.swift`, and add this to the end of `postProcess(view:commandBuffer:)`:

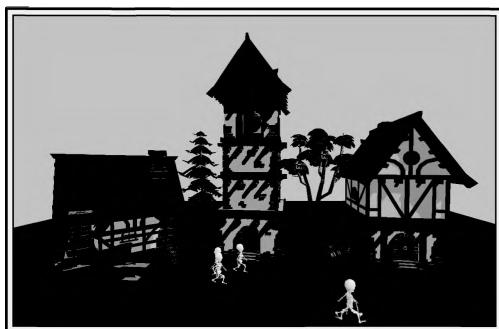
```
finalTexture = outputTexture
guard let blitEncoder = commandBuffer.makeBlitCommandEncoder()
    else { return }
let origin = MTLOrigin(x: 0, y: 0, z: 0)
let size = MTLSize(
    width: drawableTexture.width,
```

```
height: drawableTexture.height,
depth: 1)
blitEncoder.copy(
    from: finalTexture,
    sourceSlice: 0,
    sourceLevel: 0,
    sourceOrigin: origin,
    sourceSize: size,
    to: drawableTexture,
    destinationSlice: 0,
    destinationLevel: 0,
    destinationOrigin: origin)
blitEncoder.endEncoding()
```

This copies the output of the previous filter into the drawable texture. Unlike when you copied the textures to the heap, you don't have to worry about slices and mipmap levels here.

- Build and run the app, and select **Bloom**.

You'll now see the texture filtered to grayscale.



*Brightness threshold*

Notice how only some of the rendered areas were bright enough to make it to this texture. These white areas are all you need to create the bloom effect. Before using this texture, you need to add a little fuzziness to it which will make the model edges appear to glow. You can accomplish this with another MPS kernel: the Gaussian blur.

## Gaussian Blur

`MPSImageGaussianBlur` is a filter that convolves an image with a Gaussian blur with a given sigma value (the amount of blur) in both the X and Y directions.

- Still in **Bloom.swift**, in `postProcess(view:commandBuffer:)`, add the following prior to `finalTexture = outputTexture`:

```
let blur = MPSImageGaussianBlur(  
    device: Renderer.device,  
    sigma: 9.0)  
blur.label = "MPS blur"  
blur.encode(  
    commandBuffer: commandBuffer,  
    inPlaceTexture: &outputTexture,  
    fallbackCopyAllocator: nil)
```

**In-place encoding** is a special type of encoding where, behind the curtains, the input texture is processed, stored to a temporary texture and finally written back to the input texture without the need for you to designate an output texture.

The `fallbackCopyAllocator` argument allows you to provide a closure where you can specify what will happen to the input image should the in-place normal encoding fail.

- Build and run the app, and choose **Bloom**. You'll see the result of this blur.



*Brightness and blur*

## Image Add

The final part of creating the bloom effect is to add the pixels of this blurred image to the pixels of the original render.

`MPSImageArithmetic`, as its name suggests, performs arithmetic on image pixels. Subclasses of this include `MPSImageAdd`, `MPSImageSubtract`, `MPSImageMultiply` and `MPSImageDivide`.

Adding the rendered scene pixels to the lighter blurred pixels will brighten up those parts of the scene. In contrast, adding them to the black pixels will leave them unchanged.

- In `postProcess(view:commandBuffer:)`, replace `finalTexture = outputTexture` with this:

```
let add = MPSImageAdd(device: Renderer.device)
add.encode(
    commandBuffer: commandBuffer,
    primaryTexture: drawableTexture,
    secondaryTexture: outputTexture,
    destinationTexture: finalTexture)
```

This code adds the drawable texture to `outputTexture` and places the result in `finalTexture`.

- Build and run the app, and choose **Bloom**. You'll see this:



*Brightness, blur and add*

The entire scene is bathed in a mystical glow. Awesome bloom!

- In `postProcess(view:commandBuffer:)`, change the initialization of `brightness` to:

```
let brightness = MPSImageThresholdToZero(
    device: Renderer.device,
    thresholdValue: 0.8,
    linearGrayColorTransform: nil)
```

Fewer pixels will make it through the brightness filter.

- Build and run the app, and choose **Bloom**.



*Glowing skeletons*

Because the skeletons are the brightest objects in the scene, they appear to glow spookily.

## Matrix / Vector Mathematics

You learned in the previous section how you could quickly apply a series of MPS filters that are provided by the framework. But what if you wanted to make your own filters?

You can create your own filter functions and calculate convolutions yourself. However, when working with large matrices and vectors, the amount of math involved might get overwhelming.

The MPS framework not only provides image processing capability, but it also provides functionality for decomposition and factorizing matrices, solving systems of equation and multiplying matrices and/or vectors on the GPU in a fast, highly parallelized fashion. You're going to look at matrix multiplication next.

- Create a new empty playground for macOS named **matrix.playground**.
- Replace the code with:

```
import MetalPerformanceShaders

guard let device = MTLCreateSystemDefaultDevice(),
      let commandQueue = device.makeCommandQueue()
else { fatalError() }

let size = 4
let count = size * size
```

```

guard let commandBuffer = commandQueue.makeCommandBuffer()
else { fatalError() }

commandBuffer.commit()
commandBuffer.waitUntilCompleted()

```

This code creates a Metal device, command queue, command buffer and adds a couple of constants you'll need later.

- Above the line where you create the command buffer, add a new method that lets you create MPS matrices:

```

func createMPSMatrix(withRepeatingValue: Float) -> MPSMatrix {
    // 1
    let rowBytes = MPSMatrixDescriptor.rowBytes(
        forColumns: size,
        dataType: .float32)
    // 2
    let array = [Float](
        repeating: withRepeatingValue,
        count: count)
    // 3
    guard let buffer = device.makeBuffer(
        bytes: array,
        length: size * rowBytes,
        options: [])
    else { fatalError() }
    // 4
    let matrixDescriptor = MPSMatrixDescriptor(
        rows: size,
        columns: size,
        rowBytes: rowBytes,
        dataType: .float32)

    return MPSMatrix(buffer: buffer, descriptor: matrixDescriptor)
}

```

Going through the code:

1. Retrieve the optimal number of bytes between one row and the next. Whereas SIMD matrices expect column-major order, MPSMatrix uses row-major order.
2. Create a new array and populate it with the value provided as an argument to this method.
3. Create a new buffer with the data from this array.
4. Create a matrix descriptor; then create the MPS matrix using this descriptor and return it.



Use this new method to create and populate three matrices. You'll multiply A and B together, and you'll place the result in C.

- Add the following code just before creating the command buffer:

```
let A = createMPSMatrix(withRepeatingValue: 3)
let B = createMPSMatrix(withRepeatingValue: 2)
let C = createMPSMatrix(withRepeatingValue: 1)
```

- Add this code to create a MPS matrix multiplication kernel:

```
let multiplicationKernel = MPSMatrixMultiplication(
    device: device,
    transposeLeft: false,
    transposeRight: false,
    resultRows: size,
    resultColumns: size,
    interiorColumns: size,
    alpha: 1.0,
    beta: 0.0)
```

- Below the line where you create the command buffer, add the following code to encode the kernel:

```
multiplicationKernel.encode(
    commandBuffer: commandBuffer,
    leftMatrix: A,
    rightMatrix: B,
    resultMatrix: C)
```

You multiply A and B together, and you place the result in C.

- At the very end of the playground, add this code to read C:

```
// 1
let contents = C.data.contents()
let pointer = contents.bindMemory(
    to: Float.self,
    capacity: count)
// 2
(0..
```

Going through the code:

1. Read the result back from the matrix C into a buffer typed to `Float`, and set a pointer to read through the buffer.
2. Create an array filled with the values from the buffer.

► Run the playground, and click **Show Result** on the last line.



```

67  (0...count).map {
68    pointer.advanced(by: $0).pointee
69  }
70

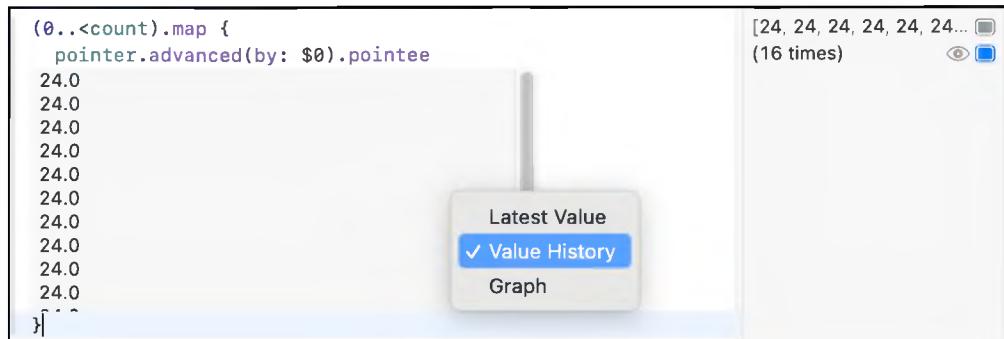
```

[24, 24, 24, 24, 24, 24...]  
(16 times)  

Show Result

*Show result*

► Right-click on the graph of the results, and choose “Value History”.



```

(0...count).map {
  pointer.advanced(by: $0).pointee
24.0
24.0
24.0
24.0
24.0
24.0
24.0
24.0
24.0
24.0
24.0
24.0
24.0
24.0
24.0
24.0
}

```

[24, 24, 24, 24, 24, 24...]  
(16 times)  

Latest Value  
✓ Value History  
Graph

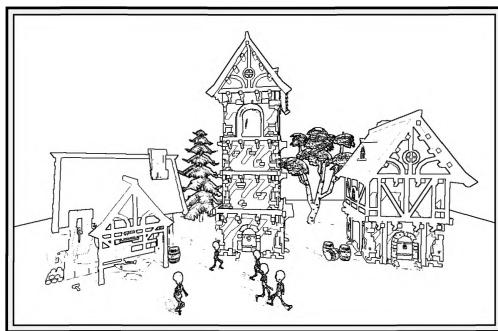
*Value history*

You'll see that the array contains 16 values, all of which are the number 24.0. That's because the matrix is of size 4×4, and multiplying one row of A with one column of B results in the value 24.0, which is 2×3 added four times.

This is only a small matrix, but you can change the size of the matrix in the `size` variable at the top of the playground, and the matrix multiplication will still be blisteringly fast.

## Challenge

You may have noticed that in the app where you did the bloom post processing, the **Outline** option does nothing. Your challenge is to fill out **Outline.swift** so that you have an outline render:



*Outline*

To achieve this, you'll first use `MPSImageSobel(device:)`, and then feed the output of the sobel filter to

`MPSImageThresholdBinaryInverse(device:thresholdValue:maximumValue:linearGrayColorTransform:)`.

If you have any difficulties, you can review the answer in the challenge folder for this chapter.

## Key Points

- Metal Performance Shaders are compute kernels that are performant and easy to use.
- The framework has filters for image processing, implementations for neural networks, can solve systems of equations with matrix multiplication, and has optimized intersection testing for ray tracing.
- Convolution takes a small matrix and applies it to a larger matrix. When applied to an image, you can blur or sharpen or distort the image.
- Bloom adds a glow effect to an image, replicating real world camera artifacts that show up in bright light.
- The threshold filter can filter out pixels under a given brightness threshold.

# Chapter 31: Performance Optimization

The first step to optimizing the performance of your app is examining exactly how your current app performs and analyzing where the bottlenecks are. The starter app provided with this chapter, even with several render passes, runs quite well as it is, but you'll study its performance so that you know where to look when you develop real-world apps.



## The Starter App

- In Xcode, build and run the starter app for this chapter.



*The starter app*

There are several render passes involved:

- **ShadowRenderPass**: Renders models to depth texture.
- **ForwardRenderPass**: Renders all models aside from rocks and grass.
- **NatureRenderPass**: Renders rocks and grass.
- **SkyboxRenderPass**: Renders the skybox.
- **Bloom**: Post processes the image with bloom.

You may find that the app runs very slowly. On my 2018 11" iPad Pro, it runs at 33 FPS. This is mostly due to the number of skeletons and quantity of grass. If your app runs too slowly, you can reduce these in GameScene.

## Profiling

There are a few ways to monitor and tweak your app's performance. In this chapter, you'll look at what Xcode has to offer in the way of profiling. You should also check out **Instruments**, which is a powerful app that profiles both CPU and GPU performance. For further information, read Apple's article Using Metal System Trace in Instruments to Profile Your App (<https://apple.co/36Qs9FY>).

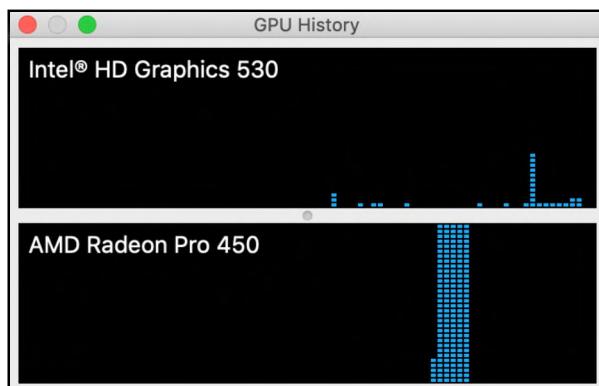
## GPU History

GPU history is a tool provided by the macOS operating system via its Activity Monitor app, so it is not inside Xcode. It shows basic GPU activity in real time for all of your GPUs. If you’re using eGPUs, it’ll show activity in there too.

► Open **Activity Monitor**, and from the **Window** menu, choose **GPU History**.

A window will pop up containing separate graphs for each GPU, showing the GPU usage in real time. You can change how often the graph is updated from the **View ▾ Update Frequency** menu. The graph moves right-to-left at the frequency rate you set.

Here’s a screenshot taken from a MacBook Pro that has a discrete GPU — AMDRadeon Pro 450 — and an integrated one — Intel HD Graphics 530:



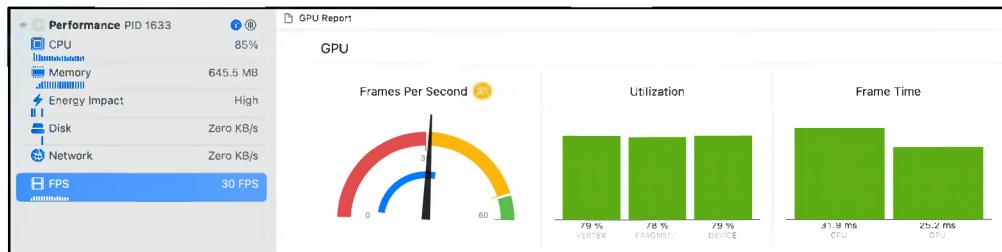
*GPU History*

The system is using the integrated Intel GPU for regular tasks, and it switches to the discrete AMD GPU when running a graphics-intensive task such as this Xcode project you’re working on.

The GPU History tool offers a quick way to see overall GPU usage, but it’s not helpful with showing GPU usage for individual running apps and processes.

## The GPU Report

- With your app running, in Xcode on the **Debug navigator**, click **FPS**.



*The GPU report*

The GPU report shows in the central pane and contains three major GPU metrics.

The first GPU report metric is **Frames Per Second**, and represents the current frame rate of your app. Your target should always be 60 FPS or better. The screenshot shows an app running on a 2018 iPad Pro. Some compromises on the number of objects displayed will have to be made to get it to run at 60 FPS.

The second GPU report metric is **Utilization**, which shows how busy your GPU is doing useful work. A healthy app will have the GPU always utilized to some extent. Having it sit idle might be an indication that the CPU has not given it enough work to do.

The third GPU report metric is **Frame Time**, and represents the actual time spent processing the current frame on the CPU and the GPU. What's most important here is that the frame does not take longer than 16.6ms which corresponds to 60 FPS.

Your GPU is not sitting idle, but the frame time is far too high, with more time spent on the CPU than the GPU.

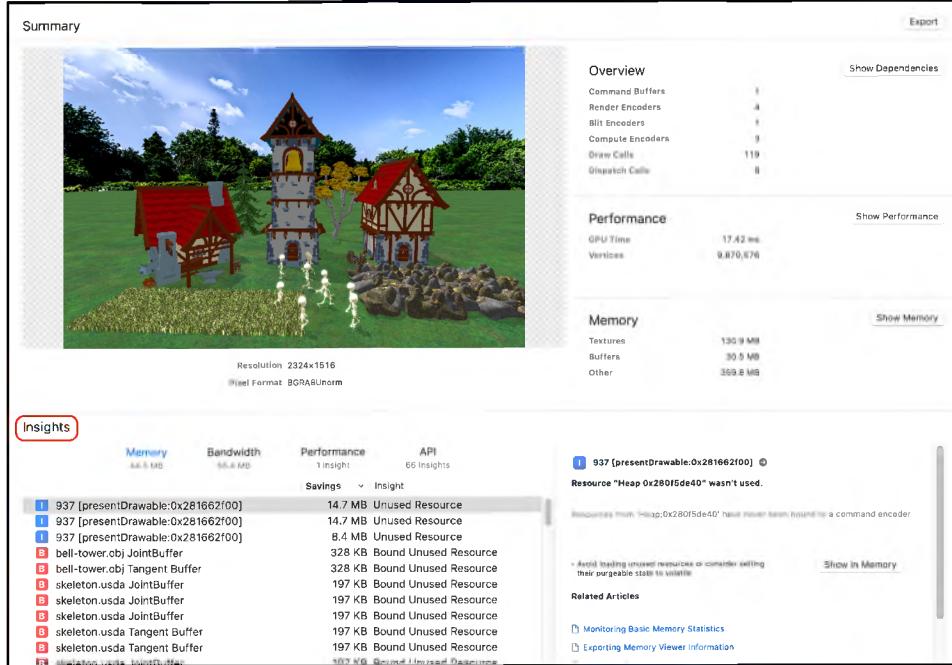
## GPU Workload Capture

In previous chapters, you captured the GPU workload to inspect textures, buffers and render passes. The GPU capture is always the first point of call for debugging. Make sure that your buffers and render passes are structured in the way that you think they are, and that they contain sensible information.

Next, you'll look at what else GPU capture can show you.

## Summary

- With your app running, capture the GPU workload, and in the **Debug navigator**, click **Summary**.



### The summary of your frame

You'll see an overview of your frame. The insights section often contains useful insights when you might bind resources on the CPU, but not use them in your shaders. The previous image shows a number of bound unused resources, most noticeably, the **Tangent Buffers**.

**Note:** To take full advantage of the GPU capture, you should add a label to all your buffers, so that you can easily track down issues. **Tangent Buffer** is a label added in **Mesh.swift**.

This insight highlights an error in your app. The app *should* be using the tangent buffer.

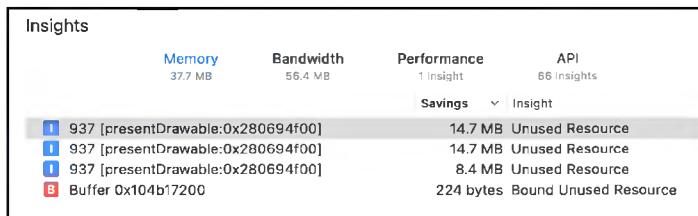
- In the **Shaders** group, open **Shaders.metal**, and locate the assignment to **worldTangent** and **worldBitangent**.

These assignments are currently set to 0, when they should be using the tangent values.

► Change the assignments to:

```
.worldTangent = uniforms.normalMatrix * in.tangent,  
.worldBitangent = uniforms.normalMatrix * in.bitangent,
```

► Build and run the app again, capture the GPU workload and check the Insights section.



#### *Insights into possible issues*

The insight issues for the tangent buffers have now gone away. The first three resources listed as unused are heaps created by the Metal Performance Shaders for the bloom effect, so there's nothing you can do about those.

**Note:** Depending on your device, you may see additional insights.

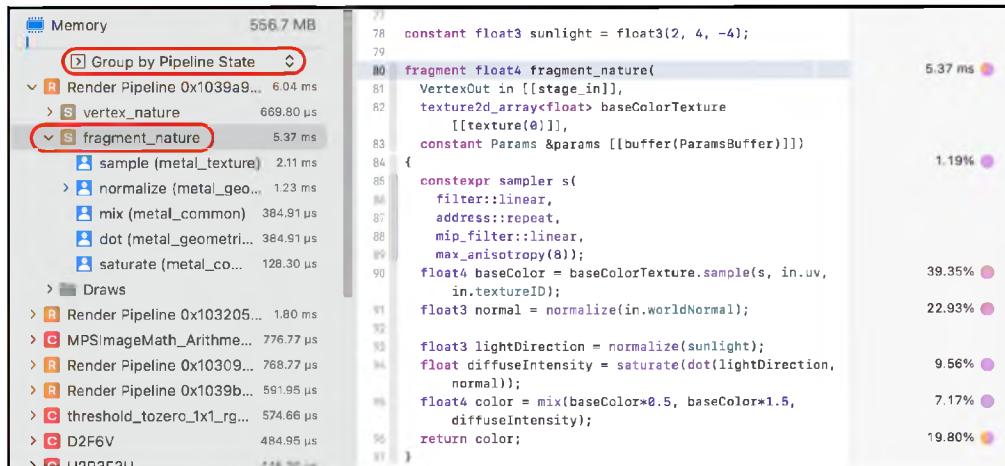
## The Shader Profiler

The shader profiler is perhaps the most useful profiling tool for the shader code you write. It has nothing to do with the rendering code the CPU is setting up, or the passes you run or the resources you're sending to the GPU. This tool tells you how your MSL code is performing line-by-line and how long it took to finish.

**Note:** The shader profiler shows you the individual line execution times (per-line cost) on iOS and Apple Silicon devices only.

► Build and run the app again — this time on a device with an Apple GPU — and capture the GPU workload.

- In the Debug navigator, switch to **Group by Pipeline State**.
- Open the top **Render Pipeline**, and open and select **fragment\_nature**.



*The shader profiler*

You can now see how much time each pipeline took during the frame.

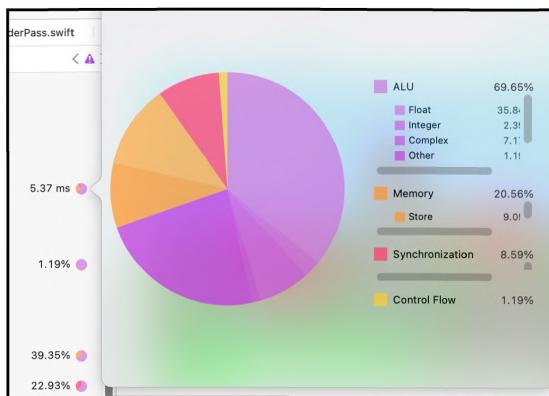
**Note:** Your times and percentages will vary depending on your iOS device, Xcode and iOS version.

The entire pipeline took **6.04ms** to complete. That time consists of **669.80** microseconds for the vertex shader and **5.37 milliseconds** for the fragment shader. That is a huge amount of time for one pipeline, so you might want to rethink how many blades of grass to render.

Drilling down the cost tree, most of the fragment shader execution time is taken by the **sample** function (2.11ms) and the **normalize()** function (1.23ms).

The total time that the shader takes to complete shows on the function header line. Inside the shader for each impactful line, you'll see the percentage the line took out of that total time. In the case of the **sample** function on line 90, that **2.11ms** correspond to **39.35%** of the **5.37ms** total shader time.

- Hover over the colored dot on the right of the function to disclose a pie chart with further information.



Pie Chart

Analyze the percentages for each GPU activity. A high number might indicate an opportunity for performance optimization.

Looking at the various GPU activities and their percentages, notice how the ALU took 69.65% of the total shader time processing the various data types and calculations involving them.

Here's the first opportunity for optimization using shader profiling. Notice that processing `floats` seems to take more time than processing other types. As you might know, a `half` is, well, half the size of a `float`, so you can optimize this one spot.

- Change all of the `floats` to `halfs` everywhere in `fragment_nature`, as well as in the `sunlight` definition line above the `fragment_nature`. (You'll have to do conversions such as `half3 normal = half3(normalize(in.worldNormal));`):

```
constant half3 sunlight = half3(2, 4, -4);

fragment half4 fragment_nature(
    VertexOut in [[stage_in]],
    texture2d_array<float> baseColorTexture [[texture(0)]],
    constant Params &params [[buffer(ParamsBuffer)]])
{
    constexpr sampler s(
        filter::linear,
        address::repeat,
        mip_filter::linear,
        max_anisotropy(8));
    half4 baseColor = half4(baseColorTexture.sample(s, in.uv,
        in.textureID));
```



```

half3 normal = half3(normalize(in.worldNormal));

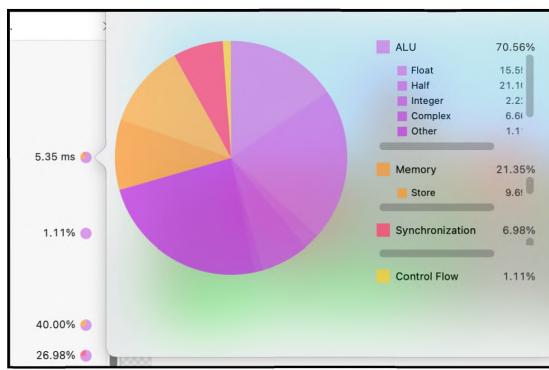
half3 lightDirection = normalize(sunlight);
half diffuseIntensity = saturate(dot(lightDirection, normal));
half4 color = mix(baseColor*0.5, baseColor*1.5,
diffuseIntensity);
return color;
}

```

- When done, click Reload Shaders.



The pie chart will also update in real time, as well as the Attachments contents if you also have the assistant editor open.



*Reloaded Pie Chart*

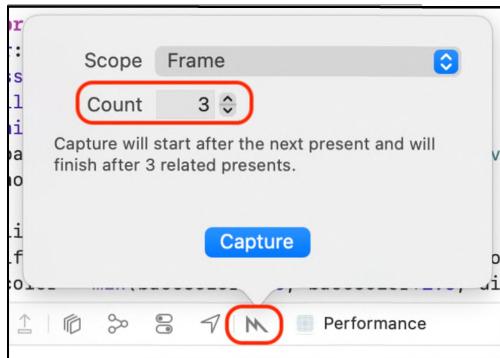
**Note:** Reloading the shader might not save your changes so make sure to take a note of what you changed and later update **Nature.metal** manually.

This function doesn't do a lot of processing, so you won't see much of an improvement, but the cost of processing `halfs` over `floats` is much less, and it's an easy change to make to your shader functions. Other ALU optimizations you can do includes replacing `ints` with `shorts`, simplifying complex instructions such as trigonometry functions (`sin`, `cos`, etc.) and other arithmetic calculations.

# GPU Timeline

The GPU timeline tool gives you an overview of how your vertex, fragment and compute functions perform, broken down by render pass.

- Build and run the app, and capture the GPU workload with a frame count of 3.

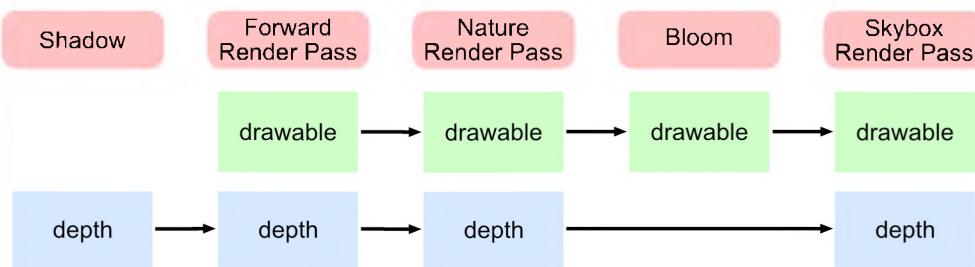


*Capture the GPU workload*

- In the **Debug navigator**, change **Group by Pipeline State** to **Group by API call**, and click on **Command Buffer**.

You'll see the dependency graph of the first frame, with each render pass texture showing how it's passed on from the previous render pass.

With a few of the Metal Performance passes removed, this is an overview of the dependency graph:



*Render Passes*

- In the Debug navigator, click on Performance.

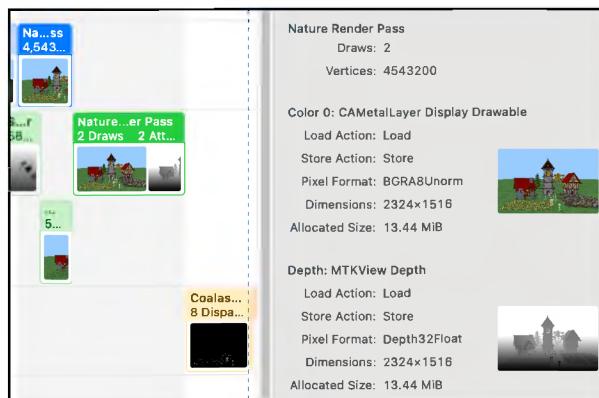


*The GPU timeline*

You'll see a track for each of your vertex, fragment and compute shaders, so you can visualize where in the timeline your shaders are performed, and how long they take to perform. The longest encoders are the Nature Render Pass and the Coalesced 3 Encoders. The orange compute shaders are the post processing Metal Performance Shaders producing the bloom effect. Now that you can see how long the bloom effect takes, you can reconsider whether it is worth the time spent on it.

On Apple GPUs, where you don't have a dependency, vertex, fragment and compute shaders can run in parallel. Unfortunately, when you look at your render passes, each render is dependent on the previous drawable, so there are a few gaps in the timeline.

- Click on each encoder in the timeline to see the encoder's attachments.

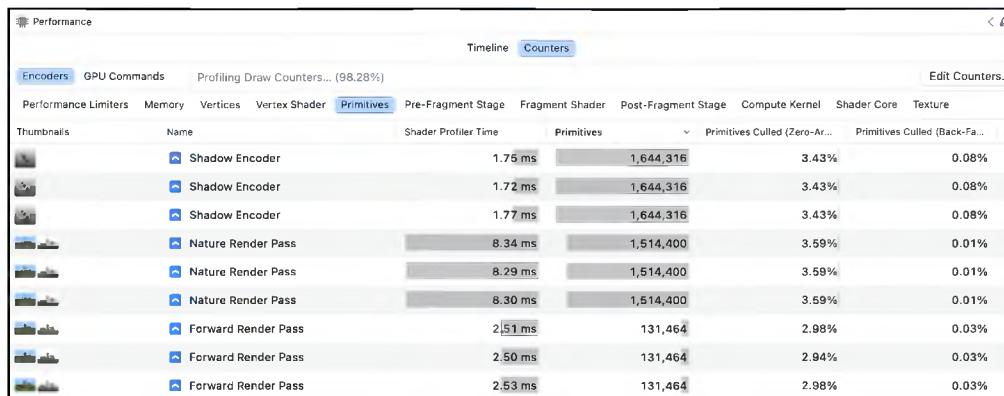


*Encoder attachments*

► Above the Timeline, select **Counters**.

There are about 150 GPU counters for you to investigate, giving precise timings and statistics. For more information on GPU counters, watch the following Apple WWDC videos:

- Explore Live GPU Profiling with Metal Counters (<https://apple.co/3HrNFxf>)
  - Optimize Metal apps and games with GPU counters (<https://apple.co/3pr1uWC>)
- In the **Counters** view, with **Encoders** selected, click on the **Primitives** column so that the column is sorted by the number of primitives in the render pass.



*GPU counters*

When rendering triangles in your draw calls, faces made up of three vertices are the **primitives** that the GPU counter report is referring to. An easy optimization is to cull unwanted faces. Currently, you're rendering everything, no matter whether the camera can see it or not. If you cull back faces, then only the faces pointing toward the camera will render.

You might think that you always want to cull back faces, but you do have to be a bit selective. For example, the tree leaves in your scene are a one-sided mesh, so if you cull the back faces, you won't see the leaves that are pointing away from you.

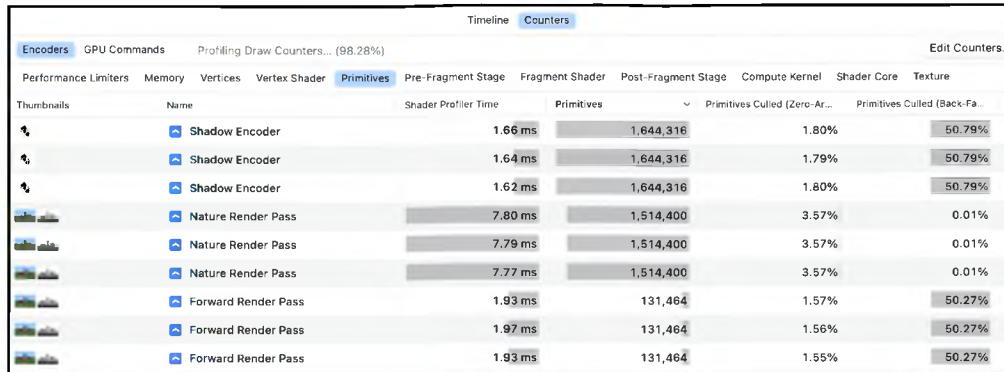
- In the **Game** group, open **Renderer.swift**, and change `static var cullFaces = false` to:

```
static var cullFaces = true
```

This change will trigger the culling already implemented in your starter app in all render passes, except `NatureRenderPass`.

- Build and run the app, capture the GPU workload and return to the Counters view above.

- Compare the result.



*Face culling implemented*

The affected shaders should run a little bit faster, now that you only render about half the primitives.

## Memory

- In the Debug navigator, click the **Memory** tool (below **Performance**) to see the total memory used and how the various resources are allocated in memory:



*Resources in memory*

You'll see how you can reduce your heap textures shortly.

Now that you know how to profile your app in Xcode, you can observe some flaws in your app, and fix a few of them.

## Instancing

Currently, you load ten skeleton meshes and draw them independently. The skeleton system could do with more efficient instanced drawing. Reducing the number of draw calls is one of the best ways of improving performance. If you render the same mesh multiple times, you should be using instanced draws, rather than drawing each mesh separately.

As an example of an instanced system, the app includes a procedural nature system. GameScene creates a rock pile of 200 rocks with three random shapes, and three random textures. It also creates a grassy patch with 50,000 grass blades, from four random shapes and seven random textures.

## The Procedural Nature System

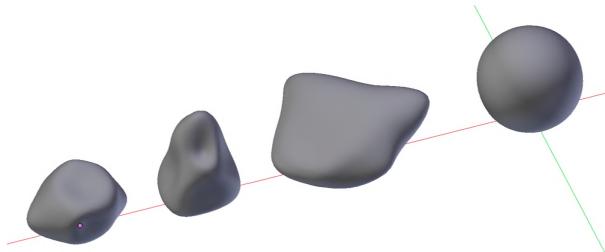
Using **homeomorphic** models, you can choose different shapes for each model. Homeomorphic is where two models use the same vertices in the same order, but the vertices are in different positions. A famous example of this is **Spot the cow** by Keenan Crane.



*Spot by Keenan Crane*

Spot is modeled from a sphere by moving vertices, rather than adding them. Because the vertices are in the same order as the sphere, the uv coordinates don't change either.

The random shapes for both rocks and grass are modeled in a similar fashion, using the same basic shape, then readjusting the vertices for each shape. Each adjusted shape is called a **morph target**.



*Homeomorphic rocks*

For the rocks, Nature loads the three vertex meshes into one buffer, and each rock, when initialized, is allocated a random number between 0 and 2. It's then simple to extract the correct mesh from the buffer in the vertex function.

The most important feature of the nature system is that, depending on how powerful your device is, it can render numerous instances with one draw call:

```
encoder.drawIndexedPrimitives(  
    type: .triangle,  
    indexCount: submesh.indexCount,  
    indexType: submesh.indexType,  
    indexBuffer: submesh.indexBuffer.buffer,  
    indexBufferOffset: submesh.indexBuffer.offset,  
    instanceCount: instanceCount)
```

In **Nature.metal**, `vertex_nature` uses the `instance_id` attribute to extract the transform information for the current instance. With the morph target, the `vertex` function renders a random shape. With the texture ID, the `fragment` function renders a random texture.

The files involved in the nature system are:

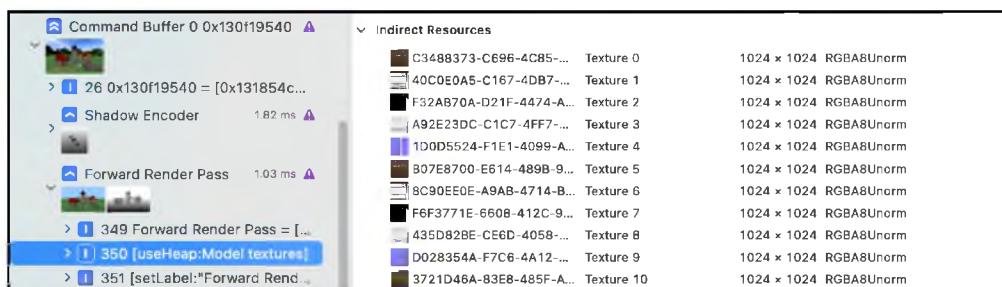
- **Common.h**: Contains a `NatureInstance` structure which holds a random texture and shape ID as well as the model and normal matrix.
- **Nature.swift**: This is in the **Geometry** group and is a cut-down version of `Model`. It loads up the mesh and creates a buffer that contains an array of `NatureInstance`, one element for each instance.

- **Nature.metal:** Contains the vertex and fragment functions.
  - **NatureRenderPass.swift:** Renders the scene's nature array, in the same way as ForwardRenderPass.
- Examine these files to see how the nature system works. You could create a skeleton system in the same way, that draws instanced skeletons.

## Removing Duplicate Textures

Textures use memory, and you should always check that you use the appropriate size for the device. The asset catalog makes this easy for you. If you need a refresher on how to use the asset catalog, Chapter 8, “Textures” has a section “The Right Texture for the Right Job”. However, you should also check that you aren’t duplicating textures.

- Build and run the app, and capture the GPU workload.
- Under **Command Buffer > Forward Render Pass**, select **useHeap**.



*The heap textures*

The indirect resources are the textures currently in the heap. One flaw in the app, is that all the barrel textures are loaded for each and every barrel. Because you render multiple barrels, they should be instanced, which is one way of fixing the problem, but there is another.

When the app loads the textures for the model `barrel.usdz` in `TextureController`, the textures aren’t allocated a name in the file, so each texture is given a unique UUID as the file name. However, if you load the model using the `obj` file format, the `mtl` file holds the file name, so `TextureController` is able to only load the texture for the first model.

► In the **Game** group, open **GameScene.swift**, and in **init()**, locate where you initialize barrels.

► Change the file name from **barrel.usdz** to:

**barrel.obj**

► Build and run the app, and you'll see that this already improves your frame rate.

► Capture the GPU workload and under **Command Buffer > Forward Render Pass**, select **useHeap**.



*A reduced heap*

The size of the heap is now significantly reduced, contributing to a substantial performance gain. Remember to optimize the simple things first, because you may discover that you need no further optimization.

You're in control of your engine. When you design your model loading process, ensure that the model structure fits your app. To get the best performance, you shouldn't be loading **obj** or **usdz** files at all. You should be loading all files from a file format that best suits your app's API. For further information about how you can do this, watch Apple's WWDC video From Art to Engine With Model I/O (<https://apple.co/3K948ls>).

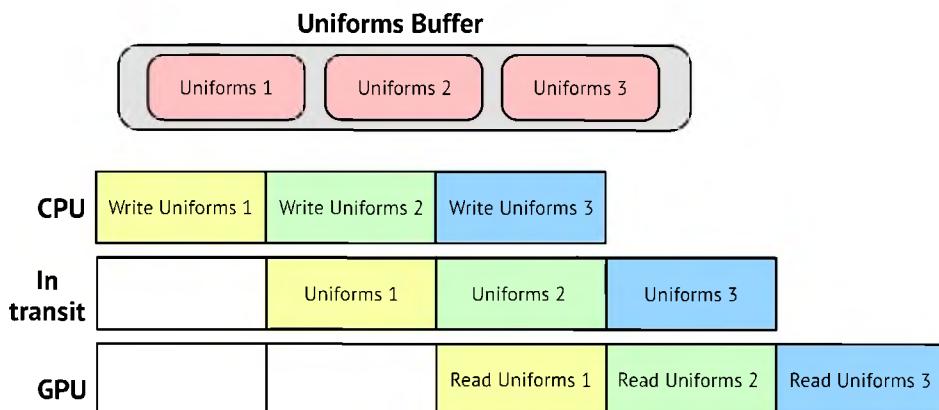
## CPU-GPU Synchronization

Managing dynamic data can be a little tricky. Take the case of **Uniforms**. You update **uniforms** usually once per frame on the CPU. That means that the GPU should wait until the CPU has finished writing the buffer before it can read the buffer.

Instead of halting the GPU's processing, you can simply have a pool of reusable buffers.

## Triple Buffering

**Triple buffering** is a well-known technique in the realm of synchronization. The idea is to use three buffers at a time. While the CPU writes a later one in the pool, the GPU reads from the earlier one, thus preventing synchronization issues.



You might ask, why three and not just two or a dozen? With only two buffers, there's a high risk that the CPU will try to write the first buffer again before the GPU finished reading it even once. With too many buffers, there's a high risk of performance issues.

► Open `Renderer.swift`, and replace `var uniforms = Uniforms()` with:

```
static let buffersInFlight = 3
var uniforms = [Uniforms](
    repeating: Uniforms(), count: buffersInFlight)
var currentUniformIndex = 0
```

Here, you replace the uniforms variable with an array of three buffers and define an index to keep track of the current buffer in use.

► In `updateUniforms(scene:)`, replace this code:

```
uniforms.projectionMatrix =
    scene.camera.projectionMatrix
uniforms.viewMatrix = scene.camera.viewMatrix
uniforms.shadowProjectionMatrix = shadowCamera.projectionMatrix
uniforms.shadowViewMatrix = shadowMatrix
```

► With:

```
uniforms[currentUniformIndex].projectionMatrix =
```

```
scene.camera.projectionMatrix  
uniforms[currentUniformIndex].viewMatrix =  
scene.camera.viewMatrix  
uniforms[currentUniformIndex].shadowProjectionMatrix =  
    shadowCamera.projectionMatrix  
uniforms[currentUniformIndex].shadowViewMatrix = shadowMatrix  
currentUniformIndex =  
    (currentUniformIndex + 1) % Self.buffersInFlight
```

Here, you adapt the update method to include the new uniforms array and make the index loop around always taking the values 0, 1 and 2.

- In `draw(scene:in:)`, before `updateUniforms(scene: scene)`, add this:

```
let uniforms = uniforms[currentUniformIndex]
```

- Build and run the app.



*Result of triple buffering*

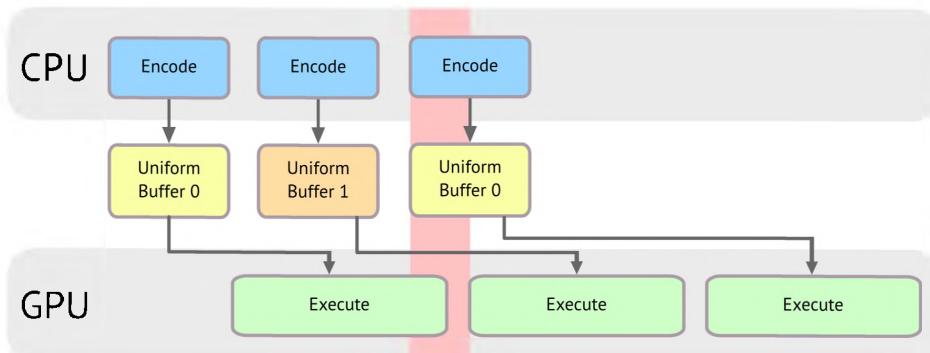
Your app shows the same scene as before.

There is, however, some bad news. The CPU can write to `uniforms` at any time and the GPU can read from it. There's no synchronization to ensure the correct uniform buffer is being read.

This is known as **resource contention** and involves conflicts, known as race conditions, over accessing shared resources by both the CPU and GPU. This can cause unexpected results, such as animation glitches.

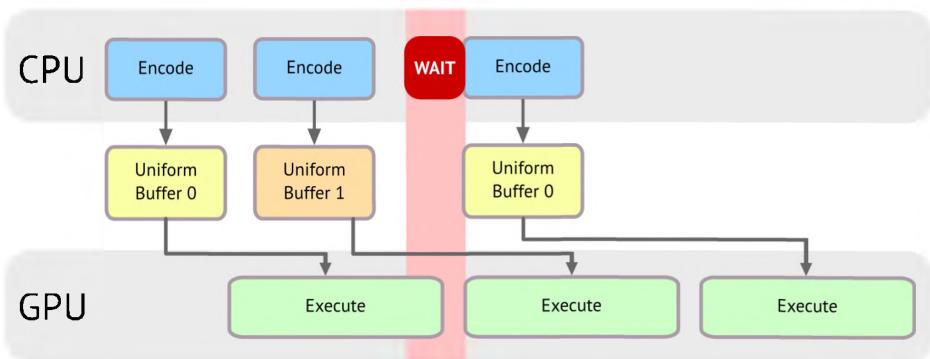
In the image below, the CPU is ready to start writing the first buffer again. However, that would require the GPU to have finished reading it, which is not the case here.

The following example shows two uniform buffers available:



#### *Resource Contention*

What you need here is a way to delay the CPU writing until the GPU has finished reading it.



A naive approach is to block the CPU until the command buffer has finished executing.

- Still in **Renderer.swift**, add this to the end of `draw(scene:in:)`:

```
commandBuffer.waitUntilCompleted()
```

- Build and run the app.

You're now sure that the CPU thread is successfully being blocked, so the CPU and GPU are not fighting over uniforms. However, the frame rate has gone way down, and the skeleton's animation is very jerky.

## Semaphores

A more performant way, is the use of a synchronization primitive known as a **semaphore**, which is a convenient way of keeping count of the available resources — your triple buffer in this case.

Here's how a semaphore works:

- Initialize it to a maximum value that represents the number of resources in your pool (3 buffers here).
- Inside the draw call the thread tells the CPU to wait until a resource is available and if one is, it takes it and decrements the semaphore value by one.
- If there are no more available resources, the current thread is blocked until the semaphore has at least one resource available.
- When a thread finishes using the resource, it'll signal the semaphore by increasing its value and by releasing the hold on the resource.

Time to put this theory into practice.

► At the top of Renderer, add this new property:

```
var semaphore: DispatchSemaphore
```

► In `init(metalView:options:)`, add this before `super.init()`:

```
semaphore = DispatchSemaphore(value: Self.buffersInFlight)
```

► Add this at the top of `draw(scene:in:)`:

```
_ = semaphore.wait(timeout: .distantFuture)
```

► At the end of `draw(scene:in:)`, but before committing the command buffer, add this:

```
commandBuffer.addCompletedHandler { _ in
    self.semaphore.signal()
}
```

► At the end of `draw(scene:in:)`, remove:

```
commandBuffer.waitUntilCompleted()
```



- Build and run the app again, making sure everything still renders fine as before.

Your frame rate should be back to what it was before. The frame now renders more accurately, without fighting over resources.

## Key Points

- GPU History, in Activity Monitor, gives an overall picture of the performance of all the GPUs attached to your computer.
- The GPU Report in Xcode shows you the frames per second that your app achieves. This should be 60 FPS for smooth running.
- Capture the GPU workload for insight into what's happening on the GPU. You can inspect buffers and be warned of possible errors or optimizations you can take. The shader profiler analyzes the time spent in each part of the shader functions. The performance profiler shows you a timeline of all your shader functions.
- GPU counters show statistics and timings for every possible GPU function you can think of.
- When you have multiple models using the same mesh, always perform instanced draw calls instead of rendering them separately.
- Textures can have a huge effect on performance. Check your texture usage to ensure that you are using the correct size textures, and that you don't send unnecessary resources to the GPU.



# Chapter 32: Best Practices

When you want to squeeze the very last ounce of performance from your app, you should always remember to follow a golden set of best practices, which are categorized into three major parts: general performance, memory bandwidth and memory footprint. This chapter will guide you through all three.



# General Performance Best Practices

The next five best practices are general and apply to the entire pipeline.

## Choose the Right Resolution

The game or app UI should be at native or close to native resolution so that the UI will always look crisp no matter the display size. Also, it is recommended (albeit not mandatory) that all resources have the same resolution. You can check the resolutions in the GPU Debugger on the **dependency graph**. Below is a partial view of the dependency graph from the multi-pass render in Chapter 14, “Deferred Rendering”:



*The dependency graph*

Notice the size of the shadow pass render target. For crisper shadows, you should have a large texture, but you should consider the performance trade-offs of each image resolution and carefully choose the scenario that best fits your app needs.

## Minimize Non-Opaque Overdraw

Ideally, you’ll want to only draw each pixel once, which means you’ll want only one fragment shader process per pixel.

If you were to draw the skybox before rendering models, your skybox texture would cover the whole render target texture. Drawing the models would then overdraw the skybox fragments with the model fragments. This is why you draw opaque meshes from front to back.

## Submit GPU Work Early

You can reduce latency and improve the responsiveness of your renderer by making sure all of the off-screen GPU work is done early and is not waiting for the on-screen part to start. You can do that by using two or more command buffers per frame:

```
create off-screen command buffer
encode work for the GPU
commit off-screen command buffer
...
get the drawable
create on-screen command buffer
encode work for the GPU
present the drawable
commit on-screen command buffer
```

Create the off-screen command buffer(s) and commit the work to the GPU as early as possible. Get the drawable as late as possible in the frame, and then have a final command buffer that only contains the on-screen work.

## Stream Resources Efficiently

All resources should be allocated at launch time — if they’re available — because that will take time and prevent render stalls later. If you need to allocate resources at runtime because the renderer streams them, you should make sure you do that from a dedicated thread.

You can see resource allocations in **Instruments**, in a **Metal System Trace**, under the **GPU > Allocation** track:

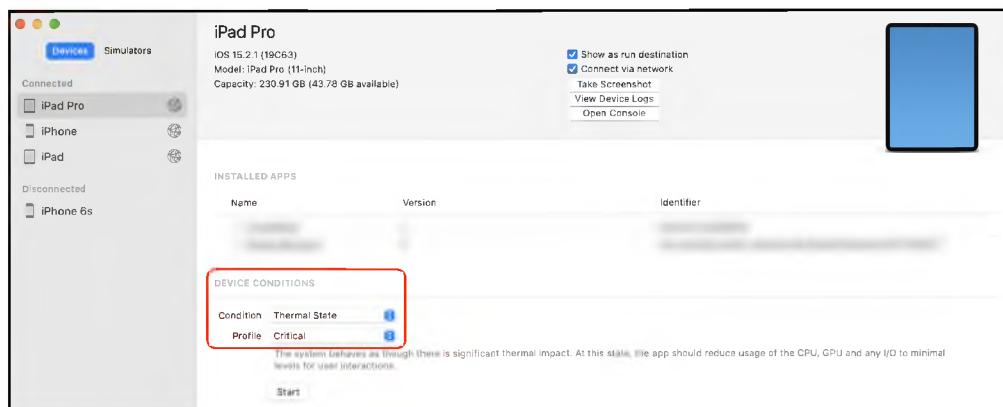


You can see here that there are a few allocations, but all at launch time. If there were allocations at runtime, you would notice them later on that track and identify potential stalls because of them.

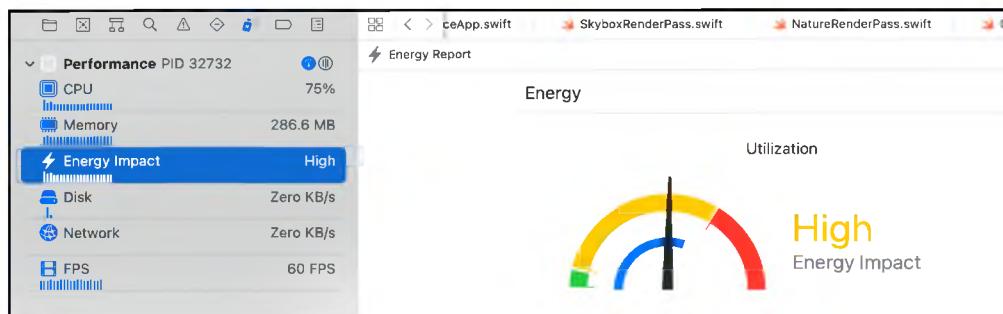
## Design for Sustained Performance

You should test your renderer under a serious thermal state. This can improve the overall thermals of the device, as well as the stability and responsiveness of your renderer.

Xcode now lets you see and change the thermal state in the Devices window from **Window ▶ Devices and Simulators**.



You can also use Xcode's **Energy Impact** gauge to verify the thermal state that the device is running at:



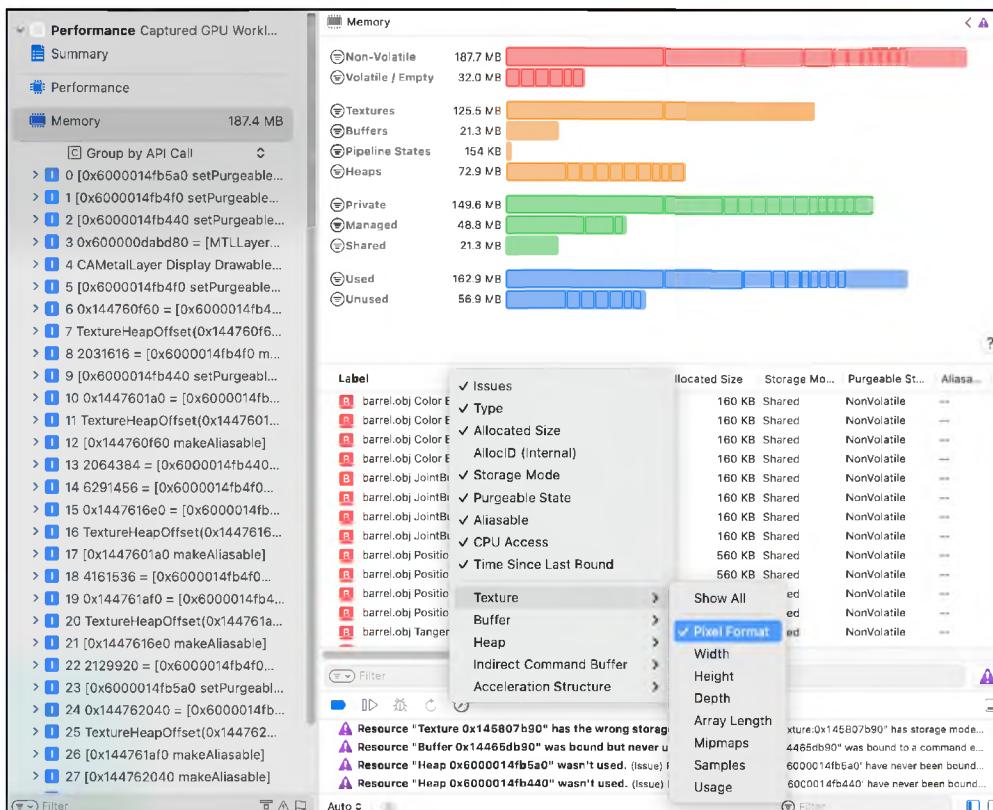
## Memory Bandwidth Best Practices

Since memory transfers for render targets and textures are costly, the next six best practices are targeted to memory bandwidth and how to use shared and tiled memory more efficiently.

## Compress Texture Assets

Compressing textures is very important because sampling large textures may be inefficient. For that reason, you should generate mipmaps for textures that can be minified. You should also compress large textures to accommodate the memory bandwidth needs. There are various compression formats available. For example, for older devices, you could use PVRTC, and for newer devices, you could use ASTC. Review Chapter 8, “Textures”, for how to create mipmaps and change texture formats in the asset catalog.

With the frame captured, you can use the Metal Memory Viewer to verify compression format, mipmap status and size. You can change which columns are displayed by right-clicking the column heading:



Some textures, such as render targets, cannot be compressed ahead of time, so you'll have to do it at runtime instead. The good news is the A12 GPU and newer supports lossless texture compression, which allows the GPU to compress textures for faster access.

## Optimize for Faster GPU Access

You should configure your textures correctly to use the appropriate storage mode depending on the use case. Use the private storage mode so only the GPU has access to the texture data, allowing optimization of the contents:

```
textureDescriptor.storageMode = .private
textureDescriptor.usage = [ .shaderRead, .renderTarget ]
let texture = device.makeTexture(descriptor: textureDescriptor)
```

You shouldn't set any unnecessary usage flags, such as `unknown`, `shaderWrite` or `pixelView`, because they may disable compression.

Shared textures that can be accessed by the CPU as well as the GPU should explicitly be optimized after any CPU update on their data:

```
textureDescriptor.storageMode = .shared
textureDescriptor.usage = .shaderRead
let texture = device.makeTexture(descriptor: textureDescriptor)
// update texture data
texture.replace(
    region: region,
    mipmapLevel: 0,
    withBytes: bytes,
    bytesPerRow: bytesPerRow)
let blitCommandEncoder = commandBuffer.makeBlitCommandEncoder()
blitCommandEncoder
    .optimizeContentsForGPUAccess(texture: texture)
blitCommandEncoder.endEncoding()
```

Again, the Metal Memory Viewer shows you the storage mode and usage flag for all textures, along with noticing which ones are compressed textures already, as in the previous image.

## Choose the Right Pixel Format

Choosing the correct pixel format is crucial. Not only will larger pixel formats use more bandwidth, but the sampling rate also depends on the pixel format. You should try to avoid using pixel formats with unnecessary channels and also try to lower precision whenever possible. You've generally been using the `RGBA8Unorm` pixel format in this book. However, when you needed greater accuracy for the G-Buffer in Chapter 14, “Deferred Rendering”, you used a 16-bit pixel format. Again, you can use the Metal Memory Viewer to see the pixel formats for textures.

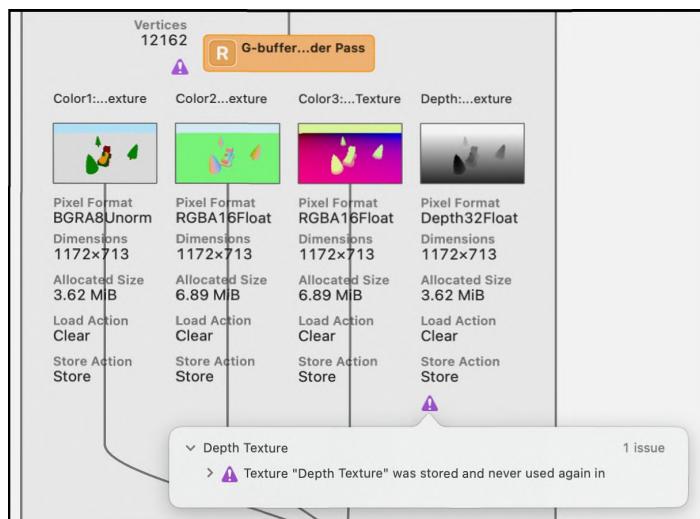


## Optimize Load and Store Actions

Load and store actions for render targets can also affect bandwidth. If you have a suboptimal configuration of your pipelines caused by unnecessary load/store actions, you might create false dependencies. An example of optimized configuration would be as follows:

```
renderPassDescriptor.colorAttachments[0].loadAction = .clear
renderPassDescriptor.colorAttachments[0].storeAction = .dontCare
```

In this case, you're configuring a color attachment to be transient, which means you do not want to load or store anything from it. You can verify the current actions set on render targets in the Dependency Viewer.



*Redundant store action*

As you can see, there is an exclamation point that suggests that you should not store the last render target.

## Optimize Multi-Sampled Textures

iOS devices have very fast multi-sampled render targets (MSAA) because they resolve from Tile Memory so it is best practice to consider MSAA over native resolution. Also, make sure not to load or store the MSAA texture and set its storage mode to `memoryless`:

```
textureDescriptor.textureType = .type2DMultisample
```

```
textureDescriptor.sampleCount = 4
textureDescriptor.storageMode = .memoryless
let msaaTexture = device.makeTexture(descriptor:
    textureDescriptor)
renderPassDesc.colorAttachments[0].texture = msaaTexture
renderPassDesc.colorAttachments[0].loadAction = .clear
renderPassDesc.colorAttachments[0].storeAction
= .multisampleResolve
```

The dependency graph will, again, help you see the current status set for load/store actions.

## Leverage Tile Memory

Metal provides access to Tile Memory for several features such as programmable blending, image blocks and tile shaders. Deferred shading requires storing the G-Buffer in a first pass and then sampling from its textures in the second lighting pass where the final color accumulates into a render target. This is very bandwidth-heavy.

iOS allows fragment shaders to access pixel data directly from Tile Memory in order to leverage programmable blending. This means that you can store the G-Buffer data on Tile Memory, and all the light accumulation shaders can access it within the same render pass. The four G-Buffer attachments are fully transient, and only the final color and depth are stored, so it's very efficient.

## Memory Footprint Best Practices

### Use Memoryless Render Targets

As mentioned previously, you should be using `memoryless` storage mode for all transient render targets that do not need a memory allocation, that is, are not loaded from or stored to memory:

```
textureDescriptor.storageMode = .memoryless
textureDescriptor.usage = [.shaderRead, .renderTarget]
// for each G-Buffer texture
textureDescriptor.pixelFormat = gBufferPixelFormats[i]
gBufferTextures[i] =
    device.makeTexture(descriptor: textureDescriptor)
renderPassDescriptor.colorAttachments[i].texture =
    gBufferTextures[i]
renderPassDescriptor.colorAttachments[i].loadAction = .clear
renderPassDescriptor.colorAttachments[i].storeAction = .dontCare
```



You'll be able to see the change immediately in the dependency graph.

## Avoid Loading Unused Assets

Loading all the assets into memory will increase the memory footprint, so you should consider the memory and performance trade-off and only load all the assets that you know will be used. The GPU frame capture Memory Viewer will show you any unused resources.

## Use Smaller Assets

You should only make the assets as large as necessary and consider the image quality and memory trade-off of your asset sizes. Make sure that both textures and meshes are compressed. You may want to only load the smaller mipmap levels of your textures or use lower level of detail meshes for distant objects.

## Simplify memory-intensive effects

Some effects may require large off-screen buffers, such as Shadow Maps and Screen Space Ambient Occlusion, so you should consider the image quality and memory trade-off of all of those effects, potentially lower the resolution of all these large off-screen buffers and even disable the memory-intensive effects altogether when you are memory constrained.

## Use Metal Resource Heaps

Rendering a frame may require a lot of intermediate memory, especially if your game becomes more complex in the post-process pipeline, so it is very important to use Metal Resource Heaps for those effects and alias as much of that memory as possible. For example, you may want to reutilize the memory for resources that have no dependencies, such as those for Depth of Field or Screen-Space Ambient Occlusion.

Another advanced concept is that of purgeable memory. Purgeable memory has three states: non-volatile (when data should not be discarded), volatile (data can be discarded even when the resource may be needed) and empty (data has been discarded). Volatile and empty allocations do not count towards the application's memory footprint because the system can either reclaim that memory at some point or has already reclaimed it in the past.

## Mark Resources as Volatile

Temporary resources may become a large part of the memory footprint and Metal will allow you to set the purgeable state of all the resources explicitly. You will want to focus on your caches that hold mostly idle memory and carefully manage their purgeable state, like in this example:

```
// for each texture in the cache
texturePool[i].setPurgeableState(.volatile)
// later on...
if (texturePool[i].setPurgeableState(.nonVolatile) == .empty) {
    // regenerate texture
}
```

## Manage the Metal PSOs

Pipeline State Objects (PSOs) encapsulate most of the Metal render state. You create them using a descriptor that contains vertex and fragment functions as well as other state descriptors. All of these will get compiled into the final Metal PSO.

Metal allows your application to load most of the rendering state upfront, improving the performance over OpenGL. However, if you have limited memory, make sure not to hold on to PSO references that you don't need anymore. Also, don't hold on to Metal function references after you have created the PSO cache because they are not needed to render; they are only needed to create new PSOs.

**Note:** Apple has written a Metal Best Practices guide (<https://developer.apple.com/library/archive/documentation/3DDrawing/Conceptual/MTLBESTPRACTICESGUIDE/index.html>) that provides great advice for optimizing your app.

## Where to Go From Here?

Getting the last ounce of performance out of your app is paramount. You've had a taste of examining CPU and GPU performance using Xcode, but to go further, you'll need to use **Instruments** with Apple's Instruments documentation (<https://help.apple.com/instruments/mac/10.0/>).

Over the years, at every WWDC since Metal was introduced, Apple has produced some excellent WWDC videos describing Metal best practices and optimization techniques. Go to <https://developer.apple.com/videos/graphics-and-games/metal/> and watch as many as you can, as often as you can.

Congratulations on completing the book! The world of Computer Graphics is vast and as complex as you want to make it. But now that you have the basics of Metal learned, even though current internet resources are few, you should be able to learn techniques described with other APIs such as OpenGL, Vulkan and DirectX. If you're keen to learn more, check out some of these great books:

Real-Time Rendering, Fourth Edition, by Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire - <http://www.realtimerendering.com>

Physically Based Rendering, Third Edition, by Matt Pharr, Wenzel Jakob and Greg Humphreys - <https://www.pbrt.org>

Computer Graphics: Principles and Practice, Third Edition, by John F. Hughes, Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, Kurt Akeley - <https://www.amazon.com/Computer-Graphics-Principles-Practice-3rd/dp/0321399528>

Fundamentals of Computer Graphics, Fourth Edition, by Steve Marschner and Peter Shirley - <https://www.amazon.com/Fundamentals-Computer-Graphics-Steve-Marschner/dp/1482229390>

Game Coding Complete, Fourth Edition, by Mike McShaffry and David Graham - <https://www.mcshaffry.com/GameCode/>



# 33

## Conclusion

Thank you again for purchasing *Metal by Tutorials*. If you have any questions or comments as you continue to develop for Metal, please stop by our forums at <https://forums.raywenderlich.com>.

Your continued support is what makes the tutorials, books, videos and other things we do at raywenderlich.com possible. We truly appreciate it.

Best of luck in all your development and game-making adventures,

– Caroline Begbie (Author), Marius Horga (Author), Adrian Strahan (Tech Editor) and Tammy Coron (FPE).

The *Metal by Tutorials* team

