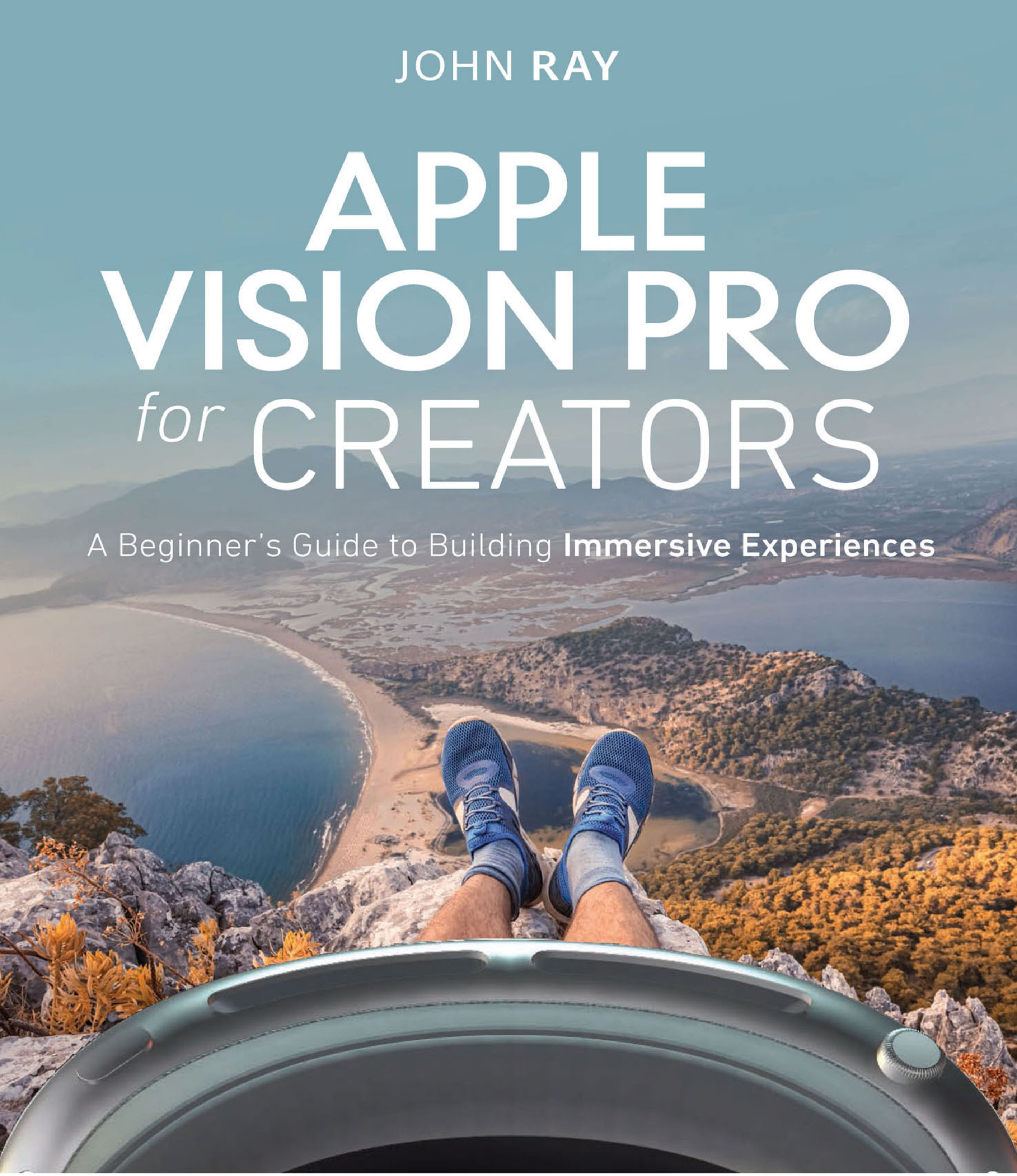

FREE SAMPLE CHAPTER

JOHN RAY

APPLE VISION PRO *for* CREATORS

A Beginner's Guide to Building **Immersive Experiences**

New Riders
VOICES THAT MATTER™

Apple Vision Pro for Creators:

A Beginner's Guide to Building Immersive Experiences

John Ray

New Riders
www.peachpit.com
Copyright © 2025 by Pearson Education, Inc. or its affiliates. All Rights Reserved.

New Riders is an imprint of Pearson Education, Inc.
To report errors, please send a note to errata@peachpit.com

Notice of Rights
This publication is protected by copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions department, please visit www.pearson.com/permissions.

Notice of Liability
The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of the book, neither the author nor Peachpit shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks
Unless otherwise indicated herein, any third-party trademarks that may appear in this work are the property of their respective owners and any references to third party trademarks, logos or other trade dress are for demonstrative or descriptive purposes only. Such references are not intended to imply any sponsorship, endorsement, authorization, or promotion of Pearson Education, Inc. products by the owners of such marks, or any relationship between the owner and Pearson Education, Inc., or its affiliates, authors, licensees or distributors.

Text Figure Credits
1.3-1.54, 2.2, 2.4-2.26, 2.27a-c, 2.28-2.32, 2.34-2.37, 3.1-3.22a-b, 3.23-3.41, 4.1-4.40, 4.43-4.51, 4.53-4.71, 5.1-5.20, 6.1-6.24, 7.1-7.21, 8.1, 8.3, 8.4, 8.6, 9.1-9.24, 9.26-9.30, 10.1-10.11, 11.1, 11.2, 11.5-11.12, 12.1-12.4, 12.6-12.62: Apple Inc

Executive Editor: Laura Norman
Editorial Services: Charlotte Kughen
Associate/Sponsoring Editor: Anshul Sharma
Senior Production Editor: Tracey Croom
Compositor: Bronkella Publishing, LLC
Proofreader: The Wordsmithery LLC
Indexer: Johnna VanHoose Dinse
Cover Design: Chuti Prasertsith
Chapter Opener Illustration: ZinetroN/Shutterstock
Cover Art: AlpakaVideo/Shutterstock and frantic00/Shutterstock
Interior Graphics: tj graham art

ISBN-13: 978-0-13-836022-1
ISBN-10: 0-13-836022-7

$PrintCode

This book is dedicated to those who seek new knowledge and try new things. You are a rare breed, and I'm honored to have you as a reader.

TABLE OF CONTENTS

ABOUT THE AUTHOR

John Ray is a lifelong Apple enthusiast and developer. He created a handwriting recognition engine at 15, published his first commercial application at 16, and continues contributing to development projects today. Over the past 25 years, John has written books on macOS, iOS, and iPadOS development, Linux, web development, networking, and computer security. He currently serves as the Senior Director of the Office of Research Information Systems at The Ohio State University. When John isn't writing, editing, or directing he is either re-creating a marine disaster in his living room or over-engineering apps and embedded systems for home automation and device integration.

ACKNOWLEDGMENTS

Many thanks to the Pearson team that made this book possible—Anshul Sharma, Charlotte Kughen, Laura Norman, Anne Groves, and everyone else behind the scenes. Writing is *much* more work than putting words on a page (that's the easy part!). Keeping me on track, making sense of my gibberish, and making sure that what I've written actually *works* is hard. Trust me, I've met myself.

INTRODUCTION

Welcome to *Apple Vision Pro for Creators*, a guide for learning how to create spatial computing experiences on the Apple Vision Pro. If you're reading this book, you probably have an idea of what Apple's headset *is*, but you might not fully appreciate how it fits in with the dozens of virtual reality headsets, augmented reality glasses, and other "tools of the future" that you can buy on the Internet. What advantage does a $3500 headset offer over a $150 pair of glasses or competitors' high-end gear like the Microsoft HoloLens?

To answer the question, we must first take a trip through the industry jargon that has sprung up as companies struggle to find some way *not* to use the words *virtual reality* or *augmented reality* with their products.

WHAT ARE VIRTUAL REALITY AND AUGMENTED REALITY?

These terms describe interactions with objects that do not exist in the real world—also known as *physical reality*. Virtual reality is typically a "replacement" for physical reality: computer generated environments where you can move and interact with items that aren't present. In virtual reality, the laws of physics (and nature itself) can be altered to present the user with otherwise impossible experiences, like flying, visiting distant planets, or just touring faraway and inaccessible places.

Augmented reality is a bit different in that it allows virtual objects and information to be mixed with physical objects. Users can interact with both physical reality and virtual reality at the same time. Augmented reality has been around in different forms for quite some time. Viewfinders on cameras that display lighting conditions, distances, and shutter speed are an example of augmented reality that we take for granted. Cars with heads-up windshield displays are another example where we can see physical reality (the road, signs, and so on) combined with virtual reality (gauges, navigation prompts, and more).

THE JARGON

When virtual reality headsets, such as the Oculus Rift DK1, initially started shipping to consumers, those of us lucky enough to obtain and develop on these devices were given a very limited set of tools. You essentially had two screens sitting on someone's face—the basic requirements for stereoscopic vision—and very little else. The way you interacted with the system varied by application, and there was rarely consistency in how you did *anything*.

It's been over a decade since these consumer headsets first appeared, and while some things have improved, many have stayed the same. There is some semblance of interface consistency on popular platforms like the Oculus Quest family, but developers are still forcing users to shift their expectations of how to work and play in three dimensions as they move between applications.

Over time, virtual reality headsets added external cameras for tracking and understanding a user's environment (some even adding quite awful "pass-thru" video to mingle the real world with virtual reality). Marketing departments were delighted to create new terms for each minor tweak introduced–hybrid reality, mixed reality, and extended reality, for example—despite no real changes from a user standpoint.

Products like Microsoft HoloLens, Google Glass, and Magic Lens *have* moved the state-of-the-art forward with augmented reality, but each have serious limitations in what they can display and how well it "mixes" with physical reality. Microsoft's HoloLens has a very limited area where virtual objects can be displayed; turn or tilt your head and they're gone. Google Glasses, on the other hand, are more like information overlays. Yes, you can see information projected into your view of the physical world, but they lack the ability to create virtual objects or immersive environments.

Recent consumer products like the XReal Air AR glasses do little more than place a flat 2D screen in front of the user. It's a relatively low-res, jittery, and poorly anchored monitor that all but obscures the physical world anyway, but hey, *it's floating in front of you!*

To call this industry and the state of AR/VR solutions "chaotic" is charitable. There are dozens of devices, each making different claims, each offering different interactivity, and each with a complete lack of consistency in experience. This confusion is frustrating for consumers and developers, and—in my opinion—it has led to a market where technology terms are thrown around with little regard to customer expectations.

THE APPLE APPROACH

Apple has entered the world of AR and VR explosively and (strangely) extremely cautiously. Rather than leaning on the various marketing terms that have been watered-down to almost no meaning, Apple is embracing the concept of *spatial computing*. Spatial computing, a term coined in the early 2000s, describes the convergence of the physical and virtual worlds. In spatial computing, there is an expectation that the headset understands the user's environment—what objects are in it, their sizes, what portions of a user's view (or other objects) are obstructed, and so on. This information is collected through myriad sensors and used to blend the physical with virtual in a way that feels natural, accounting for elements such as lighting and shadows to seamlessly meld the real and virtual.

Apple has also purposefully leaned into the *computing* portion of spatial computing, enabling the platform to run hundreds of thousands of apps at launch, while presenting a consistent user experience throughout. Other devices focus on games or niche use cases; the "vision" of the Apple Vision Pro is to create an all-in-one computer that you can use for productivity, entertainment, and gaming. It just happens to reside on your head, rather than your desktop.

What about the price? A $3500 price tag isn't far from a traditional high-end computer setup. My first personal "large" computer purchase was a G4 Cube with an Apple Cinema Display, which cost similar to *two* Apple Vision Pros. Apple isn't pinning the future of the Vision Pro on a $3500 device; they're offering it as an entry point into a new Apple platform that will expand in the coming years with cheaper, and also probably more expensive, options. This is a long-term effort, not a declaration that a single device is the pinnacle of spatial computing. This book, while specific to the Apple Vision Pro, is more about building a foundation in creating applications and experiences for the underlying operating system: visionOS. Today visionOS powers only the Vision Pro, shown in **FIGURE I.1**. Tomorrow? Who knows?

FIGURE I.1 The Apple Vision Pro

NOTE Yes, Apple itself has a tremendous marketing machine and has been known to use industry jargon and magical words in its product descriptions. The original name for visionOS was xrOS (something you may still see in Xcode and visionOS documentation.) XR is the abbreviation for "extended reality," so it's clear Apple was originally going to adopt one of the same terms as its competitors.

THE DEVICE

So, what is the Apple Vision Pro? Augmented reality headsets (such as HoloLens) use advanced transparent optics to create lenses that work like glasses, but with the ability to project virtual objects that overlap and obscure the real world. Apple has done something different.

Instead of combining the physical with the virtual through optical means, Apple has gone the route of pure virtual reality. What?! How is that possible if you can see and interact with the real world? The answer is extremely high-quality pass-thru video. You aren't seeing *through* the device; instead, you're looking at tiny screens that are mirroring the real world to your eyes. With 23 million pixels representing the world, Apple is betting that users of the Vision Pro will not even notice that they're looking at a screen.

Taking the illusion to another level, the Vision Pro has a front-facing 3D lenticular display that projects a rendering of your eyes to the outside of the device, making it appear transparent to those observing the wearer. This feature, dubbed EyeSight, provides greater engagement with the physical world and helps eliminate the isolation of wearing a headset that completely covers the eyes. The outcome is a product that *appears* to be a pair of transparent goggles, but in reality, they completely obscure the user's vision. It remains to be seen if this is the long-term approach for the visionOS platform, but as an initial product, it is a truly a unique approach to achieving the best of VR and AR worlds.

The Apple Vision Pro uses a total of 12 cameras on the inside and outside of the device for eye tracking, pass-thru video, and hand and world tracking. Speaking of which, the headset lacks dedicated controllers; the user experience relies entirely on eye-tracking, hand tracking, and gestures. This includes individual finger tracking—no giant motions or full-hand movements needed. Cameras also authenticate you to the device using eye-scanning to identify the owner of the headset.

To mix the physical and virtual, the headset includes a LiDAR sensor that measures depth by reading the time it takes for light emitted by a laser to be reflected. This can instantly create a mesh (a digital representation of physical surfaces using polygons) of the user's environment, as shown in **FIGURE I.2**. The device also uses six microphones to map how audio interacts with the objects in the environment. The result is that virtual objects can be appropriately lit in the environment, generate shadows, and be hidden by (or hide) physical objects in the room (this is known as occlusion). If the object generates audio, the sound generated considers the different surfaces in the environment to create a spatial audio experience that feels natural and mixes perfectly with the physical world.

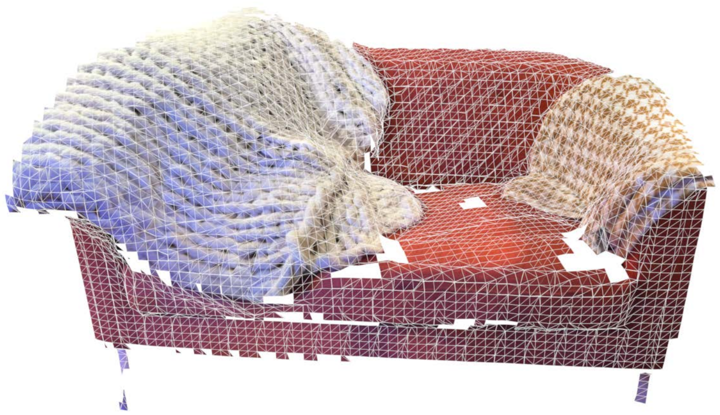

FIGURE I.2 A partial mesh of my living room loveseat (with blankets), captured by an iPhone 15 Pro LiDAR sensor

NOTE When setting up the Apple Vision Pro, you can scan your ears so that your individual ear shape is considered by the device's audio engine. Talk about thorough!

There is even more on the feature list, such as foveated rendering (using eye position to determine a user's point of interest to focus rendering power in that area), high-end Apple Silicon Processors (the R1 and M2), OLED and MicroLED technology, and on and on. Apple has packed a tremendous amount of tech into a small wearable package.

THE SOFTWARE

The raw hardware of the Apple Vision Pro is a dream for many developers, but having to deal with the dozens of sensors and cameras would be a nightmare by any measure. In typical Apple fashion, they've leveraged years of augmented reality work on the iPhone and iPad, as well as their macOS and iOS operating system experience, to create a new operating system, vision OS, that makes both using and developing for the device accessible by anyone with an interest and an ounce of motivation

Using visionOS, you gain access to all the features of the Apple Vision Pro without needing to delve into the complexities that make it work. If you want to display an object in your environment, you load the object and display it. Want to interact with the object? Attach a gesture and interact away. In Chapter 1, "Understanding the visionOS Toolkit," you'll begin by learning the Xcode development environment and a touch of the Swift programming language. By the end of Chapter 2, "From Traditional Applications to Spatial Workspaces with SwiftUI," you will have written an interactive app that displays a three-dimensional model.

The specific software and digital technologies that this book focuses on include

- **Xcode:** The platform for Apple development. Whether you're creating for a Mac or a Vision Pro, you'll be spending most of your time building your projects in Xcode.
- **Swift:** Apple's programming language for the entire Apple ecosystem. It isn't a stretch to say that once you know how to develop for the Apple Vision Pro, you can develop for *any* Apple device.
- **SwiftUI:** A Swift extension for defining user interfaces in code. Similar in some respects to HTML, SwiftUI enables you to quickly define controls, windows, and other objects for user interaction regardless of whether you're creating for iOS, iPadOS, tvOS, macOS, or visionOS.
- **Simulator:** The Simulator application lets you test your creations on your Mac without needing a headset (or an iPhone, iPad, and so on). You use the Simulator to build and test your apps and then fine-tune them on a physical device.
- **RealityKit:** This framework is the workhorse behind the capabilities of the Apple Vision Pro. It offers 3D rendering capabilities but does so with augmented reality at the forefront. It makes use of Apple's existing augmented framework (ARKit) and builds upon it with gestures and other means of interaction.

NOTE A ***framework*** is a collection of related functions that developers can use for a specific purpose. Apple platforms have frameworks for audio, web interactions, and, in the case of the Apple Vision Pro, augmented and virtual reality.

- **Windows, volumes, and spaces:** These three components make up the different scenarios you can create with visionOS. Windows are simply 2D application windows—nothing terribly special. Volumes are three-dimensional virtual objects added to the environment. Spaces, on the other hand, are entire 3D scenes that can (but don't have to) replace the physical reality entirely.
- **USD files:** Universal Scene Description files are used extensively in this book and in your projects. This file format, created by Pixar, provides a means of describing objects, materials, and even animations. Apple has standardized on these files for 3D development on its platforms. You'll most frequently encounter USDA (USD ASCII) and USDZ (USD zipped) files in the wild.
- **Reality Composer Pro:** An application for building 3D scenes in a point-and-click manner. This can be a great starting point for many projects, and you can even visualize your scenes directly on the Vision Pro without writing any code.

- **Object Capture:** An application and a collection of technologies that use photos of real-world objects to construct a virtual facsimile that you can use in your creations.
- **Materials:** A digital representation of the composition of an object, such as rubber, metal, glass, denim, fuzz, and so on.
- **Shaders:** A description, usually based on mathematical algorithms, of how light interacts with the surface of an object. Imagine an object with ridges in the surface. It would be nightmarish to create all your 3D objects with tiny ridges on their surface. A "ridge" shader might create this effect automatically so that it can be applied to any object you'd like.
- **Spatial Audio:** Audio that can be positioned in three dimensions that tracks your position and movement. Spatial audio gives the user the ability to move around different virtual audio sources and realistically changes the audio to match.
- **Scene reconstruction:** The Vision Pro enables the user to see their environment as if they were looking through glasses. Scene reconstruction takes that environment and recreates it digitally so that virtual objects can interact naturally with the real world.

This sounds like a lot, doesn't it? It is (and there's more), but it's something that you (yes, you!) can do without spending the next few years reading and watching development tutorials.

THE EXPECTATIONS

For you to get the most out of this book, we need to agree on what you can expect. First, you need the motivation to read chapters from start to finish. Concepts are introduced and reinforced through hands-on exercises. If you don't practice (even the simple stuff), you'll find it difficult to see how the different components work together. You do *not* need to be a developer, but you shouldn't be afraid of having to type a few lines of code to make a project work. Most importantly (life lesson time), this should be a topic that excites you. If it isn't, find something that does and do that instead! Watching my first projects come to life made me giddy, and I hope they do the same for you.

You don't have to do difficult math or geometry, but you should understand the difference between 2D and 3D and how coordinates are defined in three dimensions—(x, y, and z), as shown in **FIGURE I.3**. When wearing the Apple Vision Pro, the x-axis gives us positioning to the left and right and the y-axis up and down. The z-axis (the third dimension) moves objects toward and away from you. I cover these topics in more detail throughout the book, but if this makes sense to you, you're good to go! If you'd like a nice introduction to 3D concepts, Adobe has published an excellent tutorial at https://blog.adobe.com/en/publish/2020/11/09/start-3d-an-introduction-to-key-3d-concepts.

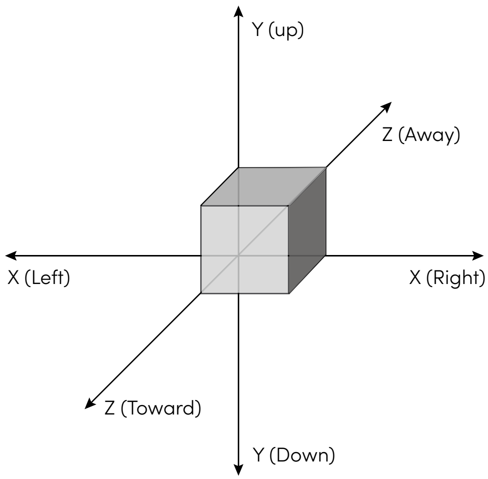

FIGURE I.3 The three axes (x, y, and z) used to define 3D positions

Regardless of your skill-level, you need at least one thing to be a successful creator: an Apple Silicon Mac. Xcode is only available on macOS, and Vision Pro development requires an M1 or later processor. It would be beneficial if you had access to a headset for testing, but this isn't required to get started. Additionally, I cover tools for the iPhone and iPad in Chapter 4, "Creating and Customizing Models and Materials," that can help with your development workflow. These devices aren't necessary to be successful, but they can help supplement the tools available on macOS.

THE PHILOSOPHY

I have spent more than three decades developing for platforms big and small, esoteric and mainstream. In recent years, I've noticed a trend of development being turned into a mundane engineering exercise. Web development, which was once something that many people enjoyed as a hobby, has become so convoluted that even small websites take months to design, debug, secure, and make accessible. Experimentation and exploration are gone—replaced with strict rules and rigidity.

Development, like art, can be a platform for self-expression and creativity. There will always be business to conduct and boring code to write, but shouldn't there be time to just *play*? I think so.

I've been thinking recently about a discussion where a peer mused "imagine what amazing creations we'd have if developers weren't obsessed with creating the perfect unimpeachable code." This cuts the crux of the problem. We've been trained that perfect code is more important than anything else, even if it affects the user experience, makes development tedious, and maintenance problematic.

My philosophy is to make things that work but to give developers the leeway to "color outside the lines". I encourage you to progress through this book looking at the techniques being presented and thinking about how you might use them for your projects. Take the examples and change them, substitute your own files and controls in place of what I present. If you think something can be done differently or better, do it!

You're in possession of the tools to bring new worlds to life. If that doesn't sound like an opportunity for fun, I don't know what does!

Let's play!

NOTE Project files and corrections for this book are available at https://visionproforcreators.com/ and www.peachpit.com/visionpro. I prompt you to download each chapter's files before you get started. Be aware that visionOS is in active development and Apple is tweaking their tools constantly, so some figures and files may have changed before you read this.

If you have questions, you can get in touch with me through the visionproforcreators.com site or via Mastodon at @johnemeryray@wisdomhole.com.

CHAPTER 8

Reconstructing Reality

When I started this book, I had a plan for where I wanted it to go and what I wanted to cover. There have been some issues that have cropped up (like a Simulator that isn't *quite* capable of fully simulating the Vision Pro), and even some code that just doesn't quite match with the developer documentation. Nonetheless, I have persevered, and you are now in the home stretch! I'm pleased to say that with the technologies covered in this chapter, you'll have a leg up on many of the other visionOS developers I've chatted with.

You're going to be using the data provider pattern established in Chapter 7, "Anchors and Planes," with additional data providers to bring more of the real world into your applications. In the Plane Detection hands-on, you may have noticed that the planes weren't quite as precise as you might hope, and objects placed in your scenes are still visible even if you walk into a different room. This chapter is going to solve those problems using the computing horsepower of the Apple Vision Pro.

This chapter focuses on three useful topics:

- **Hand-tracking:** In Chapter 7, you used a hand `AnchorEntity` to attach objects to your left and right hands. Using the full ARKit hand-tracking provider, however, you can (and will) monitor each finger joint.
- **Scene reconstruction:** See the world around you? When wearing your Vision Pro, you can literally see whatever is in your environment thanks to the high-resolution displays. However, that world is just an image. Yes, you can use a plane detector to find walls and tabletops, but with scene reconstruction, you can represent all the nooks and crannies as well.
- **Occlusion:** Occlusion means to hide or block, and it's something you experience in reality all the time. Walls hide the outdoors, closets hide your clothes, and basements hide unspeakable terrors. With the tools you've used up to this point, *nothing* hides your virtual objects (except other virtual objects). Using occlusion magic, you can make objects in the real world cover virtual objects to deliver much more immersive experiences.

Once again, what you're working on is going to require a real Apple Vision Pro. The simulator just can't provide the sensor access needed.

NOTE Be sure to head to https://visionproforcreators.com/files/ or www.peachpit.com/visionpro and download the Chapter 8 project files.

HAND-TRACKING

Most VR and pseudo-AR headsets require the use of handheld controllers that present themselves as "hands" within your view. This is generally fine for gaming, but it doesn't take long before your brain registers the disconnect between what you're seeing on the screen versus what your hands are really doing. The Apple Vision Pro is designed to use your hands as its controllers, and it does so with almost alarming accuracy.

The hand-tracking you used in the last chapter is fun and can certainly create some interesting effects, but it has very little flexibility in terms of interactions. Wouldn't you like to interact directly with objects with more than just a fingertip and a thumb? A hand-targeted `AnchorEntity` is easy to use, but by employing ARKit with a `HandTrackingProvider` (https://developer.apple.com/documentation/arkit/handtrackingprovider), you can track up to 27 different joints per hand.

Hand-tracking works in the same way as the `PlaneDataDetector`:

1. You create an ARKit session with `ARKitSession()`.
2. A data provider is created. For hand-tracking this is done with `HandTrackingProvider()`.
3. The ARKit session is `run` with the tracking provider.
4. Updates arrive containing a `HandAnchor`.
5. You process the updates however you want!

Hands are different than planes and so is the data that hand anchors provide. Let's take a look at ARKit's `HandAnchor` and what information it contains.

ARKit's HandAnchor

An ARKit hand anchor tracks a hand's position in 3D space and provides three useful properties you'll access in your upcoming code:

- **`.originFromAnchorTransform`:** The location and orientation of the base of the hand in world space.
- **`.chirality`:** The "handedness" of the update. In other words, the `.right` or `.left` hand.
- **`.hand`:** Access to the individual joints in the hand, along with the location of each joint in relation to the base of the hand.

Of these, I'd like to believe that your interest gravitates toward `handSkeleton`—because who doesn't like a skeleton? Read more about HandAnchors at https://developer.apple.com/documentation/arkit/handanchor.

Hand Skeletons and Joints

The `.handSkeleton` property is an instance of a `HandSkeleton` data structure. Within the skeleton is a collection of `joints`, with associated names and transformations.

That, unfortunately, is about all the information Apple makes *easily* available. You can get a list of all the available hand joints at https://developer.apple.com/documentation/arkit/handskeleton/jointname, but the names of the joints don't necessarily make that much sense (what is the intermediate tip of a finger?!).

For a better sense of where the different joints are located, you can turn to a developer video where Apple displays a few frames with a diagram of hand and joint locations: https://developer.apple.com/videos/play/wwdc2023/10082/?time=935.

Assuming you aren't interested in playing a video as reference material, I've provided a screen capture in **FIGURE 8.1**. This figure, however, includes the word "hand" in front of each joint, which has been removed from the actual data structure since the video was created.

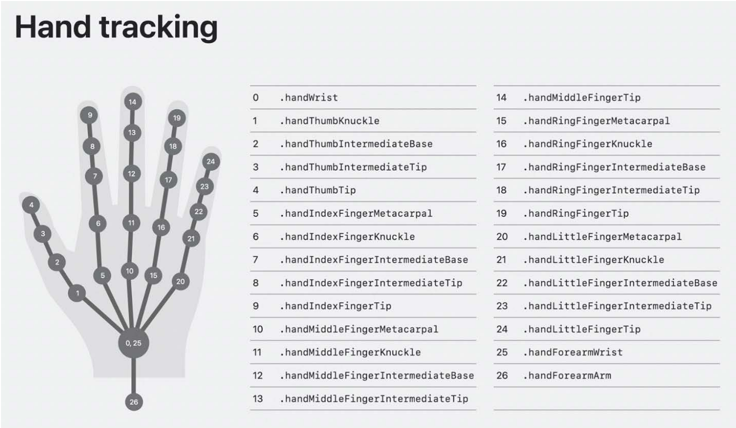

FIGURE 8.1 The joint locations on a hand— just ignore the "hand" prefix to each joint name

Accessing Individual Joint Locations

To access the current location and orientation (the transform matrix) of an individual joint within a hand anchor, you use this syntax:

```
<joint transform matrix> = <anchor>.handSkeleton?.
                     joint(<joint name>).anchorFromJointTransform
```

The transformation matrix you can get from a joint is relative to the base of the hand, so you can't use it directly. Instead, you must multiply it by the transformation matrix of the base in world space. That value is provided by `anchor.originFromAnchorTransform`:

```
<world transform matrix of joint> = <joint transform matrix> *
                                    <anchor>.anchorFromJointTransform
```

The world transform of the joint can subsequently be used to set the position of an entity. This enables you to create an entity that behaves like an `AnchorEntity` for every single joint on each hand.

Working with All Joints

When I first started coding the project in this chapter, I began by explicitly referring to individual joints and tracking just a few. After explicitly listing out about a dozen of the joints, I decided that rather than manually coding up a few joints, why not track them all?

To access a collection of all the joints in a `HandSkeleton`, you use the class property `JointName.allCases`:

```
HandSkeleton.JointName.allCases
```

From there, you can iterate over each joint with a loop like this:

```
for joint in HandSkeleton.JointName.allCases  {
    if let fingerJoint = anchor.handSkeleton?.joint(joint) {
        // Do something useful with the fingerJoint here.
    }
}
```

That's everything you need to create a tracking class. You'll be doing this as a hands-on project in a way that is slightly different from past projects. Your primary goal in this hands-on is to create a new `HandTracker.swift` class, not to build any fancy interfaces or experiences. Nonetheless, you'll want to create that class within a Mixed Immersive Space project, making it much easier to test the code.

HANDS-ON: CREATING A HAND TRACKER CLASS

One of the difficulties of being this far into the development process is that you're not going to encounter many cases where a line or two of code does something useful. Instead, you need to use established coding patterns that *all* developers use. There are three projects in this chapter, and each fits this category. Don't feel bad about not writing all the code yourself because *no one else did either!*

This project establishes a Hand Tracker class that can be used for tracking all the joints in both hands. The class publishes two variables: `rightHandParts` and `leftHandParts`. Each is a collection using the joint name as the key and an `Entity` as the value. The `Entity` is positioned according to the relevant `HandAnchor` and can be used to hold whatever you want.

To verify that it all works, in ImmersiveView.swift, you attach a `ModelEntity` to each joint in the hand skeleton, as shown in **FIGURE 8.2**. There isn't going to be much hand-holding here (unintentional pun!), because you've been through these processes several times.

FIGURE 8.2 The output: a bunch of clown noses attached to the joints of your hand

Setting Up the Project

Create a new Mixed Immersive project in Xcode named **Hand Skeleton**. Once open in Xcode, complete the usual steps to get the project ready for coding:

1. [Optional] Update the ContentView.swift file to include an introduction and the <App Name>App.swift file to size the content appropriately.
2. Remove the extra sample code from the ImmersiveView.swift file. Make sure the `RealityView` is empty.
3. This project (obviously) uses hand-tracking capabilities, the project's `Info.plist` file ("Info" within the Project Navigator) to include the key `NSHandsTrackingUsageDescription`, and a string prompt to ask for permission.

> **NOTE** If any of this sounds unfamiliar, please revisit Chapters 6 and 7 to learn more about Immersive Spaces, Data Providers, and the accompanying project setup.

Adding the HandTracker Class

Select the Hand Skeleton folder in the Xcode Project Navigator. Choose File, New, File from the Xcode menu. When prompted for the template to use, select visionOS, Swift File, and click Next. Name the new file **HandTracker** and save it to the folder with your project's other swift files. Also, be sure that the Group and Target settings remain on their default values.

Rather than adding bits and pieces of code to the class file, it makes the most sense to enter the entire contents of the file and then review it. As you already know, this is going to be very similar to the Chapter 7 `PlaneDetector` class. Replace the contents of the HandTracker.swift file with the code in **LISTING 8.1**.

If you don't feel like typing this yourself, use the HandTracker.swift file included with the Chapter 8 project archive. It's much shorter than it looks. The wrapping of the book text makes it appear more unwieldy than it is.

LISTING 8.1 Tracking Each Joint in Each Hand

```
import ARKit
import RealityKit

@MainActor class HandTracker: ObservableObject {

    private let session = ARKitSession()
    private let handData = HandTrackingProvider()
    @Published var leftHandParts: [HandSkeleton.JointName:Entity] = [:]
    @Published var rightHandParts: [HandSkeleton.JointName:Entity] = [:]
```

```
    func startHandTracking() async {
        print("Starting Tracking")

        for joint in HandSkeleton.JointName.allCases {
            rightHandParts[joint] = Entity()
            leftHandParts[joint] = Entity()
        }

        try! await session.run([handData])
        if HandTrackingProvider.isSupported {
            for await update in handData.anchorUpdates {
                switch update.event {
                case .added, .updated:
                    updateHand(update.anchor)
                case .removed:
                    continue
                }
            }
        }
    }

    func updateHand(_ anchor: HandAnchor) {
        for joint in HandSkeleton.JointName.allCases  {
            if let fingerJointTransform = anchor.handSkeleton?
                        .joint(joint).anchorFromJointTransform {

                let worldspaceFingerTransform =
                 anchor.originFromAnchorTransform * fingerJointTransform

                if anchor.chirality == .right { rightHandParts[joint]!.
                           setTransformMatrix(worldspaceFingerTransform,
                           relativeTo: nil)
                } else {
                        leftHandParts[joint]!.
                        setTransformMatrix(worldspaceFingerTransform,
                        relativeTo: nil)
                }
            }
        }
    }
}
```

The class file starts by importing ARKit and RealityKit, the two frameworks needed for this code to work.

An ARKit session is defined (`session`), as well as an instance of the `HandTrackingProvider` (`handData`). Next, the `leftHandParts` and `rightHandParts` collections are defined. Each consists of key/value pairs where the key is a joint name (`HandSkeleton.JointName`) and the value is an `Entity`. These include the `@Published` wrapper because they'll be accessed directly in your application views.

The `startHandTracking` function begins by looping over the full list of joint names:

```
for joint in HandSkeleton.JointName.allCases {
    rightHandParts[joint] = Entity()
    leftHandParts[joint] = Entity()
}
```

With Plane Detector project, you added planes to an `Entity` as visionOS detected them. It would be impossible to "use" a plane before it was detected. With the joints in a hand, however, you already know all the possible joints. You code could be much simpler if you can access *any* joint at *any* time, regardless of whether it's currently detected by the sensors. To that end, you use this loop to initialize each joint in the `rightHandParts` and `leftHandParts` collections to an empty `Entity`. Now you can access the joints in other code without issue, even if they happen to be momentarily hidden.

NOTE My experience with the `HandTrackingProvider` has been that it sometimes temporarily loses joints if you move your hands to extreme locations outside the range of the cameras, but they are very quickly reestablished as soon as the Vision Pro can see your hands again.

Finally, the ARKit `session` is started with the `handData` data provider. If the application has been granted hand-tracking permission (`HandTrackingProvider.isSupported`), a loop begins that waits for hand anchor updates (`handAnchor.anchorUpdates`). When an `update` with the event type `added` or `updated` is received, the `switch` statement calls `handUpdate`. If the `update` is of the type `removed`, nothing happens. The joint is left as-is until it is redetected.

The `updateHand` function accepts an incoming `HandAnchor` in the `anchor` variable. It loops through all the names of the joints in a hand skeleton (`HandSkeleton.JointName.allCases`), setting a `joint` variable to each name as the loop runs. Each joint's location (`anchor.handSkeleton?.joint(joint).anchorFromJointTransform`) is multiplied by the hand anchor's transform matrix in world space (`anchor.originFromAnchorTransform`), giving us a final transform matrix `worldspaceFingerTransform` that can be used to position an entity.

As the final step, the `chirality` is tested and is used to set either the `leftHandParts` or `rightHandParts` collection's entity `transform` matrix to the `worldspaceFingerTransform`.

The finished HandTracker class is capable of tracking every single joint available through visionOS and can be used much like an `AnchorEntity`. You'll do that now.

Adding Model Entities

Open the ImmersiveView.swift file in Xcode. Add an `import` statement for `ARKit` after the existing imports. This is required to access all the `HandSkeleton` joint names:

```
import ARKit
```

At the start `ImmersiveView` struct, add a new `@Observed` variable for the `HandTracker` class:

```
@ObservedObject var handTracker = HandTracker()
```

Within the `RealityView` block, create a new material (I'm using an unlit red material) and an object to anchor on your fingers. My code looks like this:

```
let material = UnlitMaterial(color: .red)

let fingerObject = ModelEntity(
    mesh: .generateSphere(radius: 0.01),
    materials: [material]
)
```

Now, add another loop through all the recognized joints. This time, add a copy of the `fingerObject` `ModelEntity` to each joint entity.

```
for joint in HandSkeleton.JointName.allCases {
    handTracker.rightHandParts[joint]!.addChild(
            fingerObject.clone(recursive: true))
    handTracker.leftHandParts[joint]!.addChild(
            fingerObject.clone(recursive: true))
    content.add(handTracker.rightHandParts[joint]!)
    content.add(handTracker.leftHandParts[joint]!)
}
```

> **TIP** You can only add one instance of a given `ModelEntity` to your content. To use it again, you have to make a copy. You can do this with the `clone` function. Typing `<model entity>.clone(recursive: true)` creates a brand-new copy of the model entity that can be used elsewhere.

Now the code in ImmersiveView.swift needs to *start* the `handTracker`. Add a task immediately following the `RealityView` code block:

```
.task() {
    await handTracker.startHandTracking()
}
```

You may now start the application, enter the immersive scene, and take a look at your sphere-covered hands!

SCENE RECONSTRUCTION

Hand-tracking can enable experiences where interactions with the environment seem very natural. However, the problem is that the environment itself still doesn't seem very natural. Plane detection provides the ability to place virtual objects on real-world surfaces like seats and tables, but it doesn't consider things like pillows on couches and the fact that literally no living human has ever kept a table surface completely clean for more than 47 seconds. As a result, virtual objects added to the planes could exist inside real-world objects that happened to be in the same location on the plane. Let's face it, plane detection is cool, but it just doesn't give us a very "exact" representation of all the different surfaces that virtual objects may encounter.

Scene reconstruction takes plane detection to another level. Think of scene reconstruction as plane detection on steroids. Rather than just looking for flat surfaces, a `SceneReconstructionProvider` (https://developer.apple.com/documentation/arkit/scenereconstructionprovider) considers *all* the incoming data from the Vision Pro to recreate the geometry of all the surroundings where the user is located. It's like taking a giant sheet and covering everything with it, tucking the sheet into all the spaces around all the different objects.

This data is provided by multiple `MeshAnchors`, each with a mesh (shape) that's constantly tracked in the environment. By adding these meshes to your content, you effectively "reconstruct" the real world within a virtual space.

With the right meshes in place, you can have objects interact with the miscellaneous "stuff" you place around yourself. Objects can roll off pillows and under tables and even fall in places that make them difficult to retrieve—making virtual life just as annoying as the real thing.

ARKit MeshAnchors

Yes, a `MeshAnchor` works in a very similar way to the hand anchors and plane anchors, so you're gonna be experiencing more déjà vu. Let's quickly cover the properties you might need when you process mesh anchor updates:

- **`.originFromAnchorTransform`:** The location and orientation of the detected shape in world space.
- **`.geometry`:** A collection of the different shapes that make up a mesh anchor.
- **`.geometry.classifications`:** A classification of each face of the geometry that makes up a mesh. Because a mesh may span multiple objects, one must look at all the different geometry classifications to see everything that has been detected. Review https://developer.apple.com/documentation/arkit/meshanchor/meshclassification if you're interested in what objects can be reported by a `MeshAnchor`.

You can learn more about MeshAnchors at https://developer.apple.com/documentation/arkit/meshanchor, but probably the most important thing to understand is that it takes work to turn a MeshAnchor into something useful. With planes, for example, you need to create a plane ModelEntity and add it to your content. MeshAnchors come to use with geometry information but not in a form you can use.

Generating Collision Shapes

To use a MeshAnchor, you need to turn it into something that can be used in your Reality View content. To do this, you take advantage of a ShapeResource class method that turns a MeshAnchor into a shape that can be used as a collision component.

You might be wondering, “What good does that do? Are you saying it doesn’t give a shape I can use to style and present a virtual object?” Yes, that’s exactly what I’m saying. You can create an entity and assign a collision component based on the anchor, and then add it to the content. This will have the effect of creating an invisible object that matches the shape and placement of real-world objects, but it only serves the purpose of allowing objects to collide with it realistically.

To generate a collision shape from a mesh anchor, you first generate the shape:

```
let <shape mesh> = try! await ShapeResource.generateStaticMesh(
                                        from: <mesh anchor>)
```

Then, you can create a new ModelEntity, set its collision component to the generated mesh, and add a physicsBody for good measure:

```
let <model entity> = ModelEntity()
<model entity>.collision = CollisionComponent(shapes: [<shape mesh>],
                                              isStatic: true)
meshEntity.physicsBody = PhysicsBodyComponent(mode: .static)
```

Like all the other data providers, this process must be repeated over and over as the headset detects or stops detecting new surfaces, so you need another new class for the implementation (which you make momentarily). But, before you do that, there’s “one more thing” I need to discuss because it will truly bring your projects to life.

Occlusion

Apple has built a heck of a device, but the Apple Vision Pro’s development tools are still in their early stages. Some tasks that have worked great on the iOS/iPadOS platforms can be painful on the Vision Pro. One of these is **occlusion**, or the process of hiding one object behind another. Your hands, for example, occlude virtual objects, which is necessary for interactions. Virtual objects hide other virtual objects that are behind them. What’s missing is for real-world objects to occlude virtual objects.

You may have noticed over the past several exercises that if you place a virtual object somewhere in the environment then walk behind a wall or put a physical object in front of it, you can *still* see the virtual object. It's like having virtual X-ray vision but can also be quite jarring and bring you out of an experience really quickly.

Occlusion Material

Apple provides a special material, called an **occlusion material**, that can be applied to virtual objects. The object becomes invisible to the viewer but still blocks virtual objects behind it:

```
let material = OcclusionMaterial()
```

You *should* be able to take this occlusion material, apply it to the model entities you create during scene reconstruction, and gain the effect of real objects blocking the virtual.

But it's not going to work. The collision shape you add to a model entity isn't a visible surface. You can't apply a material or see a model that only has a collision shape. I suspect Apple will remedy this in the future, but for now, occlusion is not simple.

Or is it?

Occlusion Meshes

As it turns out, the occlusion mesh problem has been solved reasonably well by a GitHub user named XRealityZone. Within their GitHub repository, they maintain a visionOS project called what-vision-os-can-do. This has some useful code snippets that you can use in your creations and is a combination of community contributions and code that Apple has published in its examples.

You can access the repository here:

https://github.com/XRealityZone/what-vision-os-can-do/tree/main

Within the project is a method that translates a `MeshAnchor` into a `MeshResource`, which is exactly what you need to do. You make use of a modified version of this code when you build a scene reconstruction class next. You create entity models with collision shapes and model meshes that can use any material or shader you want—including the occlusion material.

I'm sure Apple will eventually make the process easier, but if you use the `SceneReconstructor` class you're about to code, you'll have that functionality *now*.

HANDS-ON: CREATING A SCENE RECONSTRUCTOR CLASS

Here you are, once again, about to build a class that uses an ARKit session to collect data. This is *yet again* the same code pattern used for plane and hand-tracking. It's also the last time you're going to have to hear me say that. Once you've finished the class, you're going to jump into a third exercise that puts it and the hand-tracking class to good use.

In this project, you create another new class, `SceneReconstructor`, that employs a `SceneReconstructionProvider` to generate `MeshAnchors`. Each `MeshAnchor` is used to position a `ModelEntity` that is built using the geometry in the anchor. It has both collision shapes and a surface with applied material. You track all of them in an `EntityMap` collection.

In ImmersiveView.swift, you add these model entities to the `RealityView`. Users will see a version of their surroundings covered in any material you choose, as shown in **FIGURE 8.3**.

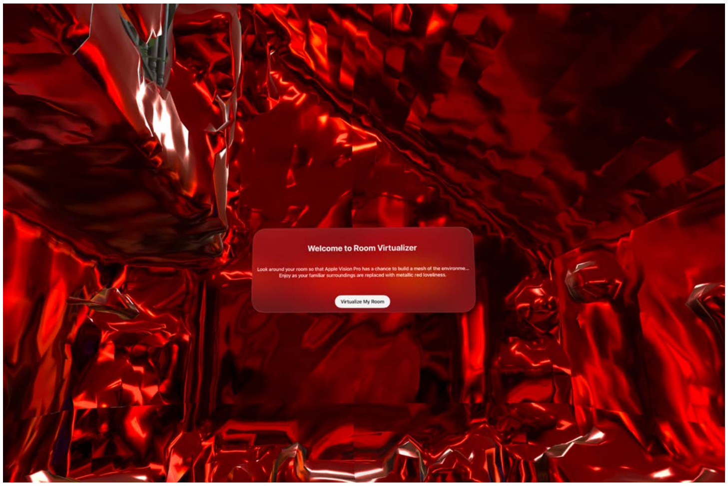

FIGURE 8.3 You are now living in the Matrix.

Setting Up the Project

Create a new Mixed Immersive project in Xcode named **Room Virtualizer** and then follow these steps:

1. [Optional] Update the ContentView.swift file to include an introduction and the <App Name>App.swift `file to size the content appropriately.`
2. Remove the extra code from the ImmersiveView.swift file. Edit the `RealityView` so that it is empty.
3. The project uses world-sensing capabilities; the project's Info.plist file (Info within the Project Navigator) needs to be updated with the key `NSWorldSensingUsageDescription`, along with a string prompt to ask for permission.

Adding the SceneReconstructor Class

Add a new Swift file named **`SceneReconstructor`** to your project. Save the file to the same location as the other Room Virtualizer Swift files. Leave the other settings at their defaults.

Open the SceneReconstructor.swift file in the Xcode editor then enter the code in **LISTING 8.2**.

LISTING 8.2 Tracking Shapes Detected by the Vision Pro

```
import ARKit
import RealityKit
import Foundation

@MainActor class SceneReconstructor: ObservableObject {

    private let session = ARKitSession()
    private let sceneData = SceneReconstructionProvider()
    private var entityMap: [UUID: Entity] = [:]
    @Published var parentEntity = Entity()

    func startReconstruction() async {
        try! await session.run([sceneData])
        if SceneReconstructionProvider.isSupported {
            for await update in sceneData.anchorUpdates {
                switch update.event {
                case .added, .updated:
                    let shape = try! await
                           ShapeResource.generateStaticMesh(from:
                           update.anchor)
                    updateMesh(update.anchor, shape: shape)
                case .removed:
                    removeMesh(update.anchor)
                }
            }
        }
    }

    func updateMesh(_ anchor: MeshAnchor, shape: ShapeResource) {
        if entityMap[anchor.id] == nil {
            let entity = Entity()
            let meshEntity = ModelEntity(mesh:
                                 anchorToMeshResource(anchor))
            let material = SimpleMaterial(color: .red, isMetallic: true)
            meshEntity.collision = CollisionComponent(shapes:
                                       [shape], isStatic: true)
            meshEntity.components.set(InputTargetComponent())
            meshEntity.model?.materials = [material]
            meshEntity.physicsBody =
                              PhysicsBodyComponent(mode: .static)
            entity.addChild(meshEntity)
            entityMap[anchor.id] = entity
```

```
            parentEntity.addChild(entity)
        } else {
            let entity = entityMap[anchor.id]!
            let meshEntity = entity.children[0] as! ModelEntity
            meshEntity.collision?.shapes = [shape]
            meshEntity.model?.mesh = anchorToMeshResource(anchor)
        }
        entityMap[anchor.id]?.transform = Transform(matrix:
                              anchor.originFromAnchorTransform)
    }

    func removeMesh(_ anchor: MeshAnchor) {
        entityMap[anchor.id]?.removeFromParent()
        entityMap.removeValue(forKey: anchor.id)
    }

    func anchorToMeshResource(_ anchor: MeshAnchor) -> MeshResource {
        var desc = MeshDescriptor()
        let posValues = anchor.geometry.vertices.asSIMD3(ofType:
                                                 Float.self)
        desc.positions = .init(posValues)
        let normalValues = anchor.geometry.normals.asSIMD3(ofType:
                                                   Float.self)
        desc.normals = .init(normalValues)
        do {
            desc.primitives = .polygons(
                (0..<anchor.geometry.faces.count).map { _ in UInt8(3) },
                (0..<anchor.geometry.faces.count * 3).map {
                    anchor.geometry.faces.buffer.contents()
                        .advanced(by: $0 *
                                  anchor.geometry.faces.bytesPerIndex)
                        .assumingMemoryBound(to: UInt32.self).pointee
                }
            )
        }
        let meshResource = try! MeshResource.generate(from: [desc])
        return(meshResource)
    }
}

extension GeometrySource {
    func asArray<T>(ofType: T.Type) -> [T] {
        assert(MemoryLayout<T>.stride == stride,
         "Invalid stride \(MemoryLayout<T>.stride); expected \(stride)")
        return (0..<self.count).map {
            buffer.contents().advanced(by: offset + stride *
```

```
                Int($0)).assumingMemoryBound(to: T.self).pointee
        }
    }

    func asSIMD3<T>(ofType: T.Type) -> [SIMD3<T>] {
        return asArray(ofType: (T, T, T).self).map
                        { .init($0.0, $0.1, $0.2) }
    }
}
```

TIP No worries if you're not up to typing all of that into Xcode. You can use the SceneReconstructor.swift file included in the Chapter 8 Room Virtualizer project instead.

The logic should be obvious by now: An ARKit `session` is created along with an instance of `SceneReconstructionProvider` (`sceneData`). Supporting data structures `parentEntity` and `entityMap` hold all the mesh model entities and a mapping between anchor IDs and model entities, respectively.

The `startReconstruction` function first verifies you have permission to monitor the environment (`SceneReconstructionProvider.isSupported`). Assuming there are no issues, it waits for an incoming `MeshAnchor` and calls `updateMesh` or `removeMesh` depending on whether an anchor has been updated/added or removed. For new and updated meshes, a `shape` is created; this is the collision shape you can *easily* generate from the anchor.

When `updateMesh` is called, the `shape` *and* the `anchor` are provided as arguments. The function checks `entityMap` to see if the anchor has been seen before. If it hasn't, a new `entity` is created—our version of an `AnchorEntity`. A `ModelEntity` named `meshEntity` is defined with the generated collision shapes, a metallic red color, a physics body, and an input target component.

NOTE The `meshEntity` is initialized with a mesh created from the function `anchorToMeshResource(<anchor>)`. This is the utility function that Apple should define for you but doesn't. It takes the `MeshAnchor` and builds a `MeshResource` that is used to give `meshEntity` a visible model.

The `meshEntity` is then added to `entity`, which, in turn, is added to the published `parentEntity`.

If an anchor *has* been seen before and needs an update, the code fetches the `entity` from `entityMap`, grabs the `meshEntity` from that, and changes its collision shapes to the updated `shape` as well as updating the visible model `mesh` with `anchorToMeshResource`.

When a `MeshAnchor` is no longer being tracked, the `removeMesh` function removes the `entity` (and the `ModelEntity` it contains) as well as any `entityMap` references to it.

The remainder of the code (`anchorToMeshResource`, `asArray`, and `asSIMD3` functions) is provided as-is with minor modifications from the community code at https://github.com/XRealityZone/what-vision-os-can-do/blob/ed7adb8c281d68aaf2cdc472986127fc11f44cca/WhatVisionOSCanDo/ShowCase/WorldScening/WorldSceningTrackingModel.swift#L70.

EXTENSIONS

Notice that the `asArray` and `asSIMD3` functions are in a block labeled with `extension`. An extension enables a developer to add new functionality to an existing class or struct—in this case, an ARKit structure named `GeometrySource` (https://developer.apple.com/documentation/arkit/geometrysource).

These two data conversion functions aren't part of `GeometrySource` by default. By adding them as an extension, they behave as if they were features originally provided by Apple.

Visualizing the Results

To view the results of all this work, you need to make some modifications to ImmersiveView.swift. Add a `sceneReconstructor` variable initialized to the new `SceneReconstructor` class at the top of the `ImmersiveView struct`:

```
@ObservedObject var sceneReconstructor = SceneReconstructor()
```

Next, add the `parentEntity` to the content within `RealityView`:

```
RealityView { content in
    content.add(sceneReconstructor.parentEntity)
}
```

Finish up by adding a task that starts scene reconstruction immediately after the `RealityView` block. Apple indicates that any scene reconstruction tasks should be started with *low* priority, which you can indicate with the `priority` argument:

```
task(priority: .low) {
    await sceneReconstructor.startReconstruction()
}
```

You can now run the application on your Apple Vision Pro and watch as your familiar surroundings are turned into a metallic red nightmare.

Congratulations! You've built hand-tracking and scene reconstructions classes that can be used in future applications. Let's wrap up by building an application that uses these classes to build a fully interactive physics playground that blends virtual and reality seamlessly.

HANDS-ON: RECONSTRUCTION

One of the nice things about creating reusable code is that once it is built, it can just be used without thinking about it again. By reusing the `HandTracker` and `SceneReconstructor` classes in this project, you can focus solely on the functionality you want to provide without getting into the nitty-gritty of data providers and ARKit sessions and all that fun. You just get to build and play.

This exercise is designed to be a playground for you, the developer. You can try different indirect and direct object interactions, mess with gravity, and just practice with all the capabilities you've been learning throughout the book.

In this project, Reconstruction, you use tap gestures to drop random objects from your fingertips. Then you can (carefully) use your hands to scoop up the objects and move them, flick them around, or use an indirect gesture to pick them up and position them throughout the environment. Using scene reconstruction, the application considers the shapes in your space in its physics simulation and virtual objects react to physical objects as you'd expect.

The finished project will likely result in a significant mess around your room, as shown in **FIGURE 8.4**. Thankfully, cleaning up is just a matter of closing the application.

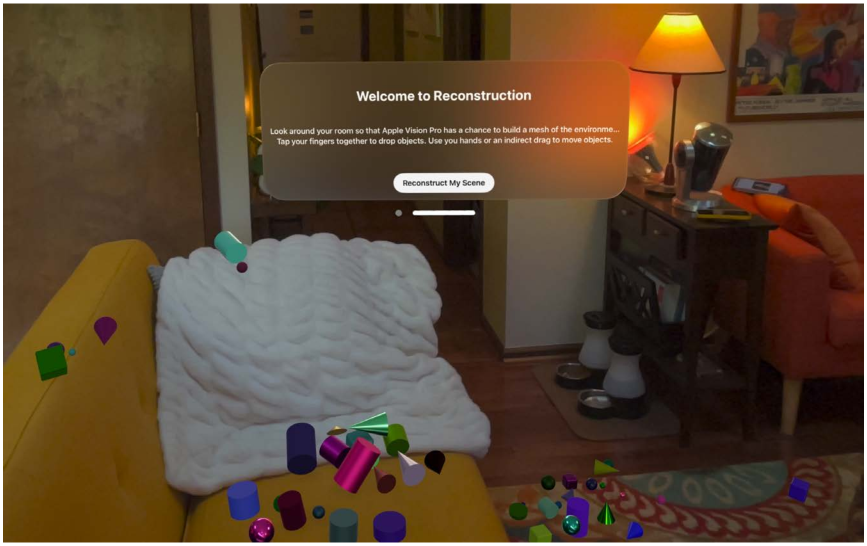

FIGURE 8.4 Place and interact with random objects scattered around your room.

Setting Up the Project

Create a new Mixed Immersive project named **Reconstruction**.

If desired, update the ContentView.swift file to include an introduction and the <App Name>App.swift file to size the content appropriately.

Remove the extra code from the ImmersiveView.swift file. Be sure the `RealityView` code block is empty.

This project needs both world-sensing and hand-tracking; update the project's `Info.plist` file to include keys and string values for `NSWorldSensingUsageDescription` and `NSHandsTrackingUsageDescription`.

Adding the HandTracker and SceneReconstructor Classes

Now add the HandTracker.swift and SceneReconstructor.swift files you created in the previous two projects to the Reconstruction project. The easiest way to do this is to choose File, Add Files to "Reconstruction" from the Xcode menu. When prompted, as shown in **FIGURE 8.5**, drill down into the Hand Skeleton project and select the HandTracker.swift file.

Leave the other settings with their defaults (Copy Items, Create Groups, and Add to Targets are all selected) and then click Add. The file now appears in your Xcode Project navigator. Repeat these steps for SceneReconstructor.swift file found within the Room Virtualizer project.

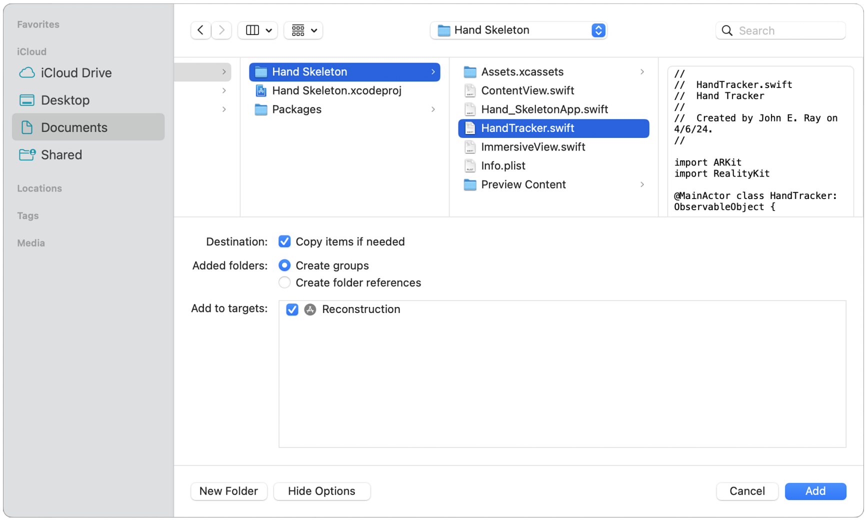

FIGURE 8.5 Importing the class files to the Reconstruction project

Before going any further (and before I forget), open the newly added SceneReconstructor. swift file and comment out the material definition for the red metallic surfaces by adding two forward slashes to the line:

```
// let material = SimpleMaterial(color: .red, isMetallic: true)
```

Add a new line that initializes `material` to the occlusion material:

```
let material = OcclusionMaterial()
```

This is typically the material you would want in the `SceneReconstructor` class. The red metallic material was only used to help visualize what the reconstruction was doing behind the scenes.

Generating Random Objects

In this project, I'm finally breaking free from the shackles of the lowly sphere and adding in codes, cylinders, and cubes (oh my!). The `generateRandomSphere` function you've used repeatedly is evolving to handle the additional shapes. Let's get this crucial functionality out of the way now. Edit ImmersiveView.swift to include the new `generateRandomObject` function (and the old `getRandomColor` function) in **LISTING 8.3**. These should be placed inside the `ImmersiveView struct`, near the very bottom.

LISTING 8.3 Randomizing Objects and Materials

```
func generateRandomObject() -> ModelEntity {
    var object: ModelEntity
    let randomChoice = Int.random(in: 0...3)
    switch randomChoice {
    case 0:
        object = ModelEntity(mesh: .generateSphere(radius:
                            Float.random(in: 0.005...0.025)))
    case 1:
        object = ModelEntity(mesh: .generateCone(height:
                            Float.random(in: 0.01...0.09),
                            radius: Float.random(in: 0.02...0.03)))
    case 2:
        object = ModelEntity(mesh: .generateCylinder(height:
                            Float.random(in: 0.01...0.09),
                            radius: Float.random(in: 0.02...0.03)))
    default:
        object = ModelEntity(mesh: .generateBox(size:
                            Float.random(in: 0.01...0.05),
                            cornerRadius: Float.random(in: 0.0...0.009)))
    }

    let material : SimpleMaterial = SimpleMaterial(color:
                        getRandomColor(),
```

```
                        roughness: MaterialScalarParameter(
                        floatLiteral: Float.random(in: 0.0...1.0)),
                        isMetallic: Bool.random())
    object.model?.materials = [material]
    object.generateCollisionShapes(recursive: true)
    object.components.set(GroundingShadowComponent(castsShadow: true))
    object.physicsBody = PhysicsBodyComponent(
        massProperties: PhysicsMassProperties(mass: 2.0),
        material: .generate(friction: 1.0, restitution: 0.1),
        mode: .dynamic)
    object.physicsBody?.angularDamping = 0.1
    object.physicsBody?.linearDamping = 0.1
    return object
}

func getRandomColor() -> UIColor {
    let red = CGFloat.random(in: 0...1)
    let green = CGFloat.random(in: 0...1)
    let blue = CGFloat.random(in: 0...1)
    let color = UIColor(red: red, green: green, blue: blue, alpha: 1.0)
    return color
}
```

There are three primary additions to the `generateRandomObject` function versus the sphere-centric version you've been using.

First, you define a generic `ModelEntity` named `object`. To decide what kind of object it will be, a random integer between 0 and 3 is calculated and stored in `randomChoice`. A `switch` statement handles generating models from each of the possibilities of `randomChoice`:

0: Sphere

1: Cone

2: Cylinder

3 (or other): Box

The parameters (radius, height, and so on) of each shape are also randomized so that the appearance changes for each model entity created. The new randomized model is stored in `object`.

> **NOTE** There is no "logic" to any of the random numbers. I decided to go with relatively small hand-sized objects, but you can increase the size and fill your room with beach balls and traffic cones if you prefer.

The second change is that you use a new component with the object. For the first time, you cast shadows with the `GroundShadowComponent`:

```
object.components.set(GroundingShadowComponent(castsShadow: true))
```

The final change is to define slightly more physics than you have in the past:

```
object.physicsBody = PhysicsBodyComponent(
    massProperties: PhysicsMassProperties(mass: 2.0),
    material: .generate(friction: 1.0, restitution: 0.1),
    mode: .dynamic)
object.physicsBody?.angularDamping = 0.1
object.physicsBody?.linearDamping = 0.1
```

Within the `PhysicsBodyComponent`, I specify a mass of 2.0 kilogram. I found this value helpful for keeping the objects from bouncing everywhere at the slightest touch. Friction is set high (1.0), and restitution (bounciness) is low at 0.1. The physics mode is dynamic, meaning the objects can fully receive and transmit energy through collisions.

You also alter two additional physics body properties: `angularDamping` and `linearDamping`, which are values between 0 and infinity that define how quickly an object slows down when it is spinning or moving, respectively. You can play with all these values to see their effects. I used what I found to offer a pleasing experience after much trial and error.

The rest of the `generateRandomObject` (and `getRandomColor`) code is the same that you've already seen and used many times before.

Initializing the Data Providers

With the supporting functions under control, it's time to initialize and start the two data providers via the `HandTracker` and `SceneReconstructor` classes. At the top of the `ImmersiveView struct`, add these lines:

```
@ObservedObject var sceneReconstructor = SceneReconstructor()
@ObservedObject var handTracker = HandTracker()
```

Use normal (hand-tracking) and low-priority (scene reconstruction) tasks to start each of the detectors running. Add these lines directly following the `RealityView` code block:

```
.task() {
    await handTracker.startHandTracking()
}
.task(priority: .low) {
    await sceneReconstructor.startReconstruction()
}
```

Now, all you need to do is make the application do something interesting. You have two data detectors up and running, so let's make use of them.

Defining the Hand Objects

One of my goals with this project was to try to enable the user to use their hands to interact with the objects added to the Reality View using just the physics simulation. This isn't (currently) a particularly easy thing to do because your hands can't *feel* objects if you try to pick them up. Squeeze too hard and the object "squirts" out of your fingers. For this reason, I've decided to add a plane to the palms of my hands so that I can "scoop" objects into a hand or pick them up and drop them into a hand. In addition to the plane, adding spheres for the joints aids in the interactivity (and provides the ability to flick objects around or pull them toward you).

Importing ARKit

Because you need to access the finger joints by name, you need ARKit imported into the ImmersiveView.swift file. Add the required import line following the other `import` statements:

```
import ARKit
```

Creating Objects and Materials

Within the `RealityView` code, define the material to use for the finger joints as well as a `fingerObject` model entity that can be copied and used at each joint. This is virtually identical to what you did in the Hand Tracker project but with some additional physics properties and a clear material:

```
let material = UnlitMaterial(color: .clear)

let fingerObject = ModelEntity(
    mesh: .generateSphere(radius: 0.005),
    materials: [material]
)
fingerObject.physicsBody = PhysicsBodyComponent(
    massProperties: .default,
    material: .generate(friction: 1.0, restitution: 0.0),
    mode: .kinematic)
fingerObject.generateCollisionShapes(recursive: true)
```

This setup gives you a high-friction sphere you can use with your finger joints. The spheres are clear, so you can't see them, but they'll be able to interact with other objects. Note that the `physicsBody` `mode` is set to `.kinematic`, which means the object is being controlled by the user.

Next, define a `palmObject` that is used to cover the palm. It's a plane and uses the same clear material and physics properties as the finger joints. Add this code following the `fingerObject` definition:

```
let palmObject = ModelEntity(mesh: .generatePlane(width: 0.09, depth: 0.09),
materials: [material])
palmObject.physicsBody = PhysicsBodyComponent(
```

```
        massProperties: .default,
        material: .generate(friction: 1.0, restitution: 0.0),
        mode: .kinematic)
palmObject.generateCollisionShapes(recursive: true)
```

You now have a finger and a palm object that are configured and can be used for your finger joints and palms.

Adding the Palm Entities

The location of the palm is based on the wrist joint, but it's going to be offset slightly from the wrist so that it roughly covers the average person's palm. Define rightPalmObject and leftPalmObject as clones of the PalmObject and then adjust their positions like this:

```
let rightPalmObject = palmObject.clone(recursive: true)
let leftPalmObject = palmObject.clone(recursive: true)

leftPalmObject.position.x += 0.07
leftPalmObject.position.y += 0.02
rightPalmObject.position.x -= 0.07
rightPalmObject.position.y -= 0.02
```

> **NOTE** As a reminder, <variable> += <value> is the same as typing <variable> = <variable> + <value>. The same goes for the subtraction version: <variable> -= <value>.

These positions, like so many other things, were a matter of trial and error. You can set the color of the material to something other than clear and see for yourself where they sit. You may want to adjust them further for your needs.

Next, add the left and right palm objects to the wrist entity contained in the handTracker.leftHandParts and handTracker.rightHandParts.

```
handTracker.leftHandParts[.wrist]!.addChild(leftPalmObject)
handTracker.rightHandParts[.wrist]!.addChild(rightPalmObject)
```

Finally, add left and right wrist entities to the RealityView content:

```
content.add(handTracker.rightHandParts[.wrist]!)
content.add(handTracker.leftHandParts[.wrist]!)
```

Adding the Finger Joint Entities

The finger joints are handled with a loop, just as you did with the Hand Tracker project. Iterate through the joint names, accessing each entity in rightHandParts and leftHandParts. For each entity, the code adds a child containing a clone of the fingerObject ModelEntity:

```
for joint in HandSkeleton.JointName.allCases {
    handTracker.rightHandParts[joint]!.addChild(
                              fingerObject.clone(recursive: true))
    handTracker.leftHandParts[joint]!.addChild(
                              fingerObject.clone(recursive: true))
    content.add(handTracker.rightHandParts[joint]!)
    content.add(handTracker.leftHandParts[joint]!)
}
```

Each entity in each hand is then added to the `RealityView` content.

Managing the User-Added Objects

Each object (sphere, cylinder, box, sphere) a user creates will be added to a parent entity named `worldObjects`. Define this variable at the top of the `ImmersiveView struct`:

```
private var worldObjects = Entity()
```

After the content additions you've already made, set `worldObjects` to be an input target for indirect gestures. This is used in conjunction with a drag gesture to move objects around. Finally, add `worldObjects` to the `content`:

```
worldObjects.components.set(InputTargetComponent(
                                       allowedInputTypes: [.indirect]))
content.add(worldObjects)
```

As objects are added to `worldObjects`, they subsequently appear within the `RealityView`.

Adding the Scene Reconstruction Shapes

The other objects you need to include in the content are possibly the most important: the scene reconstruction model entities. Without these, user-added objects have nowhere to land, so they will fall... and fall.... and fall.

Add the `sceneReconstructor.parentEntity` to the `RealityView` code as well:

```
content.add(sceneReconstructor.parentEntity)
```

The code is in place to store user-added models, finger joints and palms, and the surfaces that make up the environment. The remainder of the project is setting up the gestures that turn the environment into a playground of shiny trinkets.

Creating Random Objects with the Tap Gesture

When a user wants to add an object to the environment, they perform a tap (pinch) gesture with either of their hands. The object is created and appears to fall from their hand position. In general, objects fall from the hand that performs the gesture—or at least the hand that is being looked at when the gesture is detected.

GESTURES AND CHIRALITY

Does that last paragraph sound non-committal to you? It should. There isn't a particularly convenient way to get which hand performed the tap gesture.

To estimate which hand performed a gesture, I chose to calculate the distance of both hands to the tap location of the gesture. Whichever is closer to the gesture location is the hand that releases the object. This doesn't always work, but it does have the helpful side effect of working quite consistently if you look at the hand you want to release the object.

Add a `SpatialTapGesture` after the closing brace in `RealityView`, as in **LISTING 8.4**.

LISTING 8.4 Detect and React to Tap Gestures

```
.gesture (
    SpatialTapGesture(count: 1)
        .targetedToAnyEntity()
        .onEnded { event in
            var releaseLocation = Transform()
            let tapLocation3D = Point3D(event.convert(event.location3D,
                                        from: .local, to: .scene))

            let distanceToRight = tapLocation3D.distance(to:
                        Point3D(handTracker.
                        rightHandParts[.indexFingerTip]!.position))
            let distanceToLeft = tapLocation3D.distance(to:
                        Point3D(handTracker.
                        leftHandParts[.indexFingerTip]!.position))

            if distanceToLeft<distanceToRight {
                releaseLocation =
                   handTracker.leftHandParts[.indexFingerTip]!.transform
            } else {
                releaseLocation =
                   handTracker.rightHandParts[.indexFingerTip]!.transform
            }

            let object = generateRandomObject()
            object.transform = releaseLocation
            object.position.y = object.position.y - 0.05

            worldObjects.addChild(object)
        }
)
```

The gesture block starts by declaring that a `SpatialTapGesture` with a count of `1` is the trigger. The gesture is then targeted to *any* entity with the `.targetedToAnyEntity()` modifier.

When the tap gesture ends (`.onEnded`), the calculations begin.

First, a release location (`releaseLocation`) for the random object is defined as an empty transformation matrix. Keep in mind that this is a transformation matrix, so it also carries orientation (rotation) information in addition to the location.

In this gesture, I make use of several instances of `Point3D`, a data structure containing x, y, and z coordinates in 3D space. `Point3D` also offers a useful `distance` function that calculates the distance to another `Point3D`.

The first use is in `tapLocation3D`, a Point3D data structure derived from the location where the spatial tap event took place, converted into world coordinates. Values `distanceToRight` and `distanceToLeft` are subsequently assigned using the Point3D `distance` function to find the distance between the `tapLocation3D` and the tip of the index finger on both the right and left hands.

If `distancetoLeft` value is larger than `distanceToRight`, you set the `releaseLocation` to be the same as the transform matrix of the left index finger entity. If not, you set it to the transform matrix of the right index finger.

Lastly, an object is generated from the `generateRandomObject` function, and its `transform` matrix is set to `releaseLocation`. For good measure, the object is lowered by adjusting its `y` position. This ensures that the object appears below the user's physical hand.

> **TIP** If the object is not released from a slightly lower position than the user's hand, there's a good chance it'll collide with some of the finger joint entities or the palm plane, making it bounce around. Lowering the release location reduces this possibility. You may even want to lower it further.

Finally, the object is added as a child to `worldObjects`, at which point it appears in the environment and falls to the surface below it.

The project is now in a testable state and can be launched on the Vision Pro. You should be able to add objects, interact with them, and move them around with your hands.

As I mentioned earlier, however, trying to pick up objects with your fingers can lead to frustration. You add one more gesture: an indirect drag gesture that will make it easier to grab and move any object anywhere in the environment.

Dragging Objects

The last major piece of functionality needed in the application is the ability to look at individual objects, and then drag them to other locations (including dropping them in a user's hands.) To do this, you use a second gesture—`DragGesture`—targeted to the `worldObjects` entity that contains anything a user adds to the environment.

Dragging objects that are moving or under the effect of gravity can have some strange side effects, so part of the code needs to "turn off" gravity for the duration of the drag.

Add the second gesture code block in **LISTING 8.5** directly after or before the `SpatialTapGesture`.

LISTING 8.5 Reposition Objects with a Drag Gesture

```
.gesture(
    DragGesture()
        .targetedToEntity(worldObjects)
        .onChanged { event in
            let object = event.entity as! ModelEntity
            object.physicsBody?.isAffectedByGravity = false
            object.physicsBody?.angularDamping = 1.0
            object.physicsBody?.linearDamping = 1.0
            object.position = event.convert(
                   event.location3D, from: .local, to: .scene)
        }
        .onEnded { event in
            let object = event.entity as! ModelEntity
            object.physicsBody?.isAffectedByGravity = true
            object.physicsBody?.angularDamping = 0.1
            object.physicsBody?.linearDamping = 0.1
        }
)
```

In this gesture, you make use of both the `.onChanged` and `.onEnded` events. In `.onChanged`, you assign `object` to the entity referenced by the event (`event.entity`). You typecast the entity to `ModelEntity` because you know that the objects added are model entities, and you need to access specific features of model entities, namely the physics body.

Next, these lines "turn off" gravity and stop any spin or other motion on the object:

```
object.physicsBody?.isAffectedByGravity = false
object.physicsBody?.angularDamping = 1.0
object.physicsBody?.linearDamping = 1.0
```

If the changes to the physics body are not included, the object moves in unexpected ways while it is being dragged.

During the drag, the object's `position` is updated in to match the event's `location3D` attribute but converted to world coordinates.

When the drag gesture ends (`.onEnded`), you once again assign `object` to the entity targeted by the drag and reset its physics properties to their defaults. This means that gravity once again takes effect, and the `object` falls onto the nearest surface.

For an interesting effect, you can try leaving gravity disabled. Objects can then be positioned in the air and just hang in empty space. It's cool, but do you really need any new ways to make a cluttered mess of your homes and office?

Cleaning Up

One last block and you're done! After `ImmersiveView` is dismissed, you need to remove the entities you've added outside of the initial `RealityView` setup.

Add the code in **LISTING 8.6** as yet another modifier to the `RealityView`, similar to what you've done in other projects:

LISTING 8.6 Remove Entities from the RealityView

```
.onDisappear {
    worldObjects.children.removeAll()
    for joint in HandSkeleton.JointName.allCases {
        handTracker.rightHandParts[joint]!.children.removeAll()
        handTracker.leftHandParts[joint]!.children.removeAll()
    }
    sceneReconstructor.parentEntity.children.removeAll()
}
```

This removes all `worldObjects`, all finger joints, and the surfaces added by the scene reconstruction, leaving a blank canvas for when the immersive view is opened again.

Run the application on your Apple Vision Pro and try scooping, throwing, and making a mess with the randomly generated objects. Cleaning up after throwing a tantrum just got much easier!

> **TIP** For those without a paid developer account, you can load a maximum of four development applications to your device. If you hit the limit, you get a warning message and need to remove some of the apps before more can be installed.

SUMMARY

In this chapter, you learned about some of the most useful tools for visionOS: scene reconstruction and occlusion. Using scene reconstruction, you can rebuild your entire environment using the Apple Vision Pro sensors and compute power. Successfully combining the real and virtual is the lynchpin of creating compelling experiences. Although Apple hasn't made this process as easy as it *could* be, it is still simple enough to include in everyday projects with the help of the reusable `SceneReconstructor` class.

You also explored advanced hand-tracking with the `HandTracker` class. This code takes the complexities of working with the ARKit hand skeleton and, again, turns it into a reusable piece of code that makes entities available for every single joint in both of a user's hands.

While there is still more ahead, you have what you need to build some fun and functional applications. I'll round out your primary toolkit over the next two chapters, then show you how you can prepare your creations to reach as wide an audience as possible via the App Store.

Go Further

I highly recommend downloading and exploring the source code for Apple's scene reconstruction example: https://developer.apple.com/documentation/visionos/incorporating-real-world-surroundings-in-an-immersive-experience. It may give you some good ideas of how to manipulate and place objects differently from what we've done in these examples.

It would also be good practice to go back to the Chapter 7 plane detection example and add scene reconstruction for more precise placement of the objects within the environment. Plane detection is a *much* less resource-intensive operation than scene reconstruction, so don't disregard it entirely, but scene reconstruction does a significantly better job of enabling your physical environment to accommodate virtual objects.

With hand-tracking, you now have access to all the data that visionOS can provide. Experiment with ways that hands can be involved in natural direct and indirect gestures. An important goal for any AR or VR developer is to make the actions the user performs feel as natural as possible. The more you can make your virtual world feel real, the better. Just adding the ability to flick an object if you want to move it feels incredibly satisfying and can make you forget you're staring at a piece of glass and metal.

INDEX

B

C

D

I

J–K

O

P

Q–R

S

T

U

V

W

X–Y–Z