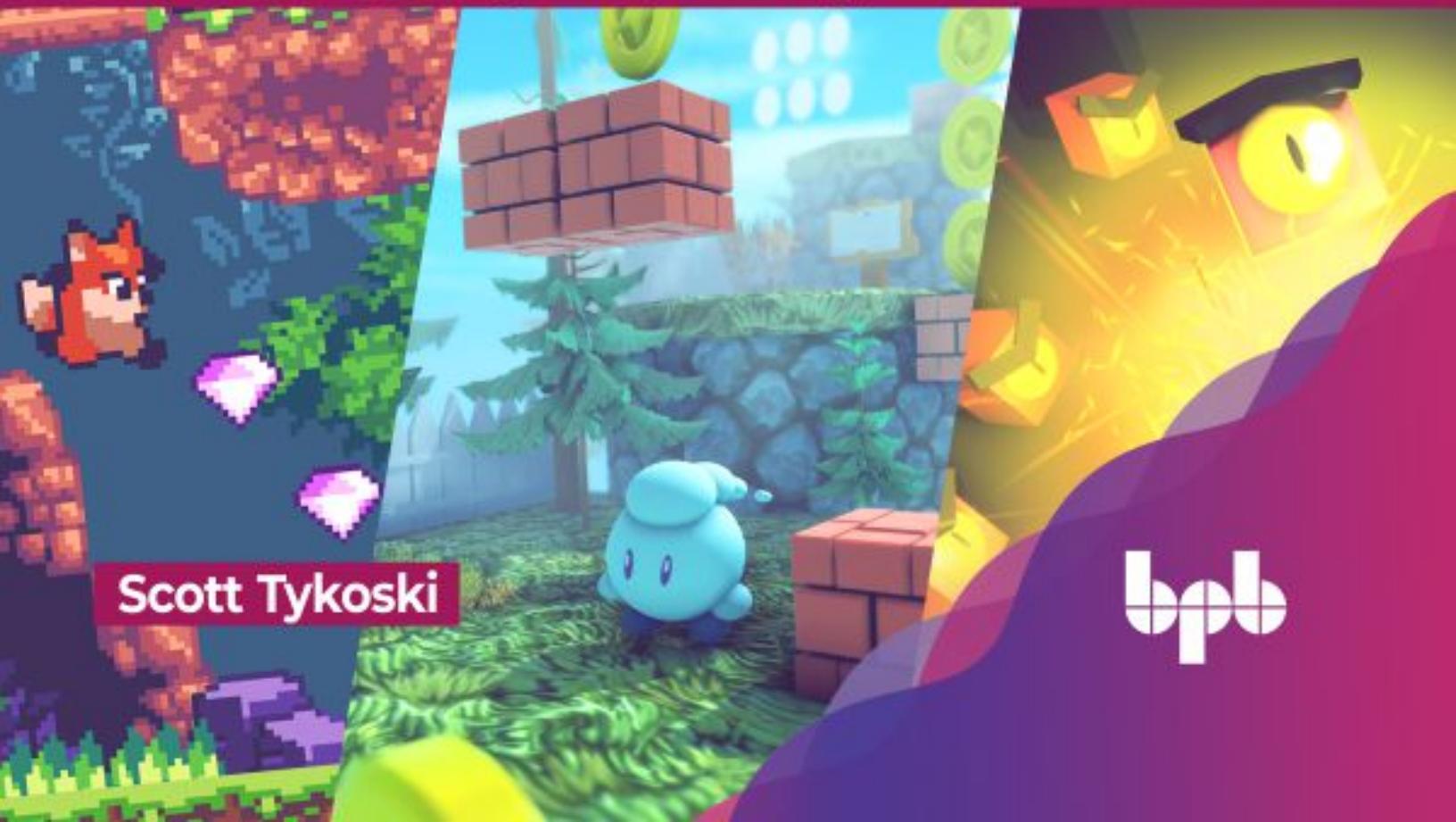


Mastering Game Design with Unity 2021

Immersive Workflows, Visual Scripting, Physics Engine, GameObjects,
Player Progression, Publishing, and a Lot More



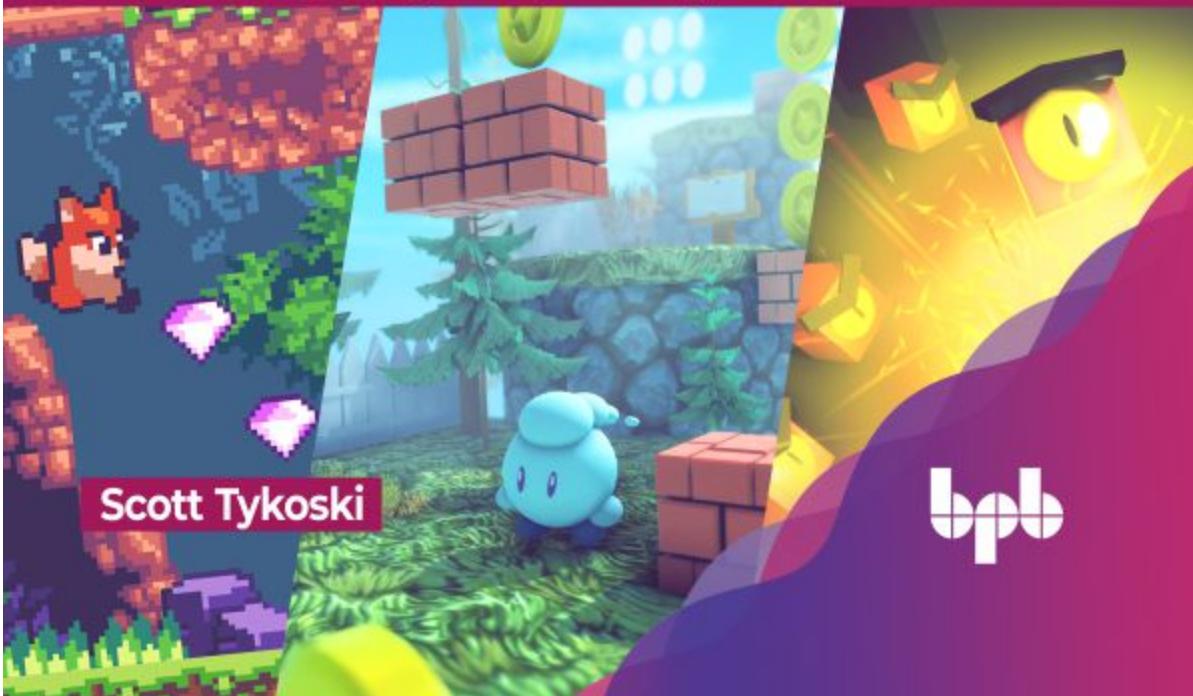
Scott Tykoski

bpb



Mastering Game Design with **Unity 2021**

Immersive Workflows, Visual Scripting, Physics Engine, GameObjects,
Player Progression, Publishing, and a Lot More



Scott Tykoski

bpb

Mastering Game Design with Unity 2021

*Immersive Workflows, Visual Scripting,
Physics Engine, GameObjects, Player
Progression, Publishing, and a Lot More*

Scott Tykoski



www.bpbonline.com

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2023

Published by BPB Online
WeWork, 119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-5551-217-8

www.bpbonline.com

Dedicated to

*My wonderful daughter **Lily**, who keeps me creative and energized.*

*My extraordinary son **Ethan**, who keeps me focused and organized.*

*My darling wife **Jenny**, who I could do none of this without.*

Thanks for helping me rotate those JPEGs.

About the Author

Scott Tykoski has been a part of the Video Game industry for the past 23 years, helping to ship 25 titles and applications across multiple platforms (PC, mobile, Xbox, Meta Quest 2, HoloLens). He led several teams and has filled almost every possible role throughout his career: developer, artist, designer, writer, animator, producer, marketing. He gives regular talks about game design to schools and local gaming dev meetups. Currently, he has taken his knowledge of Unity and design into the realm of Consultancy.

At OST, they use these emerging technologies to produce applications and AR/VR experiences for industry leaders.

Acknowledgement

There are so many people I want to thank for their support, both during the writing of this book, and the years leading up to it. First, a gigantic thank you to my wife, Jenny, and children, Ethan and Lily. Your love and support kept me going through it all. I can't wait to wrap-up so I can dedicate more time to our 'Mario Kart' family nights!

To my mom, dad, brother Mike, and sister Jodi. They say that the 'family that plays together, stays together'. I suppose all the game nights we shared put me on this track. Even though we no longer share a roof, I love knowing that our growing families will share a lifetime of joy playing games together.

I'd like to thank my former Video Game colleagues, from whom I've learned so much. Brad, Derek, Cari, Paul, Jesse, Charles, Sarah, Jeff, Kris, and everyone else I've had the pleasure of working alongside.

I want to thank the amazing team at BPB Publications, who walked me through this daunting process. I've learned so much, and I hope you share my pride in the book we've written.

And everyone else in my life that has supported me in this journey: Sandra, Allison, Jerry, Richard, Josh, Abby, Lizzy, Dianne, Finnley, Atleigh, Riley, Jim, Mo, and all my other friends and family that have been at my side over the years.

Preface

When it comes to amazing careers, working in the Video Game Industry is one of the best.

For 22 years I've had the immense pleasure of collaborating with coders, artists, and designers of the highest caliber. There was always something new to learn, and rarely was there a dull moment.

There's something undeniably satisfying about imagining, building, and releasing a game.

However, the thrill of creation, and joys of comradery, were sometimes overshadowed by a hard truth. Making games can be fairly difficult.

While there are plenty of opportunities to be creative, as well as opportunities to actually play awesome games (as, ahem, 'research') most of your time is spent solving hard problems. Coding, creating art, designing levels, architecting systems, then working with play testers to gather feedback. Once feedback is gathered (and remember, feedback is often critical and direct) you get to start the iteration loop all over again.

It can be hard, it can be frustrating, but, in the end, it can be highly rewarding.

This is why Unity is such an amazing engine and toolset. It reduces the stress and frustration of development, letting you focus on the fun, creative aspects of game design. The stuff that will set your game apart.

In this book, you will be learning game design theory, then applying these concepts directly in the Unity engine. We've broken it into 18 chapters, each tackling a different gameplay concept, with step-by-step instructions to implement these systems using Unity.

By the end, you'll understand what gameplay design entails, why certain production rules should be followed, and how to take your game from the conceptual stage to a completed product.

[**Chapter 1**](#) will give you an overview of the Unity Engine. It will walk you through the installation process, and guide you through the Editor, where

most of the game creation is done. You'll place a few objects and start to move them around using scripts.

[**Chapter 2**](#) takes you deeper into Unity's component and prefab system. Components are the C# scripts that we'll write then attach to our game objects. You'll also learn how to spawn pickup items (in this case, Coins) using pre-created objects, known as Prefabs.

[**Chapter 3**](#) teaches one of the fundamental gameplay concepts: player health. Most games will end when the player has taken enough damage, so we'll be hooking up this vital system using Unity objects and components.

[**Chapter 4**](#) is all about Unity's UI system. Learn how to use a canvas, images, buttons, and anchors to create a resolution independent screen. We'll also learn about Scene Management, and how to navigate from frontend screens to the main game.

[**Chapter 5**](#) takes a step back to ensure you know the fundamentals of Unity, specifically how to master the object manipulation tools when creating a level. This chapter also introduces you to playtesting, bug hunting, the importance of setting milestones, and how to create a 'build'. By the end, you'll have a version of your demo game that can be played by friends and family.

[**Chapter 6**](#) dives into Unity physics, the core system used to manage movement and collision between objects. Learn about the different components you'll be using, then use these systems to build additional level features in your game demo.

[**Chapter 7**](#) teaches a bit of animation theory and a lot about Unity's animation tools. Learn about, and utilize, concepts such as squash-and-stretch, easing, timing, keyframes, idle poses, and run cycles. Hook up new animations, ensuring our plucky hero exudes the proper amount of personality.

[**Chapter 8**](#) turns our attention from the hero to our enemies as you learn about AI (Artificial Intelligence). We'll cover some of the basic AI concepts, then build an enemy that can hunt down the player using Unity's built-in pathfinding systems.

[**Chapter 9**](#) lets you get crafty as we create a robust weapons system for the player. After discussing some weapon design theory, we'll be crafting both a sample Melee weapon and a Projectile weapon. We'll use our prefab

spawning knowledge to fire bullets, and get our hero animating properly when performing a melee attack.

[**Chapter 10**](#) is all about audio. We'll be discussing music, sound effects, and the wide variety of sound-related components Unity provides. We'll also start making use of the Unity Asset Store, where developers can grab music and sounds to use in their demo game.

[**Chapter 11**](#) introduces you to all the different ways Unity makes it easy to create amazing looking games. We'll start with Pro-Builder: an internal modeling tool for creating meshes directly in Unity. Then we'll jump back into the Unity Asset Store, discussing ways to leverage the talented artists in the Unity community. By the end of the chapter, you'll have an arsenal of materials, skyboxes, meshes, and post-processing effects, perfect for perfecting the look of any project.

[**Chapter 12**](#) teaches another vital tool for your graphical toolkit: the particle system. We'll learn about the various game systems that benefit from well crafted particles. After some discussion on design theory, we'll put our ideas to the test by crafting environmental, weapon, and explosive effects. The chapter wraps up with an overview of Unity's own 'particle sandbox' project - a great place to dissect expertly-made vFX samples.

[**Chapter 13**](#) steps away from graphical systems and focuses on the importance of player progression: design concepts that ensure your game excites the player without frustrating them. We cover a game's difficulty curve, pacing and reward systems, then implement these concepts with a level design example that teaches through gameplay.

[**Chapter 14**](#) covers the building of a great User Experience (UX). This somewhat nebulous concept has you taking the complexities of your game and simplifying them with well crafted UI and level design. We'll also discuss the all important 'First 10 Minutes' of your game, where you'll have limited time to hook modern gamers into your game world.

[**Chapter 15**](#) leaves the 3D world behind as we learn about the deep 2D toolset available in Unity. Sprites, tilemaps, 2D physics, and 2D animation are all covered as we consider both retro-themed games and modern 2D titles.

[**Chapter 16**](#) is a deep dive into the various game genres that players enjoy. We discuss design scope, player expectation, and the dangers of choosing a

genre that requires more effort than expected. Every genre overview will cover the history of the genre, the sub-genres that have emerged, and point you towards genre templates in the Asset Store (if you need a starting point). We'll wrap with a discussion on subverting expectations, and the ways you can take a genre and give it a unique twist.

[**Chapter 17**](#) covers the various hardware platforms that you can target with Unity. Because cross-platform development is a cornerstone of the toolset, it's important to understand the pros/cons of each option: desktop builds, mobile, console, and AR/VR. Based on the game you want to make, there's always an ideal platform for you to design for.

[**Chapter 18**](#) focuses on two of the most important aspects of completing a game: design and project management. Here you'll start solidifying a creative vision for your own game. We talk about pre-production tasks, where you'll document and visualize your game before writing any code. We'll talk about milestones and planning out tasks so you're always making progress. We'll end with tips on releasing your game and developing post-release updates that keep players engaged.

Code Bundle and Coloured Images

Please follow the link to download the **Code Bundle** and the **Coloured Images** of the book:

<https://rebrand.ly/49fe95>

The code bundle for the book is also hosted on GitHub at **<https://github.com/bpbpublications/Mastering-Game-Design-with-Unity-2021>**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **<https://github.com/bpbpublications>**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at **www.bpbonline.com** and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: **business@bpbonline.com** for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Table of Contents

1. The Unity Engine

Structure

Objective

Unity: The Broad Strokes

Installing Unity

Exploring Extras and Tutorials

Our First Project

Editor Overview

The Scene and Game View Window

The Scene

The Hierarchy Window

The Inspector Window

All About GameObjects

Placing Objects in the Scene

Moving Objects Around

The PLAY Button

Your Second Object

Using Visual Studio

Putting it All to the test

Optional Tool: Visual Scripting

Conclusion

Questions

Key Terms

2. Components and Prefabs

Structure

Objective

All about Components

Component Examples

Adding a Component to an Object

The MonoBehaviour class

Writing our ‘Player Controller’ Component

[Extending your Components in the Unity Editor](#)

[A Hop, Skip, and a Jump](#)

[Making and Spawning Prefabs](#)

[Coin of the Realm](#)

[Collision Detection](#)

[Triggers](#)

[Turning an Object into a Prefab](#)

[Spawning More Coins](#)

[Extra benefits of Prefabs](#)

[Debug Messages](#)

[Game Design 101: Iteration and White-Boxing](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

3. The Basics of Combat

[Structure](#)

[Objective](#)

[Game Design 101: Risk and Reward](#)

[Subsystems](#)

[Making our Health System](#)

[Let's Storyboard: Taking damage](#)

[Spikes: Your First Hazard](#)

[The HealthModifier Component](#)

[The Basic Bullet](#)

[Damage Feedback](#)

[The Game Session Manager](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

4. Getting to Know UI

[Structure](#)

[Objective](#)

[Game Flow \(UI\) vs Gameplay Flow \(Immersion\)](#)

[Game Flow Breakdown](#)

[The Unity UI System](#)

[Our Title Menu Canvas](#)

[Title Menu: Adding the UI Elements](#)

[Anchor Presets](#)

[Title Menu: Adding the Script](#)

[Title Menu: Button Actions](#)

[Adding Scenes to the Build Settings](#)

[A Basic HUD](#)

[The HUD Component](#)

[Game Over](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[5. Mastering the Fundamentals](#)

[Structure](#)

[Objective](#)

[The Importance of Milestones](#)

[Getting Critical](#)

[Mastering the Camera](#)

[Leveling Up](#)

[Tools of the Trade](#)

[Designing our First Level](#)

[Better Baddies](#)

[Explosive Feedback](#)

[Game Design 101: Permanence](#)

[Hunting For Bugs](#)

[Exporting ‘The Build’](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[6. The Physics of Fun](#)

[Overview](#)

[Objective](#)

[The Unity Physics System](#)

[Physics Components](#)

[Basic Collider Shapes](#)

[Mesh Collider](#)
[Rigidbody](#)
[Joints](#)
[Tweaking Physics in Code](#)
[Gameplay: Pushable Blocks](#)
[Gameplay: Seesaw Platform](#)
[Gameplay: Springboards](#)
[Gameplay: Wind Zones Strike Again](#)
[Utilizing Shadows](#)
[Gameplay Palette](#)
[Conclusion](#)
[Key Terms](#)
[Multiple Choice Questions](#)

7. The Joy of Animation

[Structure](#)
[Objective](#)
[Principles of Animation](#)
[Making A Basic Hero](#)
[The ‘Idle’ Pose](#)
[The Animation View](#)
[Keyframes and Inbetweening](#)
[Run Cycles](#)
[Animations in Action](#)
[Cleaning Up](#)
[Conclusion](#)
[Questions](#)
[Key Terms](#)

8. The Mind of the Enemy

[Structure](#)
[The Thought Process](#)
[Our Basic Enemy](#)
[New Component: AIBrain](#)
[UnityEvents and AI Actions](#)
[Hunting Down the Player](#)
[Setting up the AI Component](#)

[NavMesh](#)
[NavMesh Agents](#)
[Testing our Enemy](#)
[Conclusion](#)
[Questions](#)
[Key Terms](#)

9. Forging Your Weapon System

[Structure](#)
[Objectives](#)
[Sticks and Stones](#)
[Weapon Sandbox](#)
[Equipping Your Weapon](#)
[The Weapon Component](#)
[Basics of Melee Combat](#)
[Basics of Projectile Weaponry](#)
[The ‘Knockback’ Effect](#)
[Animating our Attacks](#)
[Weapon Trails](#)
[Game Design: Advanced Combat](#)
[Conclusion](#)
[Questions](#)
[Key Terms](#)

10. All About Audio

[Structure](#)
[Objectives](#)
[The Audio Components](#)
[Audio Listener](#)
[Audio Source](#)
[Importing Music and SoundFX](#)
[Importing Audio Clips](#)
[Creating our Sound Manager](#)
[The Basics of 3D Sound](#)
[3D Sound on Items](#)
[3D Sounds for Weapons](#)
[Mastering with Audio Mixers](#)

[Platform Considerations](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[11. A Graphical Upgrade](#)

[Structure](#)

[Objectives](#)

[Unity's Art Tools](#)

[Grabbing some Materials](#)

[ProBuilder](#)

[Building Blocks](#)

[Grid Snapping and Asset Placement](#)

[Building Level Prefabs](#)

[KitBashing](#)

[Skybox](#)

[Distance Fog](#)

[Keeping your Workspace Clean](#)

[Post Processing](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[12. So Many Particles](#)

[Structure](#)

[Objectives](#)

[About Particle Effects](#)

[Filling the 3D Space](#)

[Weapon Particles](#)

[Explosive Effects](#)

[The Unity Particle Pack](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[13. Mastering Player Progression](#)

[Structure](#)

[Objectives](#)

[The Difficulty Curve](#)

[Managing Difficulty](#)

[Onboarding and Tutorials](#)

[Teaching through Level Design](#)

[Lighting the Way](#)

[Teaching the Player](#)

[Testing the Player](#)

[Trials and Mastery](#)

[Pacing](#)

[Progression Trees](#)

[Achievements](#)

[New Game +](#)

[World Map](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

14. UX

[Structure](#)

[Objectives](#)

[UX?](#)

[Better UI for a Better UX](#)

[Iconography](#)

[Typography](#)

[World Space Prompts](#)

[The First 10 Minutes](#)

[Our Introductory Stage](#)

[Achieving Victory](#)

[A Smooth Transition](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

15. 2D vs. 3D

[Structure](#)

[Objectives](#)

[2D vs. 3D](#)
[Our 2D Sandbox](#)
[Sprites](#)
[Sprite Sheets](#)
[Tilemaps](#)
[Studying The Lost Crypt](#)
[2D Physics and Movement](#)
[2D Animation](#)
[Cinemachine Cameras](#)
[The Power of Parallax](#)
[Conclusion](#)
[Questions](#)
[Key Terms](#)

16. Mastering the Genres

[Structure](#)
[Objectives](#)
[Concerning Scope](#)
[Sidescrollers and Platformers](#)
[Top-Down Adventure](#)
[FPS - First Person Shooters](#)
[Third-Person Games](#)
[RPGs - Role Playing Games](#)
[Strategy Games](#)
[Puzzle Games](#)
[Creative Juxtaposition](#)
[Player Expectations](#)
[Conclusion](#)
[Questions](#)
[Key Terms](#)

17. Platforms and Publishing

[Structure](#)
[Objectives](#)
[Your Platform of Choice](#)
[PC / Mac / Linux](#)
[Mobile Games](#)

[Console Development](#)

[WebGL](#)

[VR/AR](#)

[Publishing.your Game](#)

[Teaming Up with Publishers](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

18. From Concept to Completion

[Structure](#)

[Objectives](#)

[Preproduction](#)

[Design Docs](#)

[Paper Prototyping](#)

[Concept Art](#)

[Mockups](#)

[Getting Creative](#)

[Project Management](#)

[The Vertical Slice](#)

[Play Testing](#)

[Analytics](#)

[Prioritizing Feedback](#)

[Milestones, Momentum and Morale](#)

[Early Access](#)

[Marketing](#)

[The Gold Master...and Beyond!](#)

[Conclusion](#)

[Questions](#)

[Key Terms](#)

[Index](#)

CHAPTER 1

The Unity Engine

We are in a Golden Age of game development.

It's amazing that, just 25 years ago, developing games looked *completely different*. The '16-bit' era was ending, the 3D era was beginning, and only large Studios had the money and muscle to create games for the masses.

During those early days of 3D, Studios were researching & developing gameplay systems as they went. Every new 3D concept came with a new major game release. Third-person camera controls and analog movement were the hallmark of *Super Mario 64*. It was *Halo: Combat Evolved* that introduced bump/normal mapping, giving objects unparalleled detail and realism. *Fable* introduced the Bloom effect, giving lit objects a fantasy-like appearance. And, more recently, *Minecraft* introduced the world to the concept of Voxels - a way to use 3D cubes to create objects, characters, or even randomly generated worlds.

Figure 1.1: Unity has been used to make blockbuster games such as *Ori: Will of the Wisps*, *Subnautica*, *Cuphead*, *Hearthstone* and *Pokemon Go!*

Video games, and the code they were built upon, were inseparably bonded. The concept of a **Game Engine** - a set of code and tools to build games upon, *available to all* - was only a *pipe dream*. Of course, if you had the money, there *were* early 3D engines that could be licensed, but the upfront cost created a steep barrier-to-entry that most developers couldn't overcome. If you wanted to make a game, you were coding everything yourself – *from scratch*.

Figure 1.2: Minecraft brought the concept of 3D 'Voxels' to gamers (and game designers).

Luckily, that Golden Age is upon us, and developers are *swimming* in game engine options. Developers that want to make a 2D game can use GameMaker Studio or Pico-8. Unreal, the powerhouse 3D engine that originally cost *millions* to license, can now be downloaded by anyone for immediate use. There are many options, each with their own strengths and weaknesses.

If you're looking for unrivaled flexibility, however, the most versatile option is, *by far*, the **Unity Engine**. Supporting 2D games, 3D games, VR and AR projects, PC, Mac, and Linux development, and even web games, **Unity** has the power to make *any* game you can imagine. Not only is it flexible - Unity's object-component system makes development and debugging *easy* and *fun*. The team at Unity is always working to improve their engine, releasing fresh updates to give you access to the latest technology. And the community of Unity developers continues to grow, eager to help other devs, and continuously creating impressive assets that can be downloaded from the **Unity Asset Store**.

With the Unity Engine, it's never been easier to create your own games.

So, let's get started!

Structure

- Unity: The Broad Strokes
- Using the Unity Hub
- LTE vs Beta Builds
- Exploring Extras and Tutorials
- The Unity Editor
- Placing Objects in the Scene
- Moving Objects around
- The Inspector Window
- The Hierarchy Window
- Making Something Move
- Installing Visual Studio
- Visual Scripting

Objective

[Chapter 1](#) is all about the *basics* of Unity. You'll be downloading the Unity tools and creating your first project. You'll learn about the different parts of the editor, where you'll create your first objects and add some simple **Physics**. By the end, you'll have written your first **Scripts** and hooked up some basic **Input** for changing the rotation of an object.

[Chapter 1](#) will give you both a strong understanding of the basics, a firm roadmap for what comes next, and a simple demo to build upon.

Unity: The Broad Strokes

Unity is one of the leading game engines, used by both independent and AAA Developers throughout the world. It can be used to make games for PC, Mac, mobile devices, consoles, and is the leading choice for VR/AR Developers. Unity allows you to make a game once, then build that project for almost any hardware platform out there.

As for the toolset, Unity's **Editor** is where you'll be spending most of your time. It allows you to quickly put together scenes, test gameplay in real time, and even pause your game if you need to debug a session in progress.

The robustness of the toolset is - in a word - phenomenal.

Coding is done using C#, a language that resembles C++, only a bit less complex (doesn't require header files, built in message handling, etc.). C# can also be compiled, helping it to run faster than most other scripting languages, and has its own memory manager so you don't have to worry about allocating and deallocating memory.

To write our code, we will be utilizing Visual Studio, which comes built-in when we install Unity.

The **Unity Hub** Is another important piece of the unity ecosystem. Here you can manage different versions of Unity, browse and download your projects, and collaborate with others in the community.

In fact, the Hub is where our journey begins.

Installing Unity

Unity provides a single tool for installing Unity and managing projects called the **Unity Hub**.

Figure 1.3: The Unity Hub – your one stop for managing projects, installing new Unity updates, viewing tutorials, and interacting with the expansive community.

1. Setting up the Hub is as easy as visiting <http://www.unity.com/downloads> and scrolling down until you find a Hub download link.

Beta vs Non-Beta: While searching, you may be presented with “Beta” options for the newest version of The Hub. For this book, it’s suggested that you download from a non-’Beta’ link. It’s often tempting to get the latest and greatest version of a Unity tool, but until you’re more familiar with Unity as a whole, we’d encourage you to use the existing, well-tested options.

2. Once you download UnityHubSetup.Exe, locate it in your Downloads folder, and run it. This will install the Unity Hub to your computer. It will also automatically launch it.
3. Now that we have the Hub, it’s time to install Unity itself. Press the **Installs** tab on the left side of the hub, then select the **Install Editor** button in the top left corner.

Figure 1.4: Press the Installs tab, then the Install Editor button to bring up the list of versions of Unity you can download.

4. This will bring up a list of the available versions of Unity you can download. We will be downloading the **2021 LTS** build.

Figure 1.5: You will be presented with several versions of Unity that can be downloaded. Select the latest 2021 LTS version.

About LTS and Beta Builds: There are two special types of builds you’ll see listed - LTS and Beta. LTS stands for “Long Term Support”. Unity provides two years of support for their releases, so 2021 LTS is a stable, feature complete version of the 2021 build. Beta builds, on the other hand, are the newest and coolest builds of Unity, but may

contain features still being implemented (and the possibility of bugs) . Since we’re looking for stability, we’ll download the latest LTS build. Once you’re more comfortable with Unity, you may want to jump into the Beta builds and help shape the future of the platform.

5. After selecting the Unity 2021 LTS release, press the **Next** button in the lower right corner.
6. You’ll then get a list of optional Modules that you can install. For now, let’s disable everything except for “Documentation” and press **Done**, which will start the download and Installation process.

Figure 1.6: You can install Modules at any time, so don’t worry about missing something during your initial setup.

With our build of Unity installed, let’s take one last look around the Hub before starting our new project.

Exploring Extras and Tutorials

While the unity Hub is primarily for managing your projects and builds of the unity engine, there are some tabs worth investigating.

The **Learn** tab, for instance, gives you access to a deep well of knowledge directly from the team at Unity. Game-design tutorials, demos, and templates are all Made available with a click of a button. Each of these is highly polished and will always result in new knowledge.

Figure 1.7: Unity provides a wealth of learning content directly in the Hub.

Let’s say you want to develop your own first person shooter game. There is an FPS Creator kit option, designed with beginner developers in mind, that can be completed in under 2 hours.

Or perhaps you’re trying to get a younger developer excited about coding. Unity has teamed up with Lego to make a series of microgame tutorials. These allow novice developers to play with block building as they learn how to code games.

This tab alone is a deep well of useful knowledge for any aspiring game developer.

There is also a tab for the **Community**, where you can read up on what the Unity team has planned for upcoming features and releases. You can also get help directly from the team, or join the forums, where an incredibly passionate and active community of Unity Developers would be happy to help with any problems you run into.

***Figure 1.8:** The Unity Community is very friendly, so don't be afraid to ask questions and look for feedback!*

And while these tabs have an exciting amount of content to dive into, we're going to start fresh with our own blank project. Return to the Projects tab and press the **New Project** button in the upper right corner.

***Figure 1.9:** The ‘New Project’ Button will get us started.*

Our First Project

1. Pressing the New button will bring up the “Create a New Project” dialog, giving you several options to choose from, as shown in [figure 1.10](#)

***Figure 1.10:** There are many templates to choose from when starting a new project.*

2. We will return to these templates in later chapters, but for now select the “3D” template option.
3. This is also where we will set our project name. Select the field that says **New Unity Project** and change that to **Mastering Unity Demo**. Let's also organize all our Unity Projects into a **Project** directory. Once you have that folder available, set that as our project location.

***Figure 1.11:** Your first project is being created. This may take a few minutes, but it's worth the wait.*

4. Press next, and Unity will start generating all the files and folders needed for your new project. This is a lengthy process, and depending

on the template you've selected, setup will take between 2 and 4 minutes.

Once complete, you'll be presented with Unity's main tool: the Editor.

Editor Overview

Welcome to the heart of game creation in Unity: The **Editor**.

Figure 1.12: The Unity Editor

There is a lot to take in here, so let's take a quick walkthrough of its key sections.

- **Views Window:** Directly in the center of the screen is the Scene/Game View window. This is where you can edit your scenes and levels, as well as test them in real time.
- **Hierarchy Window:** To the left of the Views panel is the Hierarchy list window. This contains all the objects in the current scene.
- **Inspector Window:** On the right is the inspector window, which shows all of the data related to any selected object.
- **Toolbar:** The vertical list of buttons in the Scene View is the Toolbar. This gives you the tools needed to manipulate objects. (move, resize, rotate, etc.)
- **Project Window:** All the assets related to this project. 3D models, 2D textures, sound and music files, code, and anything else required for your game to look, sound, and play as you want it to.
- **Console Tab:** Pressing the console tab brings up a debugging window called the Console. This window shows output and errors that may occur in our code.
- **Play, Pause, and Step Buttons:** Use these to test and debug your game project.

Now that you know the basics of the editor, let's dive into the centerpiece: the Scene View window.

The Scene and Game View Window

Right in the middle of everything is the Viewport window, which consists of two parts: a Scene View and a Game View. Each of these modes have a respective tab.

Figure 1.13: The Views window in “Scene” Mode.

Pressing the Scene tab when you want to edit your game. This mode lets you see all the objects, select them using your mouse, and edit them in real time. You can drag objects into your scene, move them around with the gizmos, and view their details In the inspector panel to the right.

Pressing the game tab will bring up the game as it'll be seen by the player. This is the tab you press when you want to critique visual effects, user interface look and feel, and playtest your project.

Let's keep the viewport in Scene mode until we have something worth playtesting.

The Scene

Since we're in Scene mode, it would be a good time to talk about the scene itself.

All objects in a Unity project will be placed into a 3D **Scene**. Think of it as a sandbox where all your enemies, power-ups, and level design objects will live. Scenes can be filled using the unity editor (by dragging objects into it) or at runtime (using code).

Scenes are created and managed in the Project window. A project can have multiple scenes, with many game developers Using a new scene for every level in their game.

For now, we will keep things simple and only make use of our initial scene.

The Hierarchy Window

On the left-hand side of the editor, you will find the Hierarchy Window. This is a list of everything in the scene, giving you a single spot to search for and select specific objects:

Figure 1.14: The Hierarchy Window - the master list of the objects in the scene.

At the top of this window is a **Searchbar**, where you can narrow down the listed object. For instance, if you click where the search field says **All**, and enter **camera**, you'll see that the Directional Light object disappears and only the Camera remains.

To the left of the search field there is a + button. This is a quick way to add more objects to the scene. To the right of the search field, there is a button that opens an advanced search window, where you can search for a given phrase across your entire project.

Following the search bar is the name of the current scene, which is currently **SampleScene**. Under the Scene Name is a list of all the objects in the scene. Right now, only two objects are shown: the Main Camera and a Directional Light.

If you look over at the Scene View, you can also find the listed objects shown there, represented by 2 small icons known as **Gizmos**. Select these objects in the hierarchy and note that they also get selected in the Scene view:

Figure 1.15: The default Camera and Directional Light in the Scene. Note that selecting them in the Hierarchy will also select them in the Scene View Window.

The **Camera** gizmo represents where the player will be viewing the scene from.

The Sun gizmo represents the default **Directional Light** for this scene, storing the direction, intensity, color, and shadow settings for this scene.

As your project grows, expect this list to get very long and very complex. Soon we'll learn about Parent/Child hierarchy, which will allow us to organize multiple objects in a single parent object.

For now, however, let's work on editing one of these objects.

The Inspector Window

1. Select the Directional Light object by either clicking on it in the hierarchy, or by selecting the 'sun' gizmo in the Scene View window.

- Once selected, the settings for the directional light will fill the window on the right-hand side. This is called the inspector Window, where you will be editing objects in your game:

Figure 1.16: Selecting the Directional Light gives you access to all its settings in the Inspector.

- Because we've selected a Directional Light object, many of the options listed will be directly related to lighting. As shown in [Figure 1.16](#), you'll have access to the Type of light, Color, Intensity, Shadow options, and more.
- Now select the camera object, either in the hierarchy panel or by clicking its Gizmo in the Scene View. The inspector panel will update accordingly, listing all the available data related to the camera: position, rotation, field of view, and so on:

Figure 1.17: The Camera Inspector

For now, we will keep all these settings as they are. Instead, let's add some new GameObjects into your scene.

All About GameObjects

The GameObject is the basic building block of all Unity games. Whether it's a character, a weapon, a building, an Interface Element, or even the camera or directional light - each of these is built from the basic GameObject.

While most objects in your game will be imported from external asset creation tools (Maya, 3dsMax Photoshop, and so on), Unity comes with a handful of basic objects to play with. Cubes, spheres, planes, and cylinders will give us all the basic objects we need to get our demo game started.

Placing Objects in the Scene

- The most straightforward way to place an object into your scene is to use the right click menu in the Hierarchy panel. Bring up the menu,

navigate down to **3D Object**, and select **Sphere**:

Figure 1.18: Right-click in the Hierarchy List to quickly add objects to the scene.

2. Selecting *Sphere* from this menu will place a 3D sphere object into the scene directly in the center of the viewport. You can also place a sphere from the top Menu Bar by selecting *GameObject > 3D Object > Sphere*. Either way you do it, your scene will now have a *Sphere* object in the center of the Scene View:

Figure 1.19: The new sphere in your scene. It's so round!

3. The 3D Objects that come packaged in Unity also have several pre-assigned components. Select our new *Sphere* object to see all its information listed in the inspector panel:

Figure 1.20: The Inspector panel for our new Sphere.

4. At the top you will see the **Object Name**. This is for internal use and will never be displayed to the player themselves. Feel free to come up with a naming system to give your objects instantly identifiable names. For example, let's name this one **PlayerObj_Sphere**. Click on the existing name and enter *PlayerObj_Sphere'* as our new one.
5. To the left of the name is a small check box to enable or disable this object:

Figure 1.21: Object Name Panel

6. Click on it once to see the *Sphere* disappear in the Scene View. You'll also see the object name turn gray in the Hierarchy panel. Note that, when you toggle it again, the sphere reappears, and its name is no longer greyed out.
7. This top section also includes options for Tag, Layer, and Static. These are advanced options that we will dive into later, so keep them at their defaults for now.

- Following this top section is the **Transform** panel. This is an important section that shows where the object is in 3D space, as well as its rotation and scale information:

Figure 1.22: Transform panel

Local vs World Space: The location listed here is in “Local Space”. This means that these positions are relative to the parent object. If an object says its X position is 5, but its parent Object is at -5, The “World Space” position would be 0. We’ll dive deeper into Parent/Child objects in a later chapter.

While the transform data is laid out in text fields, you can also click on one of the 3 axes (X, Y, Z), hold the left Mouse button down, and drag your cursor to change this value. This is a quick way to make broad changes to these values without needing the keyboard.

- Below the Transform component, we have several areas with Mesh and Rendering information:

Figure 1.23: Mesh Rendering Components

Rendering Meshes: A Mesh is a collection of triangles that represents a 3D object. Rendering is the process of taking those triangles and drawing them to the screen.

The Mesh Filter is simply a GameObject component that specifies the 3D mesh to display. The Mesh Renderer component has all the material, lighting, visual settings to be used when rendering the specified mesh.

Again, for now, we will keep these default settings.

Below the rendering data, you will find a Sphere Collider panel. Colliders are components that tell the physics system what the shape of a GameObject is. In this case, our sphere mesh makes use of a Sphere Collider:

Figure 1.24: The Sphere Collider Panel

The last panel in the inspector panel shows us the material being used for this game object. Included GameObjects - such as the Sphere - will use the default material, which is a white plastic looking material.

We will learn more about materials, textures and meshes in our chapter on Graphics, but for now our demo game will use default visuals.

Moving Objects Around

Now that we have our sphere placed in the 3D World, The Next Step is learning how to move it around in the Scene.

The Editor gives us two ways to do this: The transform panel and the movement Gizmo.

First, select the object you want to move around the scene. In this case, you can left click on the Sphere we just added.

If you know precisely where the object should be in 3D space, you'll want to use the transform panel we mentioned earlier. This is the area in the inspector panel right under the name of the object. Here you can type in the specific X, Y, and Z location to place the object at. You can do the same thing with scale and rotation.

There's also the transform Gizmo, which are the arrows that appear when you select the object. each of these arrows correspond with one of the axes that you can move the object to: X axis is red, Y axis is green, Z axis is blue. Use this opportunity to drag the movement Gizmo around and get an idea for how it feels. If you click directly on an arrow, you will move only in that axis. You can also select the object and drag it around on all axes, but it's an imprecise option, so use it sparingly:

Figure 1.25: The transform gizmo, in ‘Movement’ Mode.

In this case, we actually know the precise location we want our object placed. With the sphere selected, look at the right side of the screen to find the Transform Panel. Select on the entry field to the right of X and enter ‘0’. Set Y to ‘3’ and Z to ‘-2’.

When done, the position values should look like this:

Figure 1.26: Our Sphere's position, with the updated values.

Now, to ensure we entered the information correctly, let's check the Sphere in the Game View. The Game View is the scene as viewed through the Main Camera, with all lighting effects the player will see in the game.

1. Select the **Game** tab in the Views Window. If we did everything right, the sphere should be just above the halfway point in the Game View window.

Figure 1.27: If the Sphere looks like this, then you've entered the data correctly.

With the Sphere object in place, we're ready to put our little demo to the test.

The PLAY Button

At the top of the screen, almost directly in the middle, there are three very important buttons: Play, Pause, and Skip. **Play** will start (and stop) running your project. **Pause** will pause and unpause a running project. The last button is **Skip**, which will pause a game and advance to the next frame – useful when debugging.

Press the Play button now.

Figure 1.28: The Play, Pause and Skip buttons.

What you will see is the sphere...doing nothing. How terribly unexciting. I apologize for the buildup.

This is because the Sphere needs a RigidBody component to officially hook into the physics system.

Let's first stop the simulation by pressing the Play button, which will turn from Blue to Grey. Once the game is stopped, select the Sphere. At the bottom of the Inspector panel, you'll find an **Add Component** button. Press it to bring up the Components panel.

Figure 1.29: Adding a Rigidbody to our Sphere.

Select the **Physics** option, then **Rigidbody**. This will place a Rigidbody component on your Sphere object, officially making it a ‘Physics’ object.

While we’ll be diving into Physics in a later chapter, you can take a moment to look over the component we just added.

Figure 1.30: Rigidbody Inspector Panel

There is a lot of options here. Some are pretty obvious, like “Use Gravity”. Others less obvious, like “Is Kinematic”. Like with most components, let’s keep everything at their default value until we understand them better.

Now let’s press Play again. Just like magic, we see the Rigidbody component in action. The Sphere, now being affected by the physics system, is pulled downwards by gravity. One second later it’s gone, and we’re stuck staring at a blank screen. Now what do we do?

First we press the play button again to stop the game. This returns all the objects in the scene to the way they were positioned before the game started.

Second, we need some ground. So let’s place our second object.

The Importance of Stopping your Game: It may seem like a silly thing to be warned about but remembering to press that play button to stop a running game is VERY important. Because everything reverts to its starting state, it’s easy to accidentally make - and lose - changes to your project by implementing those changes during run-time. Your best practice is to ALWAYS stop your game, and verify it’s no longer running, when you need to make changes to your project. This is an important concept that, if ignored, will lead to lost work and frustration.

Your Second Object

To keep our player object from plummeting into the void, let’s add some ground for them to interact with.

1. Like with the sphere, right click in the hierarchy window and add a Cube to the scene.

Figure 1.31: Our new Cube, in its 6-sided glory!

2. Once placed, left click the cube to select it. We can now edit its position, scale, and rotation – directly in the Inspector window - to flatten the new object more ground-like.

Table 1.1: The Transform of the new Cube Object.

3. With the position values, we've moved the Cube to be located below the Sphere. The Rotation has a very slight tilt to it, as set by the 1 in the Z Rotation field. And the Scale has the cube stretched in the X and Z axes but flattened in the Y axis.

Figure 1.32: Your ground object Cube should now be flattened to look like this.

4. While we're here, let's give this object a more informative name. Click on its current name "Cube" and rename it to **LevelObj_Ground**.
5. Now that we have a ground object, as well as our player sphere, let's see what happens when we press the play button.
6. With a ground object beneath the falling sphere, the downward motion of our ball is almost instantly stopped by the ground. Then, because the ground object has a slight incline to it (thanks to Z = 1 in the Rotation panel) the sphere will slowly roll to the left, as it continues to be affected by gravity.

How exciting!

Using Visual Studio

At this point our game consists of a ball and a flattened cube. It's actually *not* that exciting.

Let's add some interaction to our objects.

This is a good time to introduce the second most important tool in your game making Arsenal: Visual Studio.

Automatically set up when you first installed Unity, Visual Studio is the preferred code editing tool of the Unity Engine. In it, you can create classes

that expose important data back into the unity editor, allowing you to actually extend the functionality of the editor by simply adding the right tags.

Visual Studio also has a system called **Intellisense**, that will expose functionality as you write it. Well experienced Developers will have used intellisense before, beginner coders will be happy to know that all the functionality you'll need the right will be shown to you through intellisense. It's pretty awesome.

To get started, we need to create our first script.

Scripts are the text files that contain code for our Unity games. specifically, scripts are written in the C# language (pronounced “C Sharp”).

We're going to write a script that allows us to tilt our Cube with the arrow keys, giving our sphere something to roll around on.

Since we'll be writing code that affects the Ground cube, we'll want to add this new script directly to our new “LevelObj_Ground” object. Select the cube to bring up the inspector panel. scroll down to the bottom of the inspector panel and press the **Add Component** button.

Figure 1.33: Add Components to your objects with this aptly named button.

1. A panel with a list of options will pop up. Scroll all the way down in this list and select **New Script**. When it asks for the name of this new script, enter “GroundObjectController”, then press the **Create and Add** button.
2. Pressing the enter key finalizes the name of your script, creates the file and class, and automatically adds it to the selected object.

Figure 1.34: This is the new script, as added to the Ground object.

3. To start editing this code, click on the grey box in the lower right corner that says “GroundObjectController”. This will open that script in Visual Studio.
4. If you're new to Unity development, much of what you see in the file is going to the unfamiliar. That's expected, which is why we will dive deeper into Components in a later chapter.

5. For now, simply find the line **void Update()**. This is the update function of any gameObject script. As you may have guessed any code present in **Update()** will be called every frame, giving you the perfect spot to tell this object how to *move* and *react*.
6. In this case, we simply want to let the user rotate the ‘ground’ object in two directions.
7. Add the following code into the **Update()** function...

```
// the speed that this object rotates
float speed = 15f;
// get the current rotation of the ground object
// based on the keys pressed
// we will be altering this value and re-applying it
Vector3 newRotation = transform.localEulerAngles;
// go through each of the direction keys: up, down, left,
right
// if any of them are pressed, alter the rotation of the
ground plane
if (Input.GetKey(KeyCode.RightArrow))
    newRotation.z -= Time.deltaTime * speed;

if (Input.GetKey(KeyCode.LeftArrow))
    newRotation.z += Time.deltaTime * speed;

if (Input.GetKey(KeyCode.UpArrow))
    newRotation.x += Time.deltaTime * speed;

if (Input.GetKey(KeyCode.DownArrow))
    newRotation.x -= Time.deltaTime * speed;

// apply the updated rotation to the ground object
transform.localEulerAngles = newRotation;
```

8. This block of code will check if any arrow keys have been pressed by the player. If so, we apply a simple rotation to that object: in this case, our ground cube.

Commenting your Code: If you’re new to game development, you may wonder what those // lines are for. Those are called **COMMENTS**, and they allow the developer to explain code in a way that’s easy to read. Comments are ignored when the code is run, and since code can be

hard to read, it's suggested that you comment your code whenever possible.

An impressive feature of Visual Studio is that it's always checking to see if the code has any errors (if it has errors, the code will not run). If we wrote these lines correctly, we should be able to click on the output window in Visual Studio and see that no errors have been found.

If errors have been found, Visual Studio will direct you to the problematic code. Simply go to any line with errors and double-check that it matches the sample code above.

Putting it All to the test

Our demo now has objects to play with and scripts that manage player interaction. Now it's time to put your hard work to the test.

Select the Game tab in the viewport panel and click the option "Play Maximized". Now, when we press the play button, the Game View will fill the screen, bringing us closer to how the game will feel when exported to the final build.

Pressing the arrow keys should rotate our 'flat cube' as expected. The ball will roll based on how you're tilting the ground object. And even without weapons, enemies, music, or fancy graphics, trying to keep that ball from falling into the abyss shows the *distant glimmer* of a fun game.

Optional Tool: Visual Scripting

In this chapter, you were introduced to Visual Studio and C#. By writing a short script, you were able to take input from the keyboard and use it to rotate an object.

Using Visual Studio for writing and debugging your game code is the preferable method of development. Once you're familiar with writing code, it's an *incredibly* useful skill that can lead to a professional career in software development across hundreds of industries.

It's also a skill that - at first - can feel daunting.

To help ease incoming developers into the world of coding games, Unity now comes pre-packaged with a **Visual Scripting** system.

Figure 1.35: A visual way to code, for those that prefer it.

For visual learners that want to see their logic graphed out, the Visual Scripting tool is a node-based solution for writing code. By connecting these nodes, you get almost all of the flexibility of writing code, without the headaches of reading and parsing it.

Be warned, however, that while this is a great tool for learning and writing smaller systems, the more complexity you add, the more difficult it becomes to manage this style of development.

Conclusion

Congratulations! You now have a project with two objects: a sphere that rolls around, and a ground plane that can be tilted as arrow keys are pressed.

It's a humble beginning, but all major releases start with a *basic foundation*. Every game that has captivated players began simply - with a just few objects and a sprinkling of interaction.

So...it may be simple, but we're in good company.

And while we're appreciating this milestone, let's list the game development systems you're making use of:

- Object Rendering
- Physics Simulation
- Input Handling
- Real-time Scene Editor

The demo you've made, along with the subsystems you've tapped into, are built upon years of work done by teams of professional, passionate developers. Work that would have taken a solo developer *months* - if not *years* - to perfect on their own.

Everything they've made - every system, mechanic, and tool they've built - are now *yours* to play with and create with.

The Golden Age of game development is here.

And we're just getting started.

Questions

1. What are some of the benefits to using Unity over building your own Game Engine?
2. True or False: If you want the most stable build of Unity, you should use a Beta or Alpha release?
3. What are the differences between LTS and Beta builds of Unity? Why will we be using a LTS build while learning the tools?
4. What gets listed in the Hierarchy Window? What about the Inspector Window?
5. What happens when you press the Play button when the game is running? What about the Skip button?
6. The Transform of a game object contains three things: the Position of the object, the Rotation of the object, and what else?
7. Colliders are important components that define what aspect of an object?
8. Unity comes with several 3D Objects that you can place in your Scene. And while each comes with a Collider component, an additional component needs to be added for the object to respond to the Physics system. What is the name of this Physics component?

In the Unity Editor, you can press Play to start running the game. If you make changes to objects while the game is running, do those changes get permanently applied? Or do they get reset when the game is stopped?

Key Terms

- **Game Engine:** A set of code, systems, and tools that have been developed with the single purpose of creating Video Games. Some engines are tied to a specific series, while other are *game agnostic*, providing a great foundation for building any new project.
- **Unity Engine:** The Unity Engine is a modern Video Game engine that was designed to be used by *anyone*. It has the tools, systems, and raw power to create amazing projects without needing a large team.

- **Unity Hub:** A app that organizes both your projects and Unity builds. The Hub is also where you'll start a new project and connect with the Unity Community of developers.
- **Unity Asset Store:** A website where you can find sample projects, models, textures, gameplay systems, and anything else you can imagine needing for a game. As you learn more about Unity and build a skillset, you can even sell your own creations on this storefront.
- **The Editor:** The Editor is the main tool for creating games in Unity. Here, you will see a 3D scene where you can place objects, adjust lights and cameras, and build objects, levels, and even entire worlds.
- **Game Object:** Any object placed in a Unity is based on a core object type called a GameObject. GameObjects can have Components added to them and have a position in 3D space, known as a Transform.
- **Transform:** The position, rotation, and scale of a game object in 3D space. An object's transform can be changed in the editor or by adjusting it in code.
- **Hierarchy:** The list on the left side of the editor. It shows all the individual objects that make up the current Scene. At the top of the Hierarchy, there's a useful search bar that can narrow down the visible objects in this list.
- **Inspector:** All the components that make up the selected GameObject are listed on the right side of the editor, in a window called the Inspector. After clicking on an object, you can edit its parameters in the Inspector Window.
- **Project Window:** This window shows the folders and assets that make up this project.
- **Console:** A window that shows errors and debug output from the game code.
- **Visual Studio:** The program used to write your game code in. Gets installed automatically when you install Unity.

CHAPTER 2

Components and Prefabs

The Unity engine gives us many powerful tools to make the games we want: a robust editor, an assortment of assets, and a wealth of systems to build from (physics, audio, rendering, and so on).

One of Unity's most underrated benefits, however, is the ease with which you can build unique GameObjects using the **Component** and **Prefab** systems. Using a handful of components, then spawning them using the prefab system, you can create almost anything – Players, Enemies, Items, Weapon, Bullets. With these two systems, you'll have the tools to start building everything your project needs to become the game you envision!

Structure

- All About Components
- Component Examples
- Adding Components to Objects
- The ‘MonoBehaviour’ Class
- Making a Hero that Runs and Jumps
- Handling Collisions
- Creating Prefabs
- Spawning Prefabs in Code
- Basic Debugging with `Debug.Log()`
- White-Boxing: What is it, and why is it important?

Objective

After reading this chapter, you'll have a full understanding of Unity's Component and Prefab systems, which allow you to create objects for your game, and spawn objects within your game.

With Components, you'll first be looking over some example cases, where you can see how this component-based system is used a wide variety of object types. You'll then get to build a Player object, complete with input handling, and a spinning gold Coin for the player to pick up.

By the end of the chapter, you'll have created several versatile components, spawned several new prefab objects, and learned several powerful tools on your path towards game development mastery!

All about Components

You've already learned about the core building block of Unity games: the `GameObject`. And while they give us a solid foundation to build a game from, their real power comes from **Components**: modular bits of code that can be combined to define the gameplay purpose of an object.

In [Chapter 1](#), you were introduced to several components as you created your Sphere and Ground objects. The **RigidBody**, **SphereCollider**, and **MeshRenderer** panels that you saw were all premade Components. You even made your own component: the **GroundObjectController**, a script that rotated the ground object as the arrow keys were pressed.

Whether you're using pre-made Components, or you're writing your own, these scripts can be built to handle anything: controller input, AI logic, particle effects, or even the management of networking for multiplayer objects.

In short: *everything* in your game will be built from Components.

Let's looks at some more examples of components in practice.

Component Examples

The modular nature of Components can sound intimidating at first. Examining some real-world examples of Components and `GameObjects` may help to clarify this method of development.

Almost all games have the concept of a 'Player' object. This can be the character, camera, or UI cursor that is moved by the user. This is Mario, Master Chief, or even your current Tetris piece.

The Player object responds to input from the user, either through controller, mouse, or keyboard. When the user presses 'Left', 'Right', 'Accept', 'Jump', or 'Fire', the Player object needs to respond accordingly.

In a third-person platformer, for instance, the Player object would be the visible hero that the camera follows. They would be built from Components related to moving in 3D space, managing Health, storing Inventory, playing Animation components, and handling Physics.

The Inspector for this 3rd-Person Player object would look something like this...

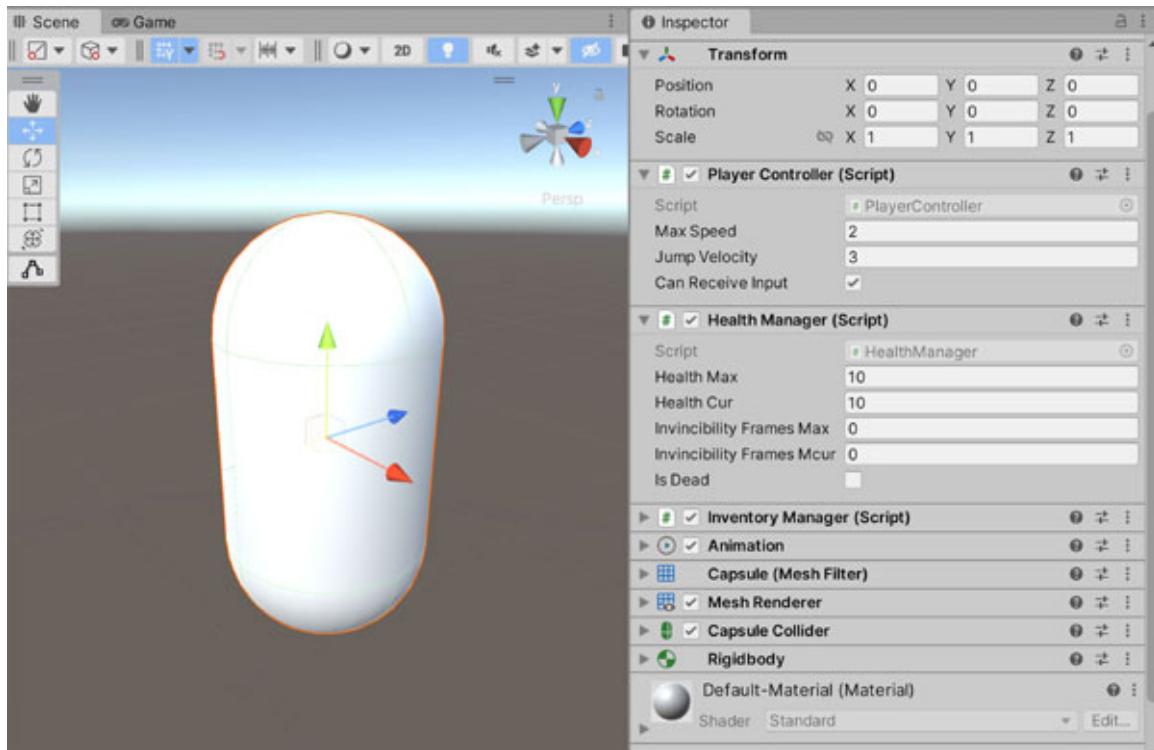


Figure 2.1: The Player Controller, Health Manager, and Inventory Manager scripts are all components that we'll be creating in later chapters.

An enemy object, on the other hand, would have no need for components related to user input. Instead, they would have an AI Brain driving their movement logic. Other than that differentiation, their components would be similar to what we found on the Player (health, physics, and so on).

The inspector for enemy objects would resemble the one shown in [Figure 2.2](#).

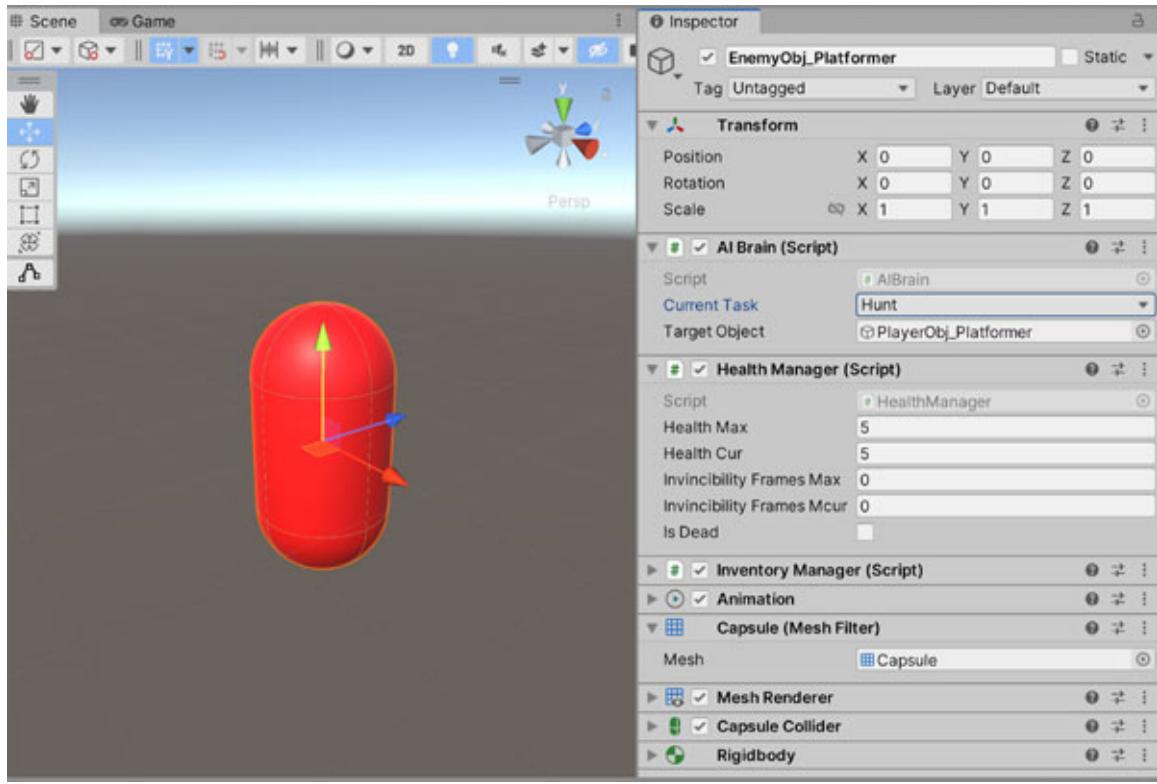


Figure 2.2: Note that, for the Enemy object, the PlayerController has been replaced with an AIBrain component.

And if we want projectile weapons in this third person platformer, we're going to need a basic Bullet object. this would be considerably different then the Player and Enemy objects, most likely having its own unique bullet component, with no concept of health, inventory, or controller input. A simple bullet object would look like this in the inspector window...

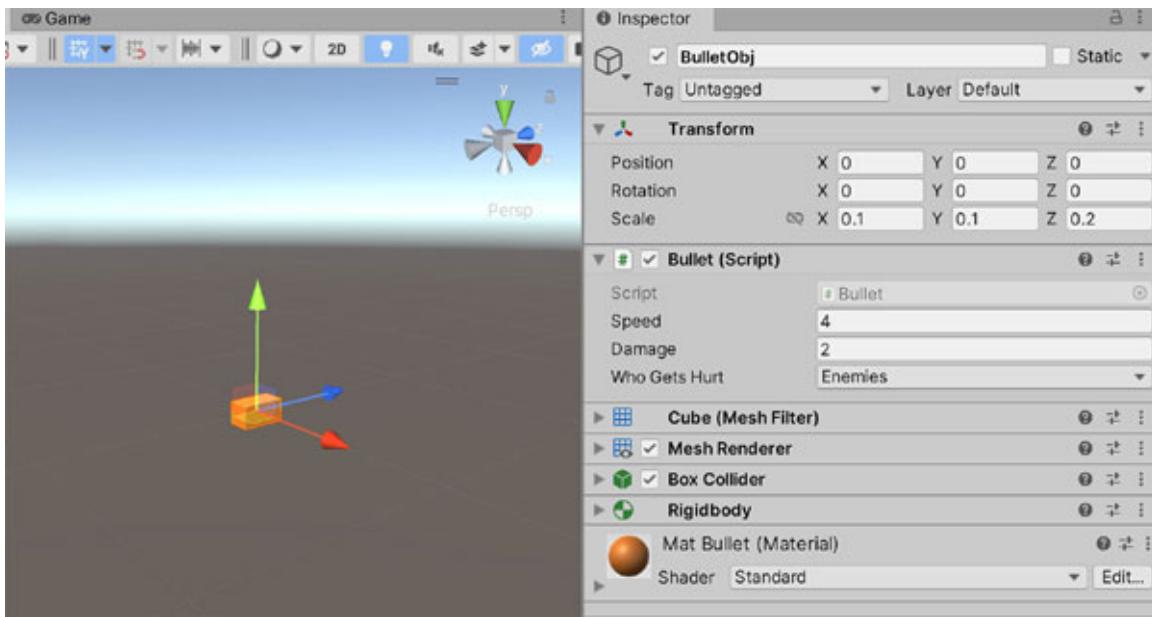


Figure 2.3: The Bullet is the simplest example: only having one unique script - “Bullet” - attached to it.

For now these are all examples. But they show you how any game object can be broken down into its core components. A Terrain object would consist of its Mesh Rendering components, some Physics data, and a unique Terrain script. NPC characters would be similar to the enemies, only instead of an AI Brain that hunts the player, it would have a component that seeks out to help the player.

And luckily, Unity makes the creation and attachment of components a simple one-step process.

Adding a Component to an Object

To add a component to an object, you’ll first need to have that object selected. Right click in the scene view or the hierarchy list to pick the object you want to add a component to. Since we’ve already added a new component to the Ground object in [Chapter 1](#), let’s select our Sphere this time.

We’re going to make the Sphere our ‘Player’ object, complete with movement and jumping capabilities.

1. Once the Sphere is selected, the inspector window will show a list of all the components assigned to it. These components should look familiar, but this time we’ll be right clicking towards the bottom and selecting the **Add Component** option.

2. When you first select Add Component, you'll be presented with a long list of pre-existing scripts provided by Unity. We'll dive into these in later chapters, but right now we want to create our own player input component on the Sphere. Select **New Script**, and enter **PlayerController** as its name.

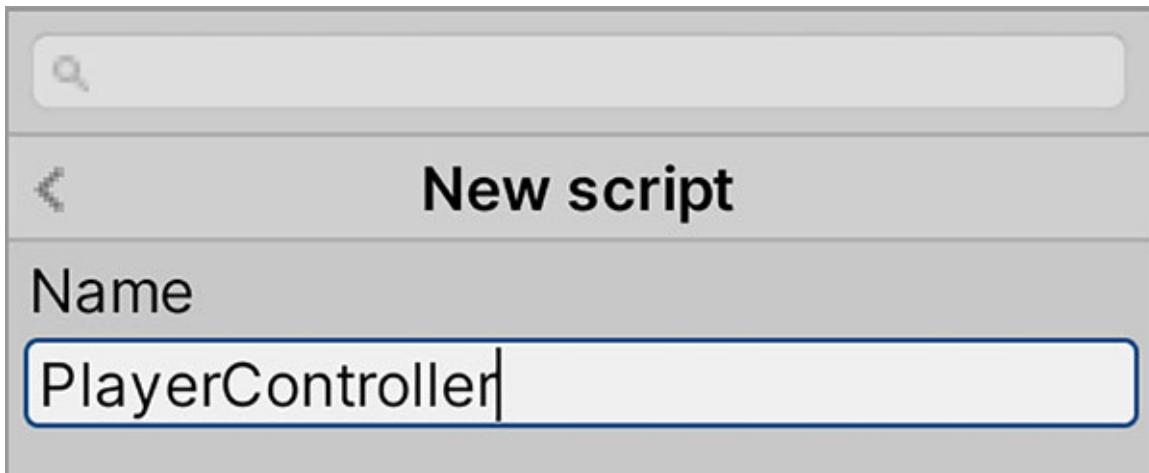


Figure 2.4: You can create a new script directly from the Add Component panel.

3. Press **Enter**, and the component will be created and added to the Sphere. You can double-click the script name to bring up the code now.

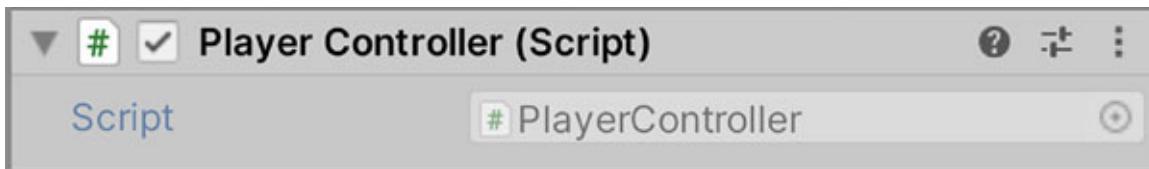


Figure 2.5: Edit the new script by either double-clicking the script name, or by Right-clicking this panel and selecting “Edit Script”.

The MonoBehaviour class

By default, any component made and attached to a gameobject will inherit from the MonoBehaviour class. This base class has several useful functions that will automatically be called by the Unity engine, giving you a great starting point when writing the code that will define your game systems.

Class Inheritance: Because C# is an Object-Oriented language, it relies heavily on the concept of Inheritance. Inheritance, in code, is the practice of building a new class, script, or component off of an existing one. By Inheriting from the base class, your new code will have all the

functionality of the base without you needing to re-write that functionality. When used correctly, it allows for the writing of fast, reliable code.

While the script will start off fairly empty, you'll see that several functions have already been defined and are ready for you to utilize:

- **Awake()**: This function gets called the moment an object is created in the Scene. It's a good place to initialize certain values, but be warned that some components and objects may not be available at this time. Any initialization code that relies on other components should go in the Start function.
- **Start()**: This function is called on the first frame that an object is active in the scene. Useful for initializing things that need other objects and components to be loaded first.
- **Update()**: This function gets called every frame, or at least as many times as possible based on the current frame rate. This is where the bulk of your components logic will probably be stored. It's also very easy for this function to get bloated, so as we write our script, we'll be doing our best to break our update calls into sub-functions. Organization is key when writing readable code that's easy to maintain and build upon.

Writing our ‘Player Controller’ Component

With our new PlayerController script created, and a better understanding of the MonoBehaviour class, it's time to write some code to make our Sphere move around.

1. We'll start with member variables, which get added after the pre-made class definition...

```
public class PlayerController : MonoBehaviour
{
    // the Rigid Body physics component of this object
    // since we'll be accessing it a lot, we'll store it as a
    // member
    private Rigidbody _rigidBody;
    // acceleration applied when directional input is received
    private float _movementAcceleration = 2;
    // the maximum velocity of this object
    private float _movementVelocityMax = 2;
```

The first member we defined, `_rigidBody`, allows us to grab the `Rigidbody` component once and use it throughout our script.

The other two members are directly related to movement speed. Our `_movementAcceleration` defines how much our velocity will change when the arrow keys are pressed. The `_movementVelocityMax` member will limit the speed that our object can go. Without this, our sphere would quickly zip off the screen. A max limit simply keeps our objects from moving too fast.

2. Our next bit of code is the `start()` function, where we'll place some initialization code.

```
void Start()
{
    _rigidBody = GetComponent<Rigidbody>();
}
```

For now, we're only initializing our `_rigidBody` variable. Since we're writing a component that has a lot of velocity adjustment, we'll be making heavy use of this component. It's a good idea to store components that are used heavily, since this cuts down on the times you're performing the `GetComponent()` lookup function.

GetComponent(): Every game object is made out of components, so at some point you'll want to access those components in your code. To do this, simply use the `GetComponent()` function, which allows you to search for, grab, and utilize any component attached to the current object. If no component of a given type is found, this will return null.

3. Onto the `Update` function, which gets called every frame. This is where we'll intercept those key presses and adjust out `Sphere`'s speed accordingly.

```
void Update()
{
    // get the current speed from the Rigidbody physics
    // component
    // grabbing this ensures we retain the gravity speed
    Vector3 curSpeed = _rigidBody.velocity;
    // check to see if any of the keyboard arrows are being
    // pressed
    // if so, adjust the speed of the player
    if (Input.GetKey(KeyCode.RightArrow))
```

```

    curSpeed.x += (_movementAcceleration * Time.deltaTime);

    if (Input.GetKey(KeyCode.LeftArrow))
        curSpeed.x -= (_movementAcceleration * Time.deltaTime);

    if (Input.GetKey(KeyCode.UpArrow))
        curSpeed.z += (_movementAcceleration * Time.deltaTime);

    if (Input.GetKey(KeyCode.DownArrow))
        curSpeed.z -= (_movementAcceleration * Time.deltaTime);

```

There are two important tasks being performed here. First, we're making sure to grab the current velocity from the RigidBody component. This ensures we're making changes to the most up-to-date movement data for this object, taking into account gravitation and external forces that may be acting upon the object.

4. Next, there is a series of `if(Input.GetKey())` checks where, if a key is being pressed, the `curSpeed` variable is adjusted. The change in velocity takes two values into account: the acceleration value to apply (`_movementAcceleration`), and the change in time (`Time.deltaTime`). By basing the change off of Delta Time, we can ensure the movement will feel the same no matter the framerate.

`Time.deltaTime`: *The MonoBehaviour Update() function has access to an important variable called `Time.deltaTime`. This tells you seconds that have passed since the last time you were in this function. Most of the time, this value is in the millisecond range (~0.01f), but if framerate issues occur, this number could get much larger. Be sure all your movement calculations take this into account, otherwise your math will yield different results based on the speed of the device.*

5. We'll finalize the `Update()` function by limiting the velocity and reapplying it to our `RigidBody` object.

```

// apply the max speed
curSpeed.x = Mathf.Clamp(curSpeed.x, _movementVelocityMax *
-1, _movementVelocityMax);
curSpeed.z = Mathf.Clamp(curSpeed.z, _movementVelocityMax *
-1, _movementVelocityMax);
// apply the changed velocity to this object's physics
component
_rigidbody.velocity = curSpeed;

```

6. If you don't see any errors, it's time to test. Return to the Unity editor, press the Play button at the top of the screen, and start pressing the arrow keys.

You should immediately notice something unexpected: the keypad input is still adjusting the rotation of the floor.

7. Fortunately, the fix for this is an easy one. Press the Play button again to stop the game, select your Ground Object, then deselect the Check Mark in the upper left corner of the panel.

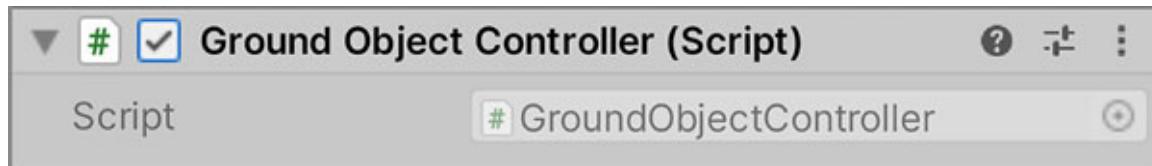


Figure 2.6: Disable components by toggling the Check Mark in the upper-left corner.

While we have the Ground selected, you'll also want to set its Z rotation to 0. If you don't, the ball will keep rolling down the tilted surface, possibly causing frustration as we test our new scripts.

8. Selecting **Play** again gives us a better idea of what our PlayerController script is doing. Pressing the arrow keys certainly does what we intended: the sphere moves around the scene, and the ground no longer rotates.

However, the movement is *very* sluggish. Because of the slow acceleration / deceleration, our Player object feels heavy and tired - not nimble and reactive, like a good hero should.

We need to tweak our components as we play our game, and Unity gives us some useful ways to do that.

Extending your Components in the Unity Editor

Another powerful aspect of Components is the ease at which they can extend the Unity Editor. In this case, we would like to edit our acceleration and max velocity variables outside the script.

There are two ways to write your code to allow this...

- **Public Members:** By using the keyword **public** instead of **private**, you can define script variables that will show up in the Inspector when this object is selected.

- **[SerializeField]**: You can apply the **[SerializeField]** attribute to your variables by placing this tag above the member definition. This allows you to retain the **private** setting, but still allow the member to be edited in the Inspector.

Private vs Public: Class members can have different levels of accessibility, as defined by the **private**, **protected**, and **public** keywords. A **private** member can only be accessed within the class it was defined. A **protected** member is similar to **private**, but can also be accessed by inherited classes. A **public** member can be accessed by anybody. If no keyword is used in the definition, Unity will assume it is a **private** member.

We're going to use the second method - **[SerializeField]**. Besides allowing these members to remain private, attributes let you do some other useful things in the editor.

Replace these two variable definitions with the following lines...

```
[SerializeField, Tooltip("How much acceleration is applied to
this object when directional input is received.")]
private float _movementAcceleration = 2;
[Tooltip("The maximum velocity of this object -
keeps the player from moving too fast.")]
private float _movementVelocityMax = 2;
```

You'll note that, besides the **SerializeField** attribute, we also added a **Tooltip**, taking our comments about these variables and turning them into text that gets displayed right in the Unity Editor. When you're in the Inspector window, you can hover over the fields to get a useful description.



Figure 2.7: Our PlayerController component with exposed members and tooltip attributes.

Now, when we play the game, we can adjust these values for real-time testing. Press **Play**, move the sphere around, then speed it up by adjusting Movement

Acceleration to 50 and Movement Velocity Max to 10.

When you press the arrow keys now, you'll notice the sphere is MUCH more responsive. You may also notice it's much easier to send the Sphere zipping off the side of the screen.

To fix this, we'll want to add some additional Friction to our PlayerController object. Bring the script up in Visual Studio, and under the Movement Velocity definition, add this line...

```
[SerializeField, Tooltip("Deceleration when no direction input is received.")]  
private float _movementFriction = 0.1f;
```

Then, in the **Update()** function, add these lines after the four **if()** checks for keyboard input...

```
// if both left and right keys are simultaneously pressed (or not  
// pressed), apply friction  
if (Input.GetKey(KeyCode.LeftArrow) ==  
Input.GetKey(KeyCode.RightArrow))  
{  
    curSpeed.x -= (_movementFriction * curSpeed.x);  
}  
  
// apply similar friction logic to up and down keys  
if (Input.GetKey(KeyCode.UpArrow) ==  
Input.GetKey(KeyCode.DownArrow))  
{  
    curSpeed.z -= (_movementFriction * curSpeed.z);  
}
```

Handling Double Key-Presses: Keyboards are an input device that accept the pressing of multiple keys at the same time. This means you have to consider how the Player object should move when both Left and Right (or Up and Down) keys are held simultaneously. The checks above may look strange - applying friction if both keys are the same value - but since we would also want the object to stop if both keys are held, it's a simple way to manage simultaneous keypresses.

When making user-controlled characters, reaction time is our top concern. The player will be expecting the hero object to move *instantly* in the direction they press. Similarly, if they're not pressing any direction key, they will expect the Player object to stop quickly as well. This is why a fast acceleration and fast

deceleration are important - without them, controls will feel ‘floaty’ and ‘sluggish’.

Because we’ve stopped the game, you’ll also notice the Acceleration and Max Velocity values have reverted to their original values. Since we were happy with the speed of the Sphere, let’s reapply those changes (Movement Acceleration to 50, Movement Velocity Max to 10).

Finalizing Changes: While it’s easy to adjust values mid-game, it’s critical to catalog what changes you want to actually apply. Changes made when the game is running are **temporary**, so once the game is stopped, you need to go back into the inspector and re-apply any changes worth keeping.

A Hop, Skip, and a Jump

We have our Sphere object moving nicely, but it wouldn’t be a proper third-person PlayerController without allowing the player to **Jump**.

There are two new members we’ll want to add to make our jumping feel good. Add these below the other members of our PlayerController script.

```
[SerializeField, Tooltip("Upwards force applied when Jump key is pressed.")]  
private float _jumpVelocity = 20;  
  
[SerializeField, Tooltip("Additional gravitational pull.")]  
private float _extraGravity = 40;
```

Then, in the Update() function, place this input check after we check for the directional keys.

```
// does the player want to jump?  
if ( Input.GetKeyDown(KeyCode.Space) && Mathf.Abs( curSpeed.y ) < 1 )  
    curSpeed.y += _jumpVelocity;  
else  
    curSpeed.y -= _extraGravity * Time.deltaTime;
```

The jump logic here states that, if the player has pressed the SpaceBar, and no longer has a significant downwards velocity, apply the upwards ‘Jump’ velocity in the Y direction.

If this doesn’t happen, the game will apply an additional gravitational force to the object, to ensure they fall at a satisfying speed.

Press play, and test jumping by pressing the SpaceBar on your keyboard. Feel free, at this point, to play with all the variables in your PlayerController script to get movement feeling how you'd like it to.

***Not Just a Number:** The values that we're using for our PlayerController script have been tweaked and adjusted as we playtested our demo. It's important to spend that time playing and tweaking, because these numbers have the power to ruin a game. For instance, while the game is running, bring up the PlayerController panel and enter these values for movement and jumping: 4, 4, 0, 10, 0. What once was a peppy, reactive hero has become heavy and sloppy. Always remember, and respect, the power of numbers when making a game.*

With Movement and Jumping hooked up for our player, it's time to move onto spawning collectable items with a new Unity system: **Prefabs**.

Making and Spawning Prefabs

Up until this point, our objects have all been placed using the Unity Editor. This is fine for early prototypes, but most games do the majority of their object spawning at runtime. This means that your code will need to have a way to spawn GameObjects within your scripts.

It may sound difficult, but Unity makes this easy with Prefabs.

Prefabs are GameObjects that have been created and bundled for later use. You will put together a GameObject with the assets and components that will represent your prefab. Then, by dragging this object into your Project window, Unity knows that this is an object that you want to spawn at runtime, turning it into a prefab.

In code, you will then be able to use the **Instantiate()** function to programmatically spawn copies of your object.

Let's dig into this process with the most common item you'll find in games: the Coin!

Coin of the Realm

There's nothing like a gold, spinning coin to pique the player's attention and get them moving in a direction. These may be used at an in-game shop, guide the player through a level, or simply collected to rack up a big score. However they're used, gold coins are a staple of video game design.

To make our gold coin, we'll start with a simple sphere. Like with our other objects, right click in the hierarchy window and select **3D Objects > Sphere**. Use the Transform panel to place it at the origin of the scene (0,0,0) and apply these scale values to it (0.5, 0.5, 0.1). By scaling it more on its Z-Axis, we can squish it down to get the approximate shape of a Coin.

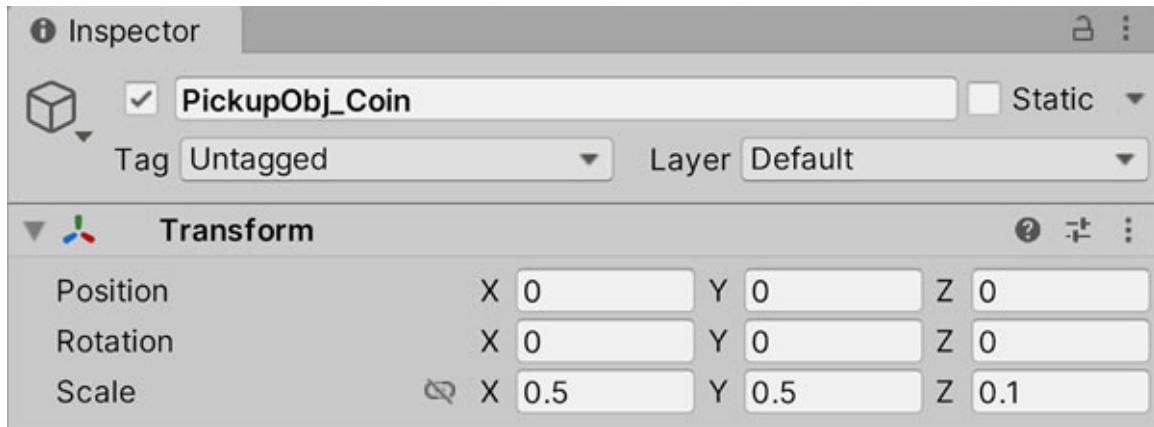


Figure 2.8: Set the scale of the new sphere to 0.5, 0.5, 0.1 - this squishes it to a 'Coin' shape.

Even though we're prototyping our game with simple shapes, we can make our coins look more interesting by applying a new **Material**. We'll get into the guts of the material system later, but for now, simply right-click in your Projects panel and select **Create > Material**. Change the 'Albedo' to a Yellow color, 'Metallic' to 1, then drag the new material from the Projects panel onto your Coin object. The coin should now look nice and shiny - letting the player know that this is something worth grabbing!

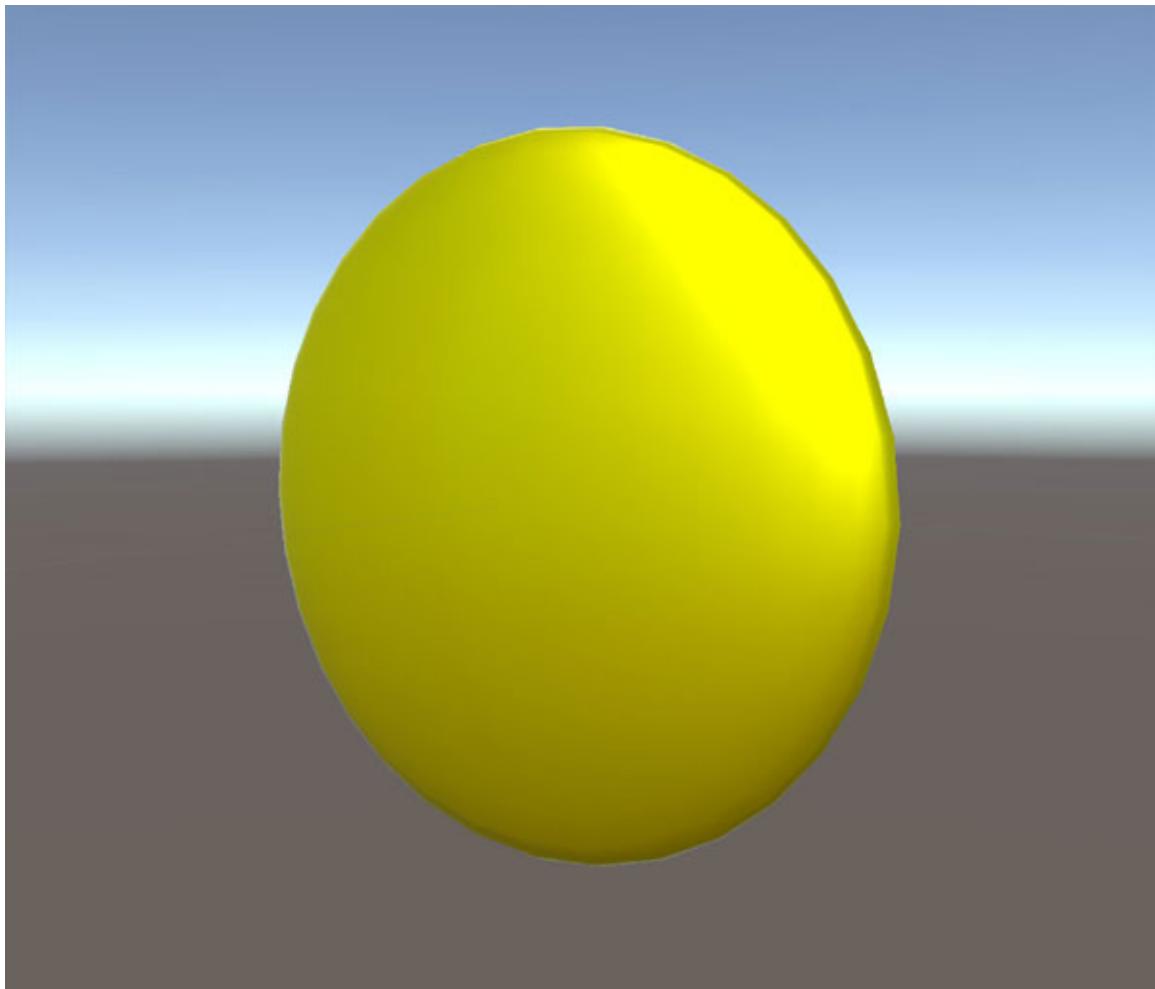


Figure 2.9: Our simple Coin object with a new Yellow Material applied to it.

When adding and creating components, it's important to make your systems as flexible as possible. This means that, instead of making a different component for every possible pick up item (Coins, Health, Ammo, Etc) we'll make a single 'pick up' component that can handle all these different gameplay mechanics.

Let's select the coin object, scroll to the bottom of the inspector, and click **New Component** > **Add Script**. Again, since the coin is only one type of pick up item we'll be adding, let's create a component called **PickUpItem**.

Since one of the main uses of Coins is to grab the players attention, and since Animation is one of the best ways to do that, it's important that we hook up some automatic rotation to our spinning coin.

Let's add a rotation speed member to our PickUpItem script...

```
[SerializeField, Tooltip("The speed that this object rotates at.")]
private float _rotationSpeed = 5;
static int s_objectsCollected = 0;
```

While we're here, we also defined a variable to store the number of objects collected by the player (`s_objectsCollected`). We'll make use of this counter in a bit.

As for our spinning animation, let's add some code to the **Update()** function that uses **Time.deltaTime** to apply a rotation change to the coin. The body of **Update()** should look like this...

```
// grab the current rotation, increment it, and re-apply it
Vector3 newRotation = transform.eulerAngles;
newRotation.y += (_rotationSpeed * Time.deltaTime);
transform.eulerAngles = newRotation;
```

With the code hooked up, let's return to the Unity Editor and press the play button. If we did our job right, the coin object should be spinning, letting the player know it's something special they can interact with.

Now let's dive into the most important part of a pickup item: detecting collisions.

Collision Detection

In Unity, the handling of collisions between two objects happens automatically in the Physics Engine. In fact, the Physics engine Is designed to handle the Collision of thousands of different objects at a time.

Luckily, right now we simply need to detect when the player hits one of our pick up items: in this case, the Spinning Coin we just made. And since both the player object and the coin object have RigidBody physics components, all we have to do is write the code to intercept the Collision message.

Game objects have several Collision messages that we can make use of...

- **OnCollisionEnter()**: This function is called when one rigid body object collides with another. A set of Collider data is passed in containing specifics about the collision, like what object was collided with and what the point of contact was.
- **OnCollisionExit()**: A function automatically called then one rigid body object stops touching another object.

- **OnCollisionStay()**: Called once per-frame when one rigid body object continues to collide with touching another physics object.

These are all functions that work with your default rigid body objects. However, sometimes, you want to make physics objects that can't be pushed around physically.

In these cases, you want to use a slightly different approach: Physics Triggers.

Triggers

All Collider components have an option called “Is Trigger”. By default this is disabled, meaning that the object has a physical presence in the 3D World (i.e. get pushed around, prevent movement, etc). However, there are times where we want to respond to collisions but we don't necessarily want those objects being moved around the scene.

For these objects, such as environmental props and pick up items, we'll set them as Triggers. Do this by enabling the ‘Is Trigger’ option on our coin's Collider component.

Figure 2.10: Trigger colliders will not apply physical force to other objects.

Our coin will no longer interact physically with other objects (ie. our player can't stand on them) but they will receive Collision information when other objects collide with it.

When using triggers, you'll need to make use of slightly different collision functions...

- **OnTriggerEnter()**: This is called when one or more trigger objects collide with each other. Will also be called if the trigger object collides with a physically-based (ie. non-Trigger) RigidBody object.
- **OnTriggerExit()**: Called when one or more trigger objects stop colliding.
- **OnTriggerStay()**: This is called every frame that a RigidBody object is colliding with a trigger object.

Now that we understand the basics of collision detection, we can set up a function that fires when the player collides with a coin.

To do this, we'll want to open up the **PlayerController** script in Visual Studio.

Go past the **Update()** code and add this function below it...

```
void OnTriggerEnter(Collider collider)
{
    // did we collide with a PickupItem?
    if (collider.gameObject.GetComponent<PickUpItem>())
    {
        // we collided with a valid PickupItem
        // so let that item know it's been 'Picked Up' by this
        // gameobject
        PickUpItem item = collider.gameObject.GetComponent<PickUpItem>
        ();
        item.onPickedUp(this.gameObject);
    }
}
```

There are a few important things going on here. First, we use the **GetComponent()** function to see if we collided with a pickup item. Since pickup items make use of the `PickUpItem` script, this is an easy way to determine what type of object we've collided with. If we *have* collided with a pickup item, we then let that script know that we're trying to pick it up.

For that to work, however, we need to actually add the function **onPickedUp()**. Open the `PickUpItem` script in Visual Studio, and add the following lines below our `Update` code block...

```
public void onPickedUp( GameObject whoPickedUp )
{
    // show the collection count in the console window
    s_objectsCollected++;
    Debug.Log( s_objectsCollected + " items picked up." );

    // destroy the item
    Destroy(gameObject);
}
```

This gives us a central location to deal with the collection of pick up items. For now, all it's doing is incrementing a global counter, writing that value to the debug console, and destroying the pickup item itself.

While all this could have been done in the player controller script, it's important to keep all of our pick-up item logic within the `PickUpItem` component. By having one place to manage this logic you make debugging and code maintenance much easier.

With your new functions written, press the play button and put your hard work to the test. If you did everything correctly, you'll be moving the player object around and collecting the single coin that exists in the scene.

Unfortunately, collecting one single coin is pretty boring. Let's put the prefab system to the test and give our player lots of coins to collect.

Turning an Object into a Prefab

With our coin object made, it's time to turn it into a prefab. Let's start with a little bit of housekeeping. In your Project window, let's add a **Resources** folder, and within that, add a **Prefabs** folder.

The Resources Folder: *One of many ways to load assets from code is by using a special Resources folder. When placed in the Assets directory, any objects stored in the Resources folder will be accessible from code. There are other ways to load assets dynamically, but if you're looking through existing projects and see a 'Resources' folder, know that it's there for a specific reason: loading objects at runtime. In future chapters we'll discuss other methods of run-time asset management.*

Once the folders are made, simply select the Coin object from the hierarchy panel and drag it into the empty Prefab folder.

Figure 2.11: The new Prefabs folder starts empty, as indicated by its empty icon.

Unity knows that when you drag an object into the Projects window, you intend to make it into a prefab. Open that prefabs folder and you will find one object in it: the coin.

Figure 2.12: Our newly minted 'Spinning Coin' prefab.

Spawning More Coins

Think of all the games that you've played where you're collecting coins. It's a task that's fun for the player, but hand-placing all those coins sounds - well - less fun. To make our lives easier, we're going to wrap up this chapter with an item spawning system!

Since we're making a system, and not an object, let's start with an Empty GameObject. Add this to the scene by right-clicking the hierarchy window and selecting **Create Empty**. Name it ItemSpawnZone and set its position to 0, -1, 0 in the Transform panel.

Next, attach a BoxCollider component to it. This will be used as the volume in which we'll be randomly spawning out items. This allows us to quickly designate an area and fill it with coins, opposed to having to hand-place every coin object in the scene.

Set the **Center** of the BoxCollider to (0, 0, 0), set its **Size** to (10, 0.1, 10), set it as a Trigger collider (otherwise our Player would be able to jump on top of it).

Figure 2.13: Our ItemSpawnZone object with BoxCollider component.

Now, add a new script to this object called ItemSpawnZone. Note that the object name and the component name do not need to match for this to work, but it does make it easier to track down objects related to specific components.

Open this new script in Visual Studio and create a component that looks as follows:

```
public class ItemSpawnZone : MonoBehaviour
{
    [SerializeField, Tooltip("Prefab to spawn in this zone.")]
    private GameObject _itemToSpawn;

    [SerializeField, Tooltip("Number of items to spawn.")]
    private float _ItemCount = 30;

    [SerializeField, Tooltip("The area to spawn these items.")]
    private BoxCollider _spawnZone;
    void Start()
    {
        // spawn the items within this area
        for ( int i = 0; i < _ItemCount; i++ )
        {
            SpawnItemAtRandomPosition();
        }
    }

    void SpawnItemAtRandomPosition()
    {
```

```

Vector3 randomPos;
    // randomize location based on the size of the associated
    BoxCollider
randomPos.x = Random.Range( _spawnZone.bounds.min.x,
_spawnZone.bounds.max.x);
randomPos.y = Random.Range( _spawnZone.bounds.min.y,
_spawnZone.bounds.max.y);
randomPos.z = Random.Range( _spawnZone.bounds.min.z,
_spawnZone.bounds.max.z);
    // spawn the item prefab at this position
Instantiate(_itemToSpawn, randomPos, Quaternion.identity);
}
}

```

There are a few concepts in this code that you may not have seen before.

- **'For' Loops:** In the **Start()** function, you'll note that we have a bit of code that loops through the item count. This will call the **SpawnItemAtRandomPosition()** once per item to spawn.
- **Random.Range():** This is Unity's go-to function for generating a random number. By passing in the max and min X, Y, and Z values of the collider, we can get a random point within the box collider volume.
- **Instantiate():** This is the function that does all the heavy lifting for spawning prefabs. By passing in an existing GameObject, a new Position and new Rotation (in Quaternions), Unity will create a duplicate of your prefab object.

Return to Unity, and the ItemSpawnZone component should look like this in the Inspector Window...

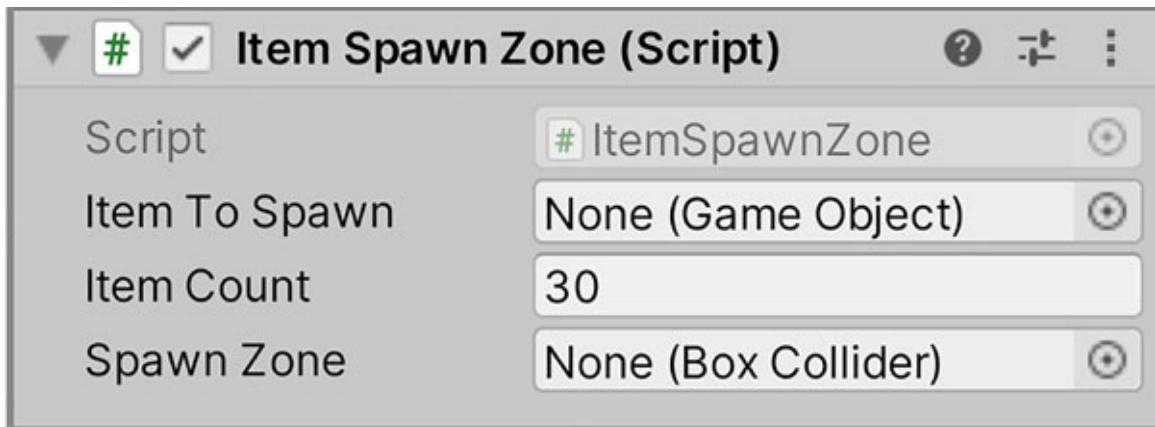


Figure 2.14: The *ItemSpawnZone* script, when first created.

As you can see, there are two empty members that will have to be assigned. To set the **Item To Spawn** value, find the Coin prefab in the Project's Assets>Prefabs folder, and drag that into where it says "None".

For the **Spawn Zone**, just drag the Box Collider component (which is above the Item Spawn Zone script in the Inspector Window) down into the empty Spawn Zone field (the spot that says "None" (Box Collider)).

Once both of these values are assigned, the panel will look like this...

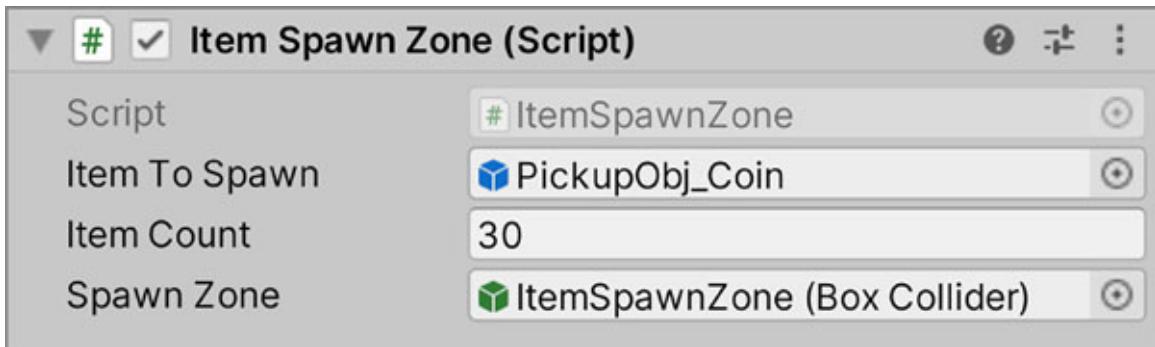


Figure 2.15: With the Item and Zone data set, we're ready to test!

Our final step is to put it all to the test! Press the play button and, if we did everything correctly, there should now be 30 coins sprinkled along the floor for the player to collect, each one rotating and disappearing when the player collides with them.

Extra benefits of Prefabs

While prefabs are a great way to fill your game with lots of items, bullets, and enemies, there are other benefits to them Beyond simply spawning 'lots of stuff'.

- Prefabs allow you to make changes in one spot, then have those changes reflected across the project.
- Prefabs allow you to organize and simplify your Project assets.
- When collaborating, prefabs allow different Developers to make tweaks without stepping on each other's toes.

Debug Messages

Now that we're starting to get into more advanced coding practices, it's a good time to talk more about debugging. Specifically, logging debug messages so you can tell what's going on in your game code while you're playing.

One of the most useful functions you'll find is the **Debug.Log()** function.

In fact, we've already made use of this in the PickUpItem script. Every time a coin is collected, we print out a message using **Debug.Log()**. Let's bring up the window that displays these messages: the Debug Console.

At the bottom of the Unity Editor there's a window with two tabs - "Project" and "Console". Select the **Console** tab and press the play button. As you collect coins, you will see a message appear in the Console Window telling you the total coins you've collected.

Figure 2.16: The debug console shows you Messages, Warnings, and Errors related to your game.

By default, this list will automatically clear every time you start a game, so if you have important data to collect, make sure you do it before restarting your game.

There are also three buttons in the upper right corner. These are visibility toggles to show and hide Messages, Warnings, and Errors.

- **Messages:** These are status updates from your code, letting you know what's happening in your components as their scripts run. Functions such as `Display.Log()` will write messages to the console window.
- **Warnings:** These are more important issues that arise, normally cautioning you that bad things are happening in your scripts.
- **Errors:** These are areas of broken code or crashes that occur in your scripts. Your game will not run if you write a script with errors in it.

As our components become more complex, we will use debug logging, as well as the Debug Console, to ensure our systems are working as expected.

Game Design 101: Iteration and White-Boxing

Now that we have a small playable demo, it's time to talk about the most important concept of game design: **Iteration**.

Iteration is the act of taking something that you've done, testing it, playing it, and improving upon it. It's developing a game using small steps.

While it may be tempting to front load your game development with thorough design documentation, the real trick to making a great game is a rapid cycle of designing, implementation, playtest, improvement.

- **Design:** On a sheet of paper, or an online document, write out the intention of what you need to hook up. Keep it simple - no more than a page.
- **Implement:** Hook up the system as designed. Keep it lightweight - the purpose of this step isn't to flesh out every possible use of a new system. It's to get something playable quick.
- **Playtest:** This is where your best game design ideas will be born. As you playtest your implemented design, you'll get your clearest vision of your game. As you play, ideas will start to fill your head. Work off these.
- **Improve:** Take the ideas collected during the Playtest phase and prioritize them. Implement the best ones and playtest again.

Repeat the process until someone else can play the game and doesn't want to give you the controller back. If others can't stop playing, then you're onto something good.

Our current ‘game’ would be considered the start of a **White-Box Prototype**. White-boxing your game needs to build it out using the most basic graphical objects available: in this case, Spears, cubes, and other basic 3D shapes.

The thought behind White-Boxing your prototype is this: if you can make a game fun with simple graphics and no sound, it will only be made better when your finalized assets are hooked up.

There's an inverse concept to this that's highly tempting: the path of “put awesome assets in and make the game around those”. Don't be seduced by it!

Always implement your ideas as quickly as possible so you can playtest them as quickly as possible. This is where the idea of white boxing shines: you can quickly hook up your design with simple cubes, spheres, and cylinders. If it's fun to play in its simplest form, it will only become better as finalized graphics, visual effects, and sound are added.

The Unity engine makes this rapid iteration cycle a breeze. Remember the importance of iteration - starting simple and building towards greatness - as we go through the rest of the book.

Conclusion

Since games require the management of *so many* assets, Unity was built with ease-of-use in mind. The Component and Prefab systems are the perfect example of this ideal – fast, flexible systems that allow you to quickly create objects and reuse systems. Component scripts can also be written in a way that exposes tweakable values as Parameters to the designer, allowing even more flexibility in a component. Once created, converting an object to a prefab makes it even **MORE** flexible, allowing it to be spawned in real-time.

And with our Player object started, and a firm understanding of Components and Prefabs, it's time to implement our first major gameplay systems: Health and Damage.

Questions

1. True or False: Most gameplay functionality is added using Components.
2. True or False: More powerful devices will call the Update() function more times than a slower device.
3. True or False: Your script must inherit from the MonoBehaviour class before it will function as a Component.
4. True or False: If you disable a Component on an object, its Update() function will still be called.
5. Which MonoBehaviour function gets called first: Awake() or Start()?
6. If you had two numbers, and you wanted to calculate a random value between those two numbers, what Unity function would you use?
7. Iteration is the process of slowly making your game – building it, testing it, and improving it based on feedback. This goes against the inclination to work for months (or years) on a game before having people playtest it. Why do you think it's better to get feedback early, and make tweaks early, rather than lock your project away until its 'perfect' and ready for outside feedback?

Key Terms

- **Components:** Modular systems (often in the form of C# files) that can be attached to any GameObject. Components are used to define the gameplay purpose of a given object and can be tweaked by the designer to allow for maximum flexibility (ex. Players and Enemies could use the same Health Component).

- **Prefabs:** An object, made out of various components and sub-objects, that has assembled and stored in your project to be used later.
- **Scripts:** A C# file that has some code and gameplay logic stored in it. If a script inherits from the MonoBehaviour class, it can be attached to an objects as a Component.
- **C#:** The coding language used in Unity scripts and components. A managed memory scripting language that blends the syntax of C++ and java.
- **IntelliSense:** A tool that helps you see what functions and members are available as you write your code.
- **Parameters:** Values in a script that can be changed in the Unity Editor.
- **MonoBehaviour:** The base class of any GameObject script. If a script inherits from the MonoBehaviour class, Unity will allow that script to be attached to any object.
- **Time.deltaTime:** A value that stores the number of seconds between this frame and the last frame. Useful when calculating a change in position between calls to the Update() function.
- **Input.GetKeyDown():** A way to determine if a key was pressed on the keyboard.
- **Instantiation:** Spawning prefabs into a game at runtime using the Instantiate().
- **Random.Range():** A function for finding a random value between two numbers. Note that this is an *exclusive* calculation, so the max value will *not* be included in the possible results.
- **Collision Detection:** When collider objects come into contact, it will cause specific collision functions to be called. Add these functions to scripts attached to gameObject to designate how they should react once a collision occurs.
- **Debug.Log():** A function that, when hit in code, will write out to the Console Window. Use it to see what the code is doing, and make sure specific functions are being called.
- **Iteration:** The cycle of game creation where you implement features, get feedback, and tweak/improve your game based on that feedback. the more you can iterate, the better your game will be.

- **White-Boxing:** Creating a game without fancy graphics, forcing the designer to focus on the interactivity and fun of the experience in its simplest form. If you can make a fun game with just colored shapes, it's going to only get better when graphics are finalized.

CHAPTER 3

The Basics of Combat

In the last two chapters, we focused on learning the basics: using Unity's editor, making GameObjects, writing components, and spawning prefabs. It's time to utilize all these concepts as we start our first major gameplay system - **Combat**.

Structure

- Risk and Reward
- Designing Subsystems
- Health Management
- Applying Damage
- Building Level Hazards
- Spawning Projectiles
- Improving Player Feedback
- Game Session Managers

Objective

After reading this chapter, you'll have a player that can take damage, bullets and hazards that deal damage, a handful of effects to show damage, and a system that respawns your player on death.

You'll also learn several game design concepts such as the importance of Storyboarding, Visual Feedback, interesting choices, the lure of interesting Risk and Reward situations.

Game Design 101: Risk and Reward

Before we dive into the systems (and subsystems) required for battle, let's first discuss the reasoning behind implementing a combat system *at all*.

Specifically, we need to talk about the core game-design theory of **Risk and Reward**.

In game design, almost every scenario can be simplified down into a core choice that the user must make. In chess, it's what piece to move and where to move it. In a side scrolling video game, it's timing when to make that jump. In an FPS it's choosing your loadout. In a business simulator, it's designing your office so everyone can access the coffee maker.

All these choices require the player to balance the **Risks** and **Rewards** of a situation. And all these smaller choices cumulate into the ultimate Risk/Reward duality of Victory or Defeat.

But what does this have to do with Combat, you ask?

When designing a game, much of the player's enjoyment comes from weighing these Risk/Reward situations and acting on them intelligently. When it comes to the health of the player, this is a Risk/Reward concept they can understand in a *primordial* way. The player - assuming they're a human - *has survival baked into their DNA*. The reward of "not dying" doesn't have to be explained in a tutorial. It's something we all understand, so it's a gameplay concept worth tapping into.

Combat systems take this guttural, primordial need to 'stay alive' and makes it a key part of your game. When your hearts get too low, health bar depletes, or HP gets in the single digits - it raises the stakes for the player.

In a word, Combat systems make games *exciting* - so let's start making our Combat systems.

Subsystems

Whenever you have a large, important gameplay system like Combat, it's bound to require the implementation of various *subsystems* that will work together in harmony.

Our initial implementation of a combat system will be built on the following subsystems:

- **Health manager:** This will be a new component that we can attach to objects that can take damage and be destroyed.
- **Health modifiers:** We will be making components that can modify the data contained within the health manager. Hazardous objects will

lower health, while power-ups will *raise* health.

- **Bullets:** One of the easiest types of weapons to implement is a projectile weapon (gun, magic staff, etc.). By spawning *bullet* sub objects, we can easily give players a way to fight off the hazards of our game world.
- **Game Session manager:** We will need an object that is keeping track of the game win / loss conditions. For this chapter, we'll be focusing on a 'lives' system and the respawning of the player when they lose all their health.

We'll also be looking ahead to related subsystems that will be implemented in future chapters.

- **Weapons:** We need objects that the player can equip. Everything from guns, swords, and whips. Ammo management will also need to be accounted for. Weapons require subsystems of their own, so we'll tackle those later.
- **UI:** The health of the player is one of the most important things to display in the UI. In the next chapter, we'll be learning all about Unity's User Interface system, and using it to display the necessary information about our combat system.

Making our Health System

The first of the subsystems we'll hook up is the **Health Manager**. This is a component that can be added to any object that can take damage and be destroyed.

Unlike other scripts created using the Inspector window, we're going to learn how to create this component directly in our Project's Asset window..

Go to the Project window and open the Assets>Scripts folder. Once there, use the right-click menu to select Create > C# Script, and use **HealthManager** as our script name.

Naming your New File: When you add a script this way, you'll notice that the name of the file is automatically highlighted. This is because Unity uses this filename to generate the class name and script. If you forget to do this, and attempt to rename the script later, you'll find that

your filename and class name will not match (with your class being called ‘NewBehaviourScript’). Be careful not to miss this critical step.

With our new HealthManager script made, let’s open it in Visual Studio and add the following member variables...

```
[SerializeField, Tooltip("The maximum health of this
object.")]
private float _healthMax = 10;

[SerializeField, Tooltip("The current health of this
object.")]
private float _healthCur = 10;

[SerializeField, Tooltip("Seconds of damage immunity after
being hit.")]
private float _invincibilityFramesMax = 1;

[SerializeField, Tooltip("Remaining seconds of immunity after
being hit.")]
private float _invincibilityFramesCur = 0;

[SerializeField, Tooltip("Is this object dead.")]
private bool _isDead = false;
```

healthMax

- This is the maximum health of the object we’re managing. For now, we’ll set the default health of any object to 10.

healthCur

- This is the current health of the object whose health we’re managing. By default, it starts at the maximum health level. When this reaches zero, the object will be set to ‘Is Dead’.

invincibilityFramesMax

- Invincibility frames represent an amount of time, in seconds, that an object is invincible after taking damage. This is an important trick in making your game fun, since it allows players a window of opportunity to escape a surprisingly deadly situation. Without invincibility frames, a combat system will result in “cheap deaths”, “

frustrating gameplay” and possibly “broken controllers from being thrown at a wall”. Always remember to implement invincibility frames!

Game Design Sidenote: You’ll notice that, by default, the max invincibility frames are set to 0, meaning that most objects will not make use of the feature. This is because most objects are enemies or destructible environment objects, where every attack should be registered. Failing to disabled Invincibility Frames here would result in combat that feels sloppy, and enemies that feel like a “bullet sponge”.

_invincibilityFramesCur

- This is the current invincibility frame countdown. If this is anything other than zero, incoming damage will be ignored.

_isDead

- Whether or not this object’s health has depleted. Because there may be some data cleanup, or perhaps an animation to play after dying, we will store the object death as a Boolean value, then handle the destruction of this game object later.

With our members in place, it’s time to hook up some functions that we’ll need to call at key points in the game.

1. We’ll start off with **AdjustCurHealth()** - the function for incrementing and decrementing an object’s health.

```
public float AdjustCurHealth(float change)
{
    // leave early if we've just been hit and we're trying to
    // apply damage
    if (_invincibilityFramesCur > 0)
        return _healthCur;

    // adjust the health
    _healthCur += change;

    // check for health limits
    if (_healthCur <= 0)
    {
```

```

    // this object has died, so start the process to destroy
    it
    onDeath();
}
else if (_healthCur >= _healthMax)
{
    // this object has more health than it should
    // so cap it to its max
    _healthCur = _healthMax;
}
// should we be invincible after a hit?
if (change < 0 && _invincibilityFramesMax > 0)
    _invincibilityFramesCur = _invincibilityFramesMax;
return _healthCur;
}

```

The function will take in a positive or negative change value, allowing us to take damage or heal an object using the same function. At the top of the function, it will also check to see if the object has invincibility frames active, then leave early if damage is attempting to be applied.

The function will also return the adjusted health, in case the code calling the function needs to update any UI.

At this point you should also notice a red line under the `onDeath()` call. This is because we're referencing a function that doesn't yet exist. If you click on the little red X icon below your code, it'll unhide Visual Studio's **Error List**. Because Visual Studio is constantly checking for problems as you write code, you'll see an error telling you exactly what function is missing.

Figure 3.1: The compiler is rightfully angry, since we called `onDeath()` before it was defined.

- Because we don't want errors in our code (and because Unity won't let you run a project with errors) let's add the missing **onDeath()** function now...

```

void onDeath()
{
    if (_healthCur > 0)

```

```

{
    Debug.Log(gameObject.name + " set as dead before health
reached 0.");
}
_isDead = true;
}

```

Almost as important as the `onDeath` function is a public **IsDead()** function, which will allow other scripts to know whether or not this object should be active.

```

public bool IsDead()
{
    return _isDead;
}

```

Another function we'll eventually need is **Reset()**, which will set all the data of this player back to its default state. Let's add that function with the following lines of code...

```

public void Reset()
{
    _isDead = false;
    _healthCur = _healthMax;
    _invincibilityFramesCur = 0;
}

```

The final bit of logic we'll be adding is in the **Update()** function, where we'll be adjusting the invincibility frame counter if necessary. We'll also put in some temporary code to simply destroy the object when it's been set to `_isDead`. While eventually we'll add animations, particle effects, and musical cues based on the player's death, it's currently easier to simply make them disappear.

```

void Update()
{
    // decrement the invincibility timer, if necessary
    if(_invincibilityFramesCur > 0)
    {
        _invincibilityFramesCur -= Time.deltaTime;
        if(_invincibilityFramesCur < 0)
            _invincibilityFramesCur = 0;
}

```

```

    }
    // handle death
    if (IsDead())
        GameObject.Destroy( gameObject );
}

```

- With the HealthManager component made, let's return to Unity and add it to our **PlayerObj_Sphere** object. Once the component is added, our player can take damage and eventually be destroyed when health reaches zero.

Figure 3.2: Use the Add Component “Search Bar” to quickly find our HealthManager component.

There's one big problem, though - our scene lacks *danger!* There are no enemies, hazards, or obstacles to present a challenge to the player. With nothing to deal damage, our HealthManager has nothing to do, and the player has nothing to fear.

Let's Storyboard: Taking damage

While most of your development time will be spent in Unity, it's important to do some of your design work with a pen and paper - writing and sketching ideas away from your computer.

One such process is called **Storyboarding**: drawing scenes from your game to mentally plan out gameplay ideas. By sketching these gameplay mechanics, you can quickly brainstorm a concept and narrow down your solution before writing any code. By Storyboarding, you can save *a lot* of wasted development time by pruning bad solutions before trying to implement them.

Let's put this to practice. We have a health system, but now we need some hazards to inflict damage on the player.

This is a great opportunity to put away the computer and grab your pencil and paper! By storyboarding different options, we can flex our creative muscles and figure out the best (and simplest) way to implement our new damage sub-system.

The K.I.S.S. Philosophy: The design philosophy of Keep It Short and Simple (KISS) should always be your goal when designing a basic object to build a new system upon. While we can have fun designing these hazards, our #1 priority is to make a simple object that we can build our ‘Damage Dealing’ components upon. We will build more complex hazards from this basic starting point, so let’s keep KISS in mind.

Figure 3.3: Storyboarding our first enemy. A post-apocalyptic warrior with a throwable Plasma Spear, perhaps?

The first idea that comes to mind is a mobile enemy. Something that moves around the playing field, and when the player comes into view, the enemy tries to attack them. I’m sure we’ve all played games with this type of enemy, which we sketched up in [Figure 3.3](#)

By filling out a few panels, we can see this enemy type *probably* requires more subsystems than we’re ready for. The logic to control movement, hunt for the player and attack them, would all be handled using AI: a complex component that will require its own chapter.

Stationary turrets are also a common level design tool. A single object that fires bullets, cannonballs, or other projectiles at regular intervals.

Figure 3.4: The turret has a lot of potential...especially last panel!

This is certainly easier to hook up than our initial enemy, but still requires a bit more coding than we’d like. In the interest of keeping our dangerous object *dangerously simple*, let’s consider the basic ‘Spike’ hazard, as we’ve sketched out in [Figure 3.5](#):

Figure 3.5: Sketching out a basic Spike hazard. Not as exciting as the other two, but certainly more achievable as our initial damage-dealing object.

A staple of almost any level design toolkit, spikes provide a simple type of object that damages the player on contact. Since we already have code that handles object collision, the only component we would need to create is the one that deals damage.

Spikes: Your First Hazard

While not the flashiest foe, Spike objects give us a simple way to deal damage to the player.

And by picking the middle ‘Floating Spike’ from our sketches, we ensure this initial enemy has a wide variety of uses (opposed to floor-based spikes, which could only be used on the ground).

1. Let’s start building our first hazard by creating three new objects in the Hierarchy window: one empty GameObject (created using the “Create Empty” option) and two 3D Cubes (created using the “3D Object > Cube” option). Rename the empty GameObject to **EnemyObj_Spikes**.

Figure 3.6: We’ll be White-Boxing our Spikes using several Cubes.

2. Select the first Cube, then rotate and flatten it in the transform panel. Set the rotation to 45 on the X-Axis, and set the scale to 0.2, also on the X-Axis.

Figure 3.7: The first Cube after being rotated and scaled

3. Next, select the second Cube - likely named **Cube (1)** - and flatten it with a scale of 0.05 on the X-Axis. This should result in a thin Cube sitting inside the thicker rotated Cube.

Figure 3.8: At this point, our rotated cubes will look like an origami ninja-star.

4. The next step is to make this object *unquestionably* dangerous. When prototyping a game, the quickest way to visually convey danger is to make your hazards **red**. This color indicates danger *instinctually*, so players will immediately understand the threat it poses.
5. If you haven’t already made a red enemy material, let’s make one now by creating a new **Materials** folder in our Assets window. Open this folder and right-click to select the **New > Material** option.

Figure 3.9: Setting the color of our new Material in the ‘Albedo’ parameter.

6. Rename this new material to “matEnemyRed” in the inspector window, then change it’s ‘Albedo’ value to something closer to red.

With your new material ready, drag it from your assets folder to the larger rotated Cube. This fresh coat of paint should instantly make it look more dangerous.

Figure 3.10: Drag the red material onto the larger cube. Now it looks like a proper Hazard!

We now need to make sure all of our objects are nested properly in the hierarchy window.

7. Start by renaming the larger Red Cube to “RedCube” and the smaller White Cube to “WhiteSpikes”. Staying in the hierarchy window, you’ll want to drag the white spikes entry onto the Red Cube entry.

This makes the spikes a child object of the Red Cube, and you’ll know it works if a small arrow appears next to the “RedCube” entry.. Any time the Red Cube moves, rotates or changes scale, those changes will also affect the “WhiteSpikes” object.

In addition to the WhiteSpikes, you’ll also want to drag the RedCube entry on to the EnemyObj_Spikes entry. The Spikes should now be a parent object with two nested subobjects.

Figure 3.11: The proper hierarchy of our Spike hazard. Note the arrows to the left of the parent objects.

We’re now going to duplicate the Red Cube object, and because we’ve given it a child object, that will also be duplicated.

8. Select the Red Cube, and **Duplicate** it using the hotkey **CTRL + D**. Do it a second time to ensure you have the original red tube and two copies. Note the (1) and (2) now appended to the names of these duplicated objects.
9. Select the first duplicate, named “RedCube (1)”. In its Transform panel, set its rotation values to **45, 90, 0**. Then select the second duplicate, “RedCube (2)”, and set it’s rotation values to **0, 45, 90**.

With those objects duplicated and rotated, we have something that looks nice and spiky...

Figure 3.12: Our Spike - Simple, but effective. Players won't want to touch these.

10. The last step is to add a ‘Sphere Collider’ component to the **EnemyObj_Spikes** parent object. Select EnemyObj_Spikes in the Hierarchy window, then add a Sphere Collider using the Inspector.
11. Select the “Is Trigger” option, then reduce the radius of the sphere to 0.75.

Figure 3.13: When dealing damage, it's always better to make your hit-detection collider too small than too big.

With the Spike object made, it's time to make a component that can deal some damage!

The HealthModifier Component

The **HealthModifier** component will be a script added to objects that can alter the health of objects they collide with. Our spikes, for instance, will have a health modifier that decreases the health of the Player object. Bullets spawned by the player, on the other hand, will have health modifiers that lower the health of enemies. And power-up items will have health modifiers that increase the player’s health.

By making this component as modular as possible, we'll be able to get a wide range of game mechanics out of it.

Let's make the script by going to your **Projects > Assets** window, opening your **Scripts** folder, and adding a new script named “HealthModifier”.

Open the new script in Visual Studio and give it the following members:

```
[SerializeField, Tooltip("Change to health when applied to an object.")]  
float _healthChange = 0;  
[SerializeField, Tooltip("The class of object that should be damaged.")]  
DamageTarget _applyToTarget = DamageTarget.Player;
```

```

public enum DamageTarget
{
    Player,
    Enemies,
    All,
    None
}
[SerializeField, Tooltip("Should object self-destruct on
collision?")]
bool _destroyOnCollision = false;

```

_healthChange

- This is the amount of damage that will be applied when a valid target is collided with.

_applyToTarget

- While our initial implementation of the damage component is intended to hurt the player object, we will eventually be hooking up weapons that can also damage enemies. To allow for this, our damage dealing component should take into account what game objects it should apply damage to. The **_applyToTarget** member, along with the **DamageTarget**, will be used to determine the validity of the collided object.

_destroyOnCollision

- If this is true, this object will destroy itself when a valid object is collided with. This will be TRUE for bullets, but FALSE for Spikes and other more permanent health modifying objects.

With our members hooked up, it's time to implement the collision logic. Add this code below the **Update()** function...

```

void onCollisionEnter(Collision collision)
{
    GameObject hitObj = collision.gameObject;
    // get the HealthManager of the object we've hit
    HealthManager healthManager =
    hitObj.GetComponent<HealthManager>();
    if (healthManager && IsValidTarget(hitObj))

```

```

{
    // apply the damage as negative health to this object
    healthManager.AdjustCurHealth(_healthChange);
    // should we self-destruct after dealing damage?
    if ( _destroyOnCollision )
        GameObject.Destroy(gameObject);
}
}

bool IsValidTarget( GameObject possibleTarget )
{
    if (_applyToTarget == DamageTarget.All)
        return true;
    else if (_applyToTarget == DamageTarget.None)
        return false;
    else if (_applyToTarget == DamageTarget.Player &&
        possibleTarget.GetComponent<PlayerController>())
        return true;
    else if (_applyToTarget == DamageTarget.Enemies &&
        possibleTarget.GetComponent<AIBrain>())
        return true;
    // not a valid target
    return false;
}

```

The collision function simply grabs the object collided with, checks to see if it matches the target type, then applies (and self-destructs) if necessary. Also, to keep our code as clean as possible, we've broken the **IsValidTarget()** check into its own separate function.

Breaking up Requirement Checks: It's always a good practice to break requirement validation checks into their own separate functions. This allows us to perform the checks in a wider variety of locations, like AI (enemy logic) and UI (displaying information to the player).

You will also notice the compiler angry about the missing **AIBrain** class. While eventually we'll create an AIBrain to drive the movement and decision making of enemies, the script we make for this Chapter will be completely empty. This allows us to use this component for bookkeeping

purposes (like checking if the object is the Player or a Spike hazard) without committing to the full implementation of the feature.

Let's deal with this compiler error by returning to the Editor. Back in Unity, make sure the Spike object is selected. In the Inspector window, scroll down and press **Add Component** > **New Script** to add a component named **AIBrain**. Our Spike now has an empty component that we can check against, and we can deal with the actual implementation of AIBrain in a later chapter.

Now that the compiler error is fixed, we can now add our finished HealthModifier component to the Spike hazard. Make sure the object **EnemyObj_Spikes** is still selected, and again press **Add Component** to add the HealthModifier script to the Spike. Make sure the component data matches [figure 3.14](#) in the inspector panel.

Figure 3.14: Our HealthModifier component, when added to the Spikes.

Let's set the damage dealt by the spike to -5, and the Target to Player. With these parameters set, the player will only survive two collisions with the spike.

Now make the prefab by dragging the spike from the Hierarchy panel into the prefabs folder of our projects asset directory.

Figure 3.15: Our Spike hazard in the Prefabs folder.

We now have a dangerous Spike prefab that can be placed throughout the level. The player will have to time jumps and movement to ensure they don't take too much damage. Our objective of a simple, straightforward damage-dealing object has been met. Congratulations!

Feel free to drag a few spikes from the prefab folder and into the scene. The more spikes, the more difficult the level.

Spike Spawn Zone: With our Spike now upgraded to a Prefab, there are some cool things we can do with it. One possibility is to utilize the Spawning Zone component - previously used for placing coins - and instead place Spikes. Do this by duplicating the "ItemSpawnZone" object in the Hierarchy (select and press CTRL + D). Rename the clone

to “SpikeSpawnZone”, set it’s “Item Count” to 10, then drag our new prefab into the “Item to Spawn” parameter. While not a necessary step, spawning more Hazards this way highlights the power of Unity’s component system, and the importance of flexible, modular code.

Figure 3.16: Any Prefabs you make can now be spawned by our SpawnZone object.

With everything in place, it’s time to press Play and put our new systems to the test. If we did everything correctly, colliding with a spike twice will result in your player object being destroyed.

Press the Play button again to stop the game, reset the scene, and re-spawn our player.

Now that the player can take damage (and be destroyed), it’s only right that we give them a way to fight back.

The Basic Bullet

There are a multitude of different ways to let the player inflict damage on enemy objects. Whether it’s jumping on the enemies’ head, casting a spell, or gobbling them up to absorb their powers - there’s an endless variety of creative ways a hero can fight their foes.

For this first iteration of our combat system, we’ll be keeping things simple by implementing a **Bullet** projectile as our weapon of choice.

Just like with the spike object, the goal here is not to implement something big and flashy, but instead hook up a basic system that we can build upon. Since projectiles can be used for a wide assortment of weapon types (laser rifles, missiles, fireballs, arrows, etc.) it only makes sense that a basic Bullet is our starting point.

1. Start by creating a new 3D Sphere object in the hierarchy window. Once made, rename it to **BulletObj_Basic**. With so many objects in our scene now, it’s more important than ever to keep objects named in an informative way.
2. With our projectile object selected, let’s make it look a bit more *bullet-like*. In the transform panel, set it’s **Scale** to **0.1, 0.1, 0.6**. This will

give it an oblong oval quality that helps to convey direction better than a basic sphere can.

3. Next, add a rigid body component, so this object will properly trigger our collision functions. Set the RigidBody's **Is Trigger** parameter to Enabled. Our basic bullet should use a trigger collider since we do not want it applying physical forces to the other objects.

The Importance of RigidBody: It's important to remember that all collision functions require a RigidBody component to work. If you only have a collider script on your object, your Collision functions (OnCollisionEnter, OnTriggerEnter, etc.) will never be called. It's easy to lose valuable time debugging collision code due to a missing RigidBody script. Remember to attach one to any objects that have Collision handling functions in their code.

Much like with the spikes, we want to give our bullets a different color so they are easily identifiable.

4. Open your **Assets > Materials** folder, right click, and select **Create > Material** in the rick-click menu. Rename this new material **matBullet** and change its **Albedo** property to a Light Blue color of your choice. You can also give it an **Emission** color, which changes the shadow tint, giving the bullet a 'self-illuminated' feel.

Figure 3.17: Our light blue, self-illuminated 'Bullet' material.

5. Drag the new material onto the Bullet object. You should see the change instantly.
6. With our basic shape and color set, it's time to make the actual Bullet script. Select the object, and use **Add Component > New Script** to add a new **Bullet** component.
7. In Visual Studio, add these members and functions...

```
[SerializeField, Tooltip("Speed of this bullet.")]  
private float _speed = 4f;  
[SerializeField, Tooltip("Normalized direction of this  
bullet.")]  
private Vector3 _direction = Vector3.zero;  
void Update()
```

```

{
    // move the bullet
    Vector3 newPos = transform.position;
    newPos += _direction * (_speed * Time.deltaTime);
    transform.position = newPos;
}
public void SetDirection( Vector3 direction )
{
    _direction = direction;
    // rotate to face the direction of movement
    transform.LookAt(transform.position + _direction);
}

```

There are two main attributes of a projectile: it's speed and direction. With those members in place, we only need the **Update()** function to move the bullet and a **SetDirection()** function that allows other components - in this case, the Player - to set the Bullet's direction.

- Now select the **BulletObj_Basic** object from the Hierarchy window and drag it into the **Project > Assets > Prefabs** folder to turn it into a prefab.

Figure 3.18: Our finalized Bullet prefab.

- With our bullet prefab ready for use, we need a way for the player to attack with it. Open the **PlayerController** script in Visual Studio, and define two new members...

```

[SerializeField, Tooltip("The bullet projectile prefab to
fire.")]
private GameObject _bulletToSpawn;

[Tooltip("The direction that the Player is facing.")]
Vector3 _curFacing = Vector3.zero;

```

We'll set the **_bulletToSpawn** parameter back in the editor, and the value we'll set down in **_curFacing** the **Update()** function. Place the following code between the handling of the Arrow Key input, but before friction is applied...

```

// store the current facing
// do this after speed is adjusted by arrow keys

```

```
// be before friction is applied  
if (curSpeed.x != 0 && curSpeed.z != 0)  
    _curFacing = curSpeed.normalized;
```

10. The last bit of code to add is an input check to see if the player wants to fire their weapon. Add this code in the **Update()** function...

```
// fire the weapon?  
if ( Input.GetKeyDown(KeyCode.Return) )  
{  
    GameObject newBullet = Instantiate(_bulletToSpawn,  
    transform.position, Quaternion.identity);  
    Bullet bullet = newBullet.GetComponent<Bullet>();  
    if (bullet)  
        bullet.SetDirection( new Vector3( _curFacing.x, 0f,  
        _curFacing.z ) );  
}
```

11. Now, if the player presses the Enter key, a bullet will be spawned in the direction our player is facing.
12. Return to the Unity editor, select the Player sphere, and drag the BulletObj_Basic prefab from the Project window and into PlayerController's **Bullet to Spawn** parameter.

Figure 3.19: Set the Bullet to Spawn by dragging the prefab from the Assets>Prefabs folder into the PlayerController component.

We're now ready to test!

13. Press Play and move your player around with the arrow keys. Pressing *Enter* will now fire a bullet in the direction you were facing last, and if you hit too many spikes, your player will disappear.

And while everything is in and working, something feels *off* when our player takes damage...

Damage Feedback

If you've played games before, you've probably noticed that our implementation of 'taking damage' doesn't *feel* right. Hitting spikes

certainly kills you eventually, but the visual feedback for taking damage lacks a *punch*. You can't tell that you're being hurt until it's too late.

This is part of **Player Feedback** - the concept where, when something important happens in the game code, you need to communicate that event to the player. And if there's one event we want the player to be aware of, It's when they take damage.

First, let's visually communicate our Invincibility Frames.

The health system Gives the player those nice 'invincibility frames' after being hit, but currently does a lousy job showing you when the player is actually invincible. The most common way to show invincibility frames in a game is to flicker the visibility of the Invincible object.

Do that now by opening the HealthManager script and adding the following code to the **Update()** function.

```
// handle visibility
if (GetComponent<MeshRenderer>())
{
    if (_invincibilityFramesCur > 0)
    {
        // toggle rendering on/off
        if (GetComponent<MeshRenderer>().enabled == true)
            GetComponent<MeshRenderer>().enabled = false;
        else
            GetComponent<MeshRenderer>().enabled = true;
    }
    else
    {
        GetComponent<MeshRenderer>().enabled = true;
    }
}
```

This code first checks to see if the object has a valid MeshRenderer script (this is the component that draws the object on the screen). If valid, it then checks if our current Invincibility Frames. If we're invincible, it toggles the enabled state between True and False. If we're *not* invincible, the MeshRenderer defaults to its enabled state.

Enabling and Disabling Components: All MonoBehaviour components can make use of the ‘Enabled’ member. This is a Boolean (a type that can be True or False) that determines if that script should be run. In the above code, we’re toggling the visibility of an object by Enabling / Disabling the component that renders the object (the MeshRenderer), but it can be used for any component Unity provides. You can even Enable or Disable your own components, if you ever find it necessary. And who knows....you may even need to do it in the next paragraph.

Next, let’s apply a subtle Camera Shake when damage is taken.

A nice, unobtrusive way to indicate that you’ve taken damage is to give the camera a little shake. We can easily do this by selecting the main camera object, adding a new script called **CameraShake**, and filling the class with these lines of code...

```
[SerializeField, Tooltip("Magnitude of the shake effect.")]
float _shake = 0.05f;

Vector3 _startPos;

private void Start()
{
    // store the starting position
    _startPos = transform.position;
}

void Update()
{
    // if enabled, give camera a little shake
    Vector3 newPosition = new Vector3();
    newPosition.x = _startPos.x + Random.Range(-_shake, _shake);
    newPosition.y = _startPos.y + Random.Range(-_shake, _shake);
    newPosition.z = _startPos.z + Random.Range(-_shake, _shake);

    transform.position = newPosition;
}
```

All this code does is store the starting location of our camera, then randomize its position every **Update()** frame.

If you were to run the game at this point, you’d notice that the camera is perpetually shaking. We can keep this from happening in the Unity Editor

by deselecting the check mark next to the **CameraShake** component.

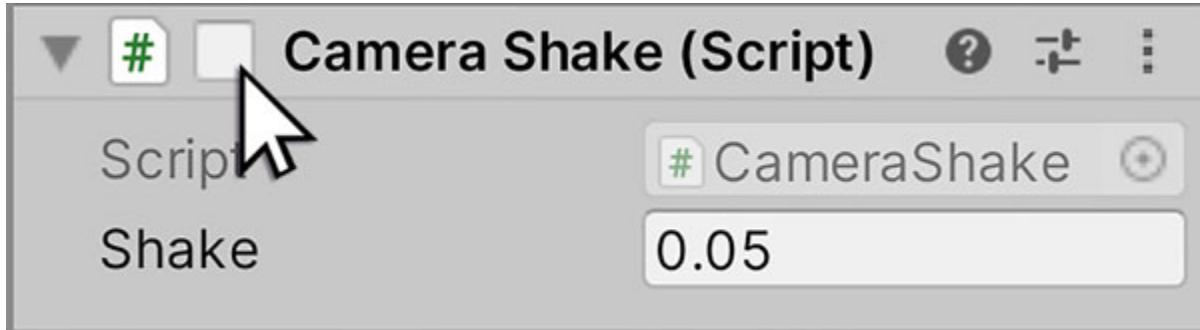


Figure 3.20: Disable components in Unity using the Check Mark toggle in the upper left corner.

With the component ready, we need to add code to toggle it. Find the **Update()** function in our **HealthManager** script, and add the following lines...

```
// toggle camera shake for the player
if (GetComponent<PlayerController>())
{
    CameraShake camShake = Camera.main.GetComponent<CameraShake>
    ();
    if (camShake)
        camShake.enabled = (bool)({_invincibilityFramesCur > 0});
}
```

This checks to see if the object is the Player, then enables (or disables) the CameraShake component based on the status of the invincibility frames. If the frame counter is greater than 0, the statement is True and the Camera Shake is enabled. If the counter is 0 or less, the statement is False, disabling the Camera Shake.

If you press play and collide with a spike, you will now see your player sphere flicker and the camera shake a bit.

For now, we'll stop with those two improvements, but *basic* feedback is far better than NO feedback.

Unfortunately, we still have a problem. When the player takes 2 hits and dies, the game enters a broken state known as a **Soft Lock**.

Hard-Lock vs. Soft-Lock: There are two gameplay states that you want to avoid at all costs: Hard-Lock and Soft-Lock. When a game locks up, it means that the player is forced to restart the game to keep playing. In

a ‘Hard-Lock’, this is because the game has crashed, or is stuck in a recursive loop that it can’t escape from. A ‘Soft-Lock’ means that the code is running as expected, but we’ve failed to design a way for the player to continue their session via input. Both are considered game-breaking bugs and should be considered a top-priority fix.

To fix our softlock state, we need to communicate with the player that they lost a life and, if they have more lives available, respawn them into the scene.

This will require our final component of the chapter - a script to manage the game state and respawn our player: the GameSessionManager.

The Game Session Manager

Since you can’t have a combat system without managing the death of the player, the last component of this chapter is **GameSessionManager**.

This vital component will drive the flow of your game, continuing the game when you lose a life, and ending the game when all lives are depleted. In later chapters, this will be the system that is constantly checking the winning conditions of a level. Even our pause system will go through this game-wide management system.

Let’s start by placing an empty GameObject in the hierarchy. Name it **GameSessionManager**, then create and add a new script of the same name.

Once in the GameSessionManager code, we only need to add three member variables...

```
[Tooltip("Remaining player lives.")]  
private int _playerLives = 3;  
  
[SerializeField, Tooltip("Where the player will respawn.")]  
private Transform _respawnLocation;  
static public GameSessionManager Instance;
```

These members will help us to manage the respawning of the player if they lose all their health. The **_playerLives** value will keep track of the times a player can respawn. The **_respawnLocation** transform will hold a location that we’ll use to respawn the player at.

Our third member, **Instance**, is used for a new programming concept: the **Singleton** object. A singleton is a component where there should only be

one of these active at a time. Where a coin, bullet, or enemy is meant to be duplicated and multiplied, the `GameSessionManager` is designed where there should only be *one instance* at a time.

Singletons are a concept we'll learn more about in a later chapter, but for now, we'll need to write an `Awake()` function that looks like this...

```
void Awake()
{
    // the GameSessionManager is a Singleton
    // store this as the instance of this object
    Instance = this;
}
```

With all of our three members in place, and our `Instance` member being stored, we can now add the function that will respawn the player when necessary.

```
public void onPlayerDeath( GameObject player )
{
    if (_playerLives <= 0)
    {
        // player is out of lives
        GameObject.Destroy(player.gameObject);
        Debug.Log("Game over!");
    }
    else
    {
        // use a life to respawn the player
        _playerLives--;

        // reset health
        HealthManager playerHealth =
        player.GetComponent<HealthManager>();
        if (playerHealth)
            playerHealth.Reset();
        if (_respawnPostion)
            player.transform.position = _respawnPostion.position;
        Debug.Log("Player lives remaining: " + _playerLives);
    }
}
```

The **onPlayerDeath()** function handles the specifics of managing a player's remaining lives and respawning the player, if necessary. Let's open the HealthManager script and, in the **Update()** function, tweak the code where it manages our **_isDead** logic...

```
// handle death
if (_isDead)
{
    if (GetComponent<PlayerController>())
        GameSessionManager.Instance.onPlayerDeath(gameObject);
    else
        GameObject.Destroy(gameObject);
}
```

Where we used to simply destroy any dead object, the health manager will now treat the death of the player differently than the death of other objects.

Save All: When juggling the code of several scripts, it's easy to end up making changes and forgetting to save them. A good habit is to press SHIFT+CTRL+S before returning to Unity. This hotkey combination will ‘Save All Files’, meaning all changes will be saved and made ready for testing. Save yourself wasted debugging time and frustrations by making SHIFT+CTRL+S a key part of your development routine.

The last step is to hook up the transform object used for respawning. This can be an empty gameobject, so use the Hierarchy window to add an empty object named “RespawnLoc”, and set its position to **0,10,0**.

Figure 3.21: This empty GameObject represents the location that the player will respawn.

With the respawn locator created, let's place it into our GameSessionManager component. Select the GameSessionManager object in the Hierarchy, and drag the RespawnLoc object into the Inspector panel where it says “Respawn Position”.

Figure 3.22: Drag the RespawnLoc object into the GameSessionManager component.

With everything in place, it's time to test our new code. Press the play button, and run into a few of the spikes. Once you've lost all your health,

the sphere will respawn at the position of the RespawnLoc object. Once you've lost all your Lives, the player sphere will disappear and be lost forever.

Conclusion

With several new components created and tested, we now have a solid foundation that our combat system can be built upon.

We also learned about storyboarding, and the concept of choosing simple solutions when implementing new systems. Instead of creating an intelligent enemy that has to hunt down the player, we went with simplicity and implemented a basic Spike hazard. The same thing is true with our weapon implementation. We remembered to *Keep It Short and Simple*, and ensure our foundations are solid before implementing bigger and bolder ideas.

Don't worry, though - we'll get to 'bigger and bolder' soon enough.

And while we did get our **GameSessionManager** implemented and respawning the player, there's still a soft lock when all lives are spent. We're now at the point where the game requires more than a single scene to properly flow.

It's time to tackle game flow head on. It's time to learn about UI!

Questions

1. We talked about how Combat presents the player with interesting (and possibly deadly) choices. What can you learn from your favorite games when it comes to designing systems that provide compelling choices? What choices have you encountered, in your favorite games, that have stuck with you?
2. Risk and Reward is a concept that balances the difficulties of a game with a payoff for overcoming those difficulties. Again, thinking back to previous games you've played, what games did a great job challenging you, then giving you a satisfying reward? What about games that challenged you, only to give a lame reward that caused disappointment?

3. True or False: When designing a new system, you should make it as complex as possible, covering all possible use-cases before playtesting it.
4. Creating informative Visual Feedback is a critical part of the design process. You need to communicate to the player (through subtle methods) what's happening in your game so they can respond accordingly. We've added a Camera Shake to show the player when they've taken damage, but what other methods of Visual Feedback have you seen used in games?
5. Combat is an easy system to get your player excited, but what games have you played that *don't* have a combat system? What did these games do to keep you engaged?

Key Terms

- **Risk and Reward:** The game design concept where the player must balance the Risks and Rewards of a situation, and the enjoyment of intelligently making the correct choices.
- **Subsystems:** Smaller, modular components that come together to create a larger gameplay system.
- **Storyboarding:** Using pen and paper to sketch out gameplay ideas. Great for brainstorming and design sessions.
- **Health Manager:** The component that handles the health and damage taken by an object.
- **Nested Objects:** Placing an object inside another object sets one as the Parent, one as the Child. When the Parent moves, rotates, or scales, the Child object will too.
- **Projectile:** An object, used for attacking purposes, that inflicts damage on the target. Normally destroyed on impact.
- **Player Feedback:** The concept that your game needs to communicate big events to the player, either through visuals, sound, or controller rumble.
- **.enable:** An important member of any MonoBehaviour script. Allows you to enable or disable a component in code.

- **Camera Shake:** An effect where the camera's position is subtly randomized. Normally indicates the player has taken damage, or something big is happening in the scene.
- **Hard and Soft Locks:** A game state where the player cannot continue without physically restarting the game. Caused by an infinite loop, broken game flow, or some other critical issue.
- **Singleton:** A globally accessible component, where only one instance exists at a given time. Normally used for systems or management scripts.

CHAPTER 4

Getting to Know UI

Video games, by their very nature, are *amazingly* complex.

Juggling player stats, level navigation, items, and victory/loss conditions give the player a lot to manage. This becomes even harder when a horde of enemies is programmed to *end your existence*.

All this complexity makes a well-crafted **User Interface (UI)** and polished **User Experience (UX)** vital to the enjoyment of any game. What information do you show the player? What kind of sound cues should you use? Where should you use large fonts vs. tiny fonts?

What is the best RGB value to use for the color *red*? Hint: It's not 255,0,0.

Designing the best User Interface for your game can be challenging. Fortunately for you (and your users) Unity provides great tools for building and iterating on your UI/UX.

Structure

- Game Flow vs. Gameplay Flow
- Common Screens in a Game
- The ‘Unity UI’ System
- UI Elements
- The Importance of Anchors
- Making a Title Menu
- Anchor Presets
- Scene Management
- Prototyping a HUD (Heads Up Display)
- Handing ‘Game Over’

Objective

By the end of this chapter, you'll have a strong understanding of Unity's UI systems, from the base Canvas to the Images, Buttons, and Text objects that you'll build your screens from.

We'll also cover the game screens that many games make use of, and go through the process of implement several of them for our demo.

Game Flow (UI) vs Gameplay Flow (Immersion)

Before jumping into the UI, there are two important game design concepts you've probably heard of, but may require differentiation: **Game Flow** vs **Gameplay Flow**.

Game Flow is a player's UI experience from the opening Title Screen, through the Frontend UI, into the main game, and everything in-between. It's the ease and quickness that players can navigate to the point where they're actually 'playing' the game. Designing a game for the modern gamer means you only have 10 minutes to captivate them. A strong Game Flow ensures they'll be enjoying themselves as quickly as possible.

Designing with **Gameplay Flow** in mind, however, is creating immersive 'Second-to-Second' gameplay that the user can't stop playing (known as the 'One-More-Turn Effect' in strategy games). This is making movement feel great, combat feel responsive, and ensuring the player always knows where to go. When someone talks about great *Gameplay Flow*, they tend to be talking about well-crafted, polished gameplay mechanics and pacing.

In this chapter, we'll deal more with **Game Flow** and the UI needed to navigate the overall experience. *Gameplay Flow*, in the form of level design, player progression, and onboarding, will come in later chapters.

Game Flow Breakdown

With our White-Boxed prototype in a playable state, we should take a step back and look at our Game Flow. Specifically, how should the screens take the player from starting the application to enjoying the core experience.

Many of these concepts may seem obvious to a veteran game designer (and even veteran gamers), but it's important to itemize each screen when planning the UI flow of our demo.

- **Title Menu:** This is where you first get to impress the player. The game logo needs to look great, the music needs to set the tone, and any visuals needs to foreshadow the adventure ahead. This is also the main hub from which the player will access things like loading games, setting options, viewing the store, and being informed when community events occur.
- **Main Game HUD:** The Heads Up Display - or HUD - are the UI elements displayed when the game is actually being played. Things like Health, score, and available lives will be presented to the player here.
- **Pause Menu:** All games should allow the player to stop the action and step away from the game for a bit. When this happens - normally when the Start button is pressed - a Pause Menu will appear. This should also have a way to bring up the Options Menu, allowing players to tweak Visual and Audio settings without having to revisit the Title Menu.
- **Game Over screen:** When the player wins or loses, you'll want to bring up a screen letting them know of their Victory or Defeat. To complete our Game Flow loop, this needs to return them to the Title Screen.

Other screens worth mentioning...

- **Inventory Screen:** Any game where you're gathering resources, collecting loot, or equipping items, you'll need a corresponding Inventory Screen. This is traditionally a grid of items that the player has collected, and is normally tied directly to the Pause Menu.
- **Dialog Popup:** If you have a game with NPCs, or perhaps chatty enemies, you'll need to put some thought into a dialogue pop-up window. This will normally pause the action to give the player time to

read and absorb the dialogue presented. Character portraits also help give personality to the inhabitants of your game world.

- **Cutscenes:** There are times when large story events need to be presented visually to the player. These Cutscenes require their own screen to be presented, either as a pre-rendered movie or in-game scripted event. They should always have some way to skip them, for players that are more invested in gameplay than story. Speedrunners like a skip option, too.
- **Options menu:** A robust options menu may feel like busy work, but ensuring that every player has the options they need to enjoy your game is a great use of your time. Screen Resolution options, GFX quality settings, Audio sliders, and Colorblind toggles all ensure the largest number of players will be able to enjoy your game.
- **Loading Screen:** If developing a game for PC or console, you're most likely going to have save points integrated into the game loop. If this is the case, you'll need a Loading Screen hooked up so the player can easily pick up where they left off.
- **Storefront and Community Screens:** If you're developing a game with a strong online component, it's important to give the player easy access to your online Store and Community forums. Building an online community is easiest when tools for notification and engagement are built right into the game.

With the key Game Flow concepts defined, it's time to dive into the Unity UI systems that we'll be learning, utilizing, and mastering.

The Unity UI System

Built into the Unity Engine is a suite of well tested UI objects that you can use when building your screens. The **Unity UI** objects are all built from GameObjects, meaning all the MonoDevelop functionality you've learned in previous chapters will be 100% applicable here.

Adding UI elements is done just like adding 3D objects, like the Sphere and Cubes from previous chapters. Right-Click in the Hierarchy window, and use the UI panel to view the objects you can make use of (as shown in [Figure 4.1](#)).

Figure 4.1: The Right-Click menu has all the User Interface objects you'll need.

You can easily access all available UI objects through the Right-Click menu. Let's go through the most important UI options now...

Image

Use an **Image** object to display icons or pictures in your UI. These use 2D sprites (PNGs, JPEGs, BMPs, etc.) to display an image on the screen.

Text - TextMeshPro

A **Text** element consists of a display string (the visible text), the font, font size, and text alignment options. Much like using a Word Processor, these parameters can all be easily tweaked in the editor's Inspector window.

TextMeshPro: Unity uses a font rendering system called TextMeshPro, which allows crisp text to be displayed as a 2D or 3D element. It's a powerful component of the UI system that makes text look great at different resolutions.

Button - TextMeshPro

Place a **Button** anywhere in the UI where you would like to take input from the user ([figure 4.2](#)). Buttons can be activated by either a mouse click, a keyboard press, or controller input. Buttons consist of Display Text, an Image, and different visual styles for the various button states (selected, highlighted, disabled, etc.).

Figure 4.2: This is indeed a button. It uses TextMeshPro for displaying crisp text at any scale.

Canvas

This is the required parent object of all UI elements. The **Canvas** manages the placement and scale of the UI objects within it, so if the resolution of the game changes, the canvas will update its child objects accordingly.

Besides the above UI Objects, there are several other aspects of the Unity UI system that you'll want to familiarize yourself with.

Font Assets

Since many UI Objects will have a text element associated with them, it'll be important to understand Font Assets and how they work in Unity.

Placement Anchors

The **Placement Anchor** of a UI element allows the canvas to know where to place an object in relation to other UI elements. Anchors can be used to snap an element to the side of its parent, indicate where to snap, and even control the stretching of an element along a given axis.

Placement anchors are a powerful aspect of Unity UI that we'll dive into shortly.

Figure 4.3: Anchors give you control over the relative placement of UI elements: very useful for building resolution-independent interfaces.

Object Stretching

The same tool that allows you to adjust the placement anchor of an object can also be used to set its **Stretching** preference. If you have an object that should span the width or height of its parent, you would set that by adjusting Stretching.

Now that we have a better understanding of Unity's UI system and objects, let's start hooking up our first new screen: The **Title Menu**.

Our Title Menu Canvas

The Title Menu Is one of the first screens a player will see. It's your opportunity to make a great initial impression, so when you eventually ship a game, make sure the title screen looks, sounds, and feels *amazing*.

For this demo, however, we'll be keeping our Title Menu *very* simple. A black background, some text, and a 'Start Game' button will serve as our prototype.

Our first step is to make a new scene. In the Projects window, create a new folder called **Scenes**, and once in made, use the right click menu to create a new Unity scene, as shown in [Figure 4.4](#).

Figure 4.4: In the Projects window, Use Create > Scene to make a new scene for our Title Menu.

Rename this new scene "TitleMenu" and double-click to open it. You are now presented with a completely fresh scene, consisting only of the Main Camera and Directional light that we were introduced to in [Chapter 1](#).

Since the purpose of this scene is to create a new UI menu, let's right click in the Hierarchy window and select **UI > Canvas**.

The canvas is the backbone of the Unity UI system. If you're going to be placing text, images, or buttons in your scene, they need to be children of a Canvas object.

Select your new Canvas so we can look over its Parameters in the inspector window...

Rect Transform

The position, rotation, and scale of a UI element is controlled by a Rect Transform component. This is similar to the basic Transform component, only with the addition of Width, Height, and Anchor parameters. All UI objects make use of a Rect Transform to determine where, and how, they should be placed.

Figure 4.5: Position, size, rotation, and anchor parameters of a Canvas object.

You'll notice that almost every setting in the Rect Transform is Disabled (you can tell by the gray text). These values are set by the Canvas component, as explained in the warning "Some Values Driven by Canvas". If you want to change these values, we'll have to do it by modifying that component.

The 'Canvas' Component

The Canvas component is what actually manages the size and positioning of any UI elements within it. As you can see in [Figure 4.6](#), the Canvas component contains several parameters of importance when setting up your UI...

Figure 4.6: The actual Canvas component.

- **Render Mode**
 - **Screen Space – Overlay**
 - The UI elements are drawn over all the other game elements, ensuring the in-game objects remain separate from the UI objects. Most game UIs use this rendering mode.
 - **Screen Space - Camera**
 - The canvas is rendered in 3D space, at a given distance in front of a specified Camera. All UI elements will be affected by any Camera properties (Field of View, Post Processing Effects, etc.) and can possibly clip into 3D objects.

- **World Space**
 - The UI is rendered in world space, like any other 3D object would be. Can result in clipping between the UI and other game objects, but is useful for screens that need to exist in the game world (computer terminals, keypads to unlock doors, and so on).
- **Pixel Perfect**
 - A rendering option to force objects to render on exact pixels. This will result in a crisp image, provided your transform values are rounded to whole numbers. Pixel Perfect rendering, mixed with granular transform values, may result in rendering oddities.
 - *This option is only for Screen Space rendering modes.*
- **Sort Order**
 - Since you can have multiple canvases in one scene, this value determines the drawing order of those. Larger Sort values will appear in front of Smaller sort values.
- **Target Display**
 - If you’re designing a project to work across multiple monitors, this lets you specify which display will show the elements of this Canvas.
 - *This option is only for Overlay Rendering mode.*

The ‘Canvas Scaler’ Component

A component that you can use for managing the Scaling of a Canvas object. While the default setting is “Constant Pixel Size”, we’re going to change the mode to “Scale with Screen Size”, which produces the most ideal results across monitors of various sizes.

Figure 4.7: The Canvas Scaler component, which lets you adjust how the UI Elements resize at various screen resolutions.

While a UI Scale Mode of “Constant Pixel Size” keeps all the objects the same size, no matter the resolution, the “Scale with Screen Size” will size and position of our objects in proportion to the **Reference Resolution**.

In all the screens we'll be making, set the **UI Scale Mode** to **Scale with Screen Size** and set the **Reference Resolution** to **1080 × 1920**.

Title Menu: Adding the UI Elements

With the Canvas in place, it's time to start adding UI Elements to it. Luckily, since we're in 'White-Boxing' mode, we only need to add three objects...

- **Game Name Text:** We'll add some text that will display the name of our game demo.
- **Start Game Button:** We'll need a button that can be clicked to start the game.
- **Black Background:** To keep our Title Menu simple, yet identifiable as different from our Main Game, we'll be adding a black background image.

First, let's add that Text element. Right-click on the Canvas object in the Hierarchy window. By right-clicking *directly on the object*, you ensure the UI elements made will be instantly assigned as children of that Canvas.

In the Right-Click popup, select the **UI > Text-TextMeshPro** option. Once created, you'll note the Text object is nested in the Canvas, and the name of the object is selected, if you want to rename it. Let's change the name to **GameNameText**.

With our text object selected, let's look over the TextMeshPro component in the Inspector window...

Figure 4.8: There are several useful formatting options when configuring your Text.

Let's quickly run through the most important of the **TextMeshPro** parameters...

Text input	The text that is displayed.
Text Style	The Style Sheet category you want to use.
Font Asset	Which font to use. New fonts require the creation of a unique TMP Font Asset.
Font Style	Set whether a font is Bold, Italic, Underline, etc.
Font Size	How large, or small, the font should be.
Auto-Size	Toggle this option if you want the font to intelligently resize when the text is too long.
Vertex Color	The color of the text.
Color Gradient	Toggle this option if you want the font to blend between multiple specified colors.
Spacing Options	Various font kerning settings.
Alignment	How the text should be justified within the specified RectTransform dimension.
Wrapping	Should the text wrap to a new line when it extends past the

	width of the object.
Overflow	Should the text extend outside the bounds of the object?

Table 4.1: The TextMeshPro parameters.

For our GameNameText object, let's make it larger by setting the Rect Transform **Width** to 900 and **Height** to 300. Set the **Text Input** parameter to "Mastering Unity: Demo Game", the **Font Size** to 64, and **Alignment** to Center.

Now, to make the text stand out a bit more, let's add a black background to our Title Menu. Following [figure 4.9](#), this will be a UI Image object, which can be made by Right-Clicking the Canvas object, and selecting **UI > Image**.

Figure 4.9: The Right-Click popup makes it easy to add Images to our Canvas.

Rename the new Image to "BlackBG", and with the object selected, go to the Inspector window and change the Image's **Color** parameter to Black (0,0,0). The end result will be a black square in the middle of the screen, as seen in [figure 4.10](#)

Figure 4.10: The current Title Screen. Our 'Black BG' is more of a 'Little Black Square'.

While this is the first step towards our 'Black Background', we'll need to learn about another UI system before we can get this object filling the screen.

Anchor Presets

Since we want to fill the background with the Black Image we've just made, we're going to need to adjust its Rect Transform settings. In particular, we need to make use of **Anchors**. With the **BlackBG** object selected, go to the **Rect Transform** component and click the Squares icon with the text "Middle Center" (shown in [Figure 4.11](#)).

Figure 4.11: Clicking this image will bring up the Anchor Presets.

Anchor Presets are an important tool when building your UI, especially when making a project that needs to run at multiple resolutions. Based on the selected Anchor position, and any Stretching preset you choose, your UI elements will alter their positions based on the game's resolution. Picking the right Anchor options is critical to making a flexible UI that works across various platforms (especially mobile, which we'll discuss in a later chapter).

As you can see in [Figure 4.12](#), there are 3 horizontal and 3 vertical Anchor snapping options. These are the relative sides used when calculating the position of an object. If our Anchor is in the **Left/Middle**, then resizing the screen - or simply resizing the parent object - will result in the object retaining its position relative to the left side of the parent object. By default, new UI elements use a **Middle/Center** Anchor, meaning they're placed relative to the centerpoint of their parent object.

Figure 4.12: Anchor Presets determine which side of the parent this UI Element snaps to.

It's a concept that is better explained using images.

In [figure 4.13](#), we have a gray background object with the Width set to 300 and the Height set to 200. Within that object, there are several UI child elements: A bar at the top, a toolbar on the leftmost side, a button in the lower-right corner, and a message box directly in the middle.

Figure 4.13: Our screen example at the starting dimensions...but how will it look when resized?

By default, all of these objects have a middle center anchor, which positions them relative to the *center* of their parent. If we assign the appropriate anchor presets to the top, left, and bottom-right objects, you can see the difference when we resize the parent, as shown in [Figure 4.14](#)

Figure 4.14: Left: All objects with a Middle-Center anchor. Right: objects anchored to the appropriate sides of their parent object.

As you can see in the first image, the objects all stay positioned relative to the center, and all the objects you would expect to snap to the sides are now floating in the middle of the screen. The second image shows a resized screen with the anchors properly set. The top bar snaps to the top, the toolbar snaps to the left side, and the button in the lower right corner stays positioned in that corner.

It bears repeating: *Anchors are vital when building a resolution independent UI.*

With the proper anchors set, deploying your game across multiple platforms will be *considerably* easier. Without consideration to your UI anchors, much of your porting time will be spent fixing a multitude of UI placement bugs.

As for our title screen, we're less concerned with placement anchors and more with the **Stretching** parameters available to our object.

Figure 4.15: You can stretch an UI object Horizontally, Vertically, or both.

Much like anchors controlling the placement of UI elements, stretching controls how they are scaled as a parent object is resized. There are

stretching options for vertical scaling, options for horizontal scaling, and the option that we are looking for: Stretch on All Axes.

With our Black BG object Selected, bring up the **Anchor Presets** menu and select **Stretch All** (the option in the lower-right corner).

The first thing you'll realize is that *nothing visually changed* when you selected this option. That is because Unity will update the values in your Rect Transform to retain the size and position of your newly-anchored UI element.

To fix this, simply set the **Left, Right, Top, and Bottom** values of your Rect Transform to **0**. These values represent a stretching offset, used to buffer a stretched object. When set to 0, there will be no buffer around a stretched element, which is normally what you want.

Remember: if a Stretched UI element is being scaled in a strange way, it's normally because Rect Transform contains unintended values.

Figure 4.16: The Left, Top, Right, and Bottom values let you specify an offset from each side. Setting them all to 0 will make an object stretch to the size of its parent.

With our black background scaling up to fill the entire canvas, we've suddenly lost visibility of our GameNameText object. This is because UI draw order is directly related to the hierarchy position. Objects lower in the hierarchy list will be drawn on top of objects higher in the list.

If we drag GameNameText from above the BlackBG to below it, it'll become visible again.

Figure 4.17: Drag UI elements in the Hierarchy window to manage their drawing order.

Our final UI element to add is a **Button** that will take us into the game.

Once again, right click on the canvas object and select **UI > Button - TextMeshPro**. Rename the new object from “Button” to “StartGameBtn”.

With this new Button still selected, go to the Rect Transform component and fill the following values.

- **Pos Y:** -60
- **Width:** 300
- **Height:** 50

Next, back in the Hierarchy window, click the arrow next to StartGameBtn to show its child objects. The Button prefab comes with a Text child that we can make use of. Select “Text (TMP)” and, in the TextMeshPro component, set it’s **Text Input** to “Start Game”.

Figure 4.18: The current Title Menu. Not much to look at, but it gets the job done.

With our objects placed, it’s time to make a component for our Title Menu.

Title Menu: Adding the Script

We’re at the point where the “Start Game” button needs to, you know, *start the game*.

To accomplish this, we are going to create a new script called **TitleMenu** to manage all the possible interactions that will happen on this screen.

Start by selecting the Canvas object. Because all UI elements are built from **GameObjects**, adding a new component is just like when we added scripts to our Player, Spikes and Bullets.

With the Canvas selected, go to the Inspector window and use **Add Component > New Script** to add a component called **TitleMenu**.

Figure 4.19: Adding components to UI Elements is like adding them to any other GameObject.

Once this new script has been opened in Visual Studio, we only have one function to add...

```
/// <summary>  
/// When the user presses the 'Start Game' button,  
/// we need to load the MainGame scene.  
/// </summary>  
  
public void onPressStartGameBtn()  
{  
    SceneManager.LoadScene("SampleScene");  
}
```

This will be the function we call when the button is pressed, which will load the actual game scene for the user to start playing.

Adding a Summary: By using the three slashes /// before a function, you can define a descriptive ‘Summary’ of that block of code. Visual Studio will then use that as a tooltip whenever we make use of this function. A very useful commenting tool for your ever-expanding codebase.

You’ve probably already noticed, but we seem to have angered the compiler again. The red line under **SceneManager** means we’re trying to use a system that isn’t defined.

Figure 4.20: The SceneManager isn’t defined. Luckily, there’s a quick way to fix that.

Whenever we make a script that needs to access the scene management system, we'll also need to put a **Using** statement at the top of the file. In the **TitleMenu** C# code, scroll to the top and add the line...

```
using UnityEngine.SceneManagement;
```

This lets the script know that we will be making use of all the functions available in `UnityEngine.SceneManagement`. Our compile error is now fixed, allowing us to move to our last step: connecting the button in our UI to our new **onPressStartGameBtn()** function.

Title Menu: Button Actions

Our prototype TitleMenu is almost done. We have UI elements in place, and a basic component hooked up for handling input.

Our final step is to make the “Start Game” button do what it was born to do: start the game.

If you’re still in Visual Studio, you’ll want to return to Unity and select the “StartGameBtn” button in our TitleMenu scene. The Inspector panel will show the **Button** component, which we’ll need to make use of.

Interactable	A Boolean parameter that indicates whether the button will receive input. If a button is not Interactable, it will automatically be set to its Disabled state.
Transition	How this button displays and transitions between various states (Rollover, Highlighted, Disabled, etc.). By default, buttons use the Color Tint transition, simply changing the color of the associated Image. The other Transition methods include Sprite Swap (changing the base Image) and Animation (which utilizes Unity’s animation system).

Navigation	The various ways that a button will handle navigation using the Keyboard arrow keys or Controller directional pad.
Visualize	Toggle visibility of the Navigation settings. Will display arrows showing the flow of one UI control to the next.
OnClick()	A list of Actions that will be called when this button is pressed.

Table 4.2: The parameters of the Button component.

As with most prefabs, we'll be keeping all these parameters set to their default *except for OnClick()*.

The **OnClick()** list is how we'll tell the Button what function on which to call when it's pressed. Right now, the first OnClick entry will be set to *No Function* and *None (Object)*. We need to tell the Button to call the **onPressStartGameBtn()** function in our **TitleMenu** script.

Do this by dragging the Canvas object from the Hierarchy window to where it says *None (Object)*.

Figure 4.21: Dragging to the target object of this OnClick() action.

Next, select the dropdown button that says *No Function*. This will show you all the functions that can be called from this button. Select **TitleMenu > onPressStart GameBtn**.

You've just set up your first **Action!** The Action system is another powerful way to write flexible systems in Unity, and we'll perform a deeper dive into it later.

For now, let's press the Play button to test our new Title Screen.

The screen comes up correctly. The black background fills our view and the button can be clicked on...but something's wrong. When we look in the Console, we see an error. *The game is mad that we haven't added our scenes to 'Build Settings'.*

Let's learn how to fix that.

Adding Scenes to the Build Settings

At the top left corner of the unity editor, select **File > Build Settings**. You'll see a menu that looks something like this...

Figure 4.22: This empty 'Scenes In Build' list is going to cause problems when loading Scenes.

We'll be learning more about build settings in the next chapter book, but for now we need to add our scenes to the Oak to be empty list at the top title "Scenes In Build". Any scenes that we reference in code Need to be added to this list.

There are few steps required, so let's add our scenes now...

1. Press the add open scenes button now to add our **TitleMenu** scene.
2. Close the build settings menu
3. In our project window, open the Assets > Scenes folder
4. Double click on SampleScene to open it
5. If Unity asks you to save changes to TitleMenu, make sure you press **SAVE**
6. Press **Ctrl + Shift + B** to bring up the **Build Settings** menu.
7. Press [Add Open Scenes] to add the SampleScene to the list
8. Close the Build Settings menu
9. In the Project window, double-click the TitleMenu scene to open it again.

The project now knows that the **SampleScene** and **TitleMenu** scenes will be referenced in code.

Press the play button once again.

If you select the “Start Game” option now, you’ll see the scene change from our Title Menu to our Main Game scene! Our Title Screen, while simple, is now working as intended.

Now it’s time to use our newfound UI skills to improve the Main Game UI.

A Basic HUD

When making a game, one of the most important parts of your User Interface is the **Heads Up Display**, or **HUD**.

Used to display vital information to the user, a well-crafted HUD will ensure the player always knows what they need to know. A poorly made HUD will either lack the vital information, or show TOO MUCH data, overwhelming the player.

For our prototype, there are three main stats we want the player to know at all times...

- **Health:** How much health does the player object have?
- **Coins:** How many coins have been collected?
- **Lives:** How many times can the player die before it’s Game Over.

Let’s start our HUD by opening **SampleScene**.

Right-click in the Hierarchy window and select **UI > Canvas**. Remember: you always need a canvas when placing UI elements.

Like with the **TitleMenu** canvas, go to the **Canvas Scaler** component and change **UI Scale Mode** to **Scale With Screen Size**. Then set the **Reference Resolution** to **X: 1080** and **Y: 1920**.

With our Canvas ready, let's place the UI text for 'Current Health'. For the sake of organization, it's always smart to create **Container** objects for your icons and text to be nested within. This allows you to rearrange entire elements easily as you iterate on your UI layouts. And expect there to be much iteration and polish as you receive player feedback.

Let's create a container object for our the player's current health. In the Hierarchy window, right-click on the **Canvas** object and select **UI > Image**. With the image selected, do the following...

1. Rename the Image **CurHealthContainer**
2. In the **Rect Transform** component, set **Width: 250** and **Height: 50**
3. In the Image component, change the Color to 255, 0, 255, 100

Figure 4.23: We're going to set the container image to be AS PINK AS POSSIBLE. This obnoxious little rectangle has a very important purpose.

If we did everything right, we'll now have an obnoxiously pink rectangle in the middle of our canvas object. This specific RBG value is an old programming color called 'Magic Pink'. It's a temporary color used by developers that will be removed later. For now, it'll help us know exactly the bounds of our container rectangle.

With our ugly pink container image made, let's fill it with some text objects. Right-click on **CurHealthContainer** and select **UI > Text - TextMeshPro**. This will add a text object displaying "New Text" in a white font.

Select this new object and change its name from **Text(TMP)** to **CurHealthTitle**.

In the TextMeshPro component, set the following values...

1. Change **Text Input** to **Health**.
2. Change **Vertex Color** to black (0,0,0)
3. Change vertical **Alignment** to **Middle**.

Figure 4.24: Our current ‘Health’ text. Let’s make sure it’s properly Left-Justified.

You’ll notice that despite having a ‘Left Justified’ Alignment value, the text is still somewhat offset. That is because the actual object that the text is rendered has a Middle-Center Anchor. To ensure our text is *actually* Left Justified, we’re going to need to change that Anchor.

Click on the Anchor icon in the Rec Transform component (the two squares with a red cross in the middle). This brings up the Anchor Presets, which we’ve used in the title screen to set Stretching options, but here we’ll be setting **Anchor Position AND Pivot Point**.

Figure 4.25: Set the Left-Middle Anchor reset WHILE holding the Shift key.

What we want to do is Anchor this object to the left side of the parent object AND set the pivot point of the text object to the leftmost side. It’s a bit confusing, unfortunately, but you need both the Anchor and Pivot Point to be in the Left-Middle before the text will look properly Left Justified.

One last step, and that is to zero-out all the position values in the text’s Rect Transform. Remember, when you change the Anchor information, Unity will ‘update’ the position values so the object retains its current location - only now with the changed Anchor info. By the X and Y values to 0, we’ll finally get our left justified “Health” title.

Figure 4.26: Our Health text is now properly Left-Justified within its container parent.

We’ll now make our ‘Value’ text. This will be displayed as Current Health / Max Health (e.g. 5/10).

1. Duplicate the **CurHealthTitle** object with Ctrl+Shift+D
2. Rename this new Text object to **CurHealthValue**
3. Change **Text Input** to “10/10”.
4. Change horizontal **Alignment** to **Right**
5. Bring up the **Anchor** popup in Rect Transform
6. Hold SHIFT and select Right Middle
7. Set Pos X to 0

Once created, our updated Health display will resemble [Figure 4.27](#)

Figure 4.27: In the Game View window, your Current Health container should now look like this.

Most games have important information, such as Health, displayed in the Upper-Left corner. Let’s move that now by selecting the **CurHealthContainer** object and performing the following actions …

1. Open the Anchor popup
2. Hold SHIFT and select TOP-LEFT
3. Set Pos X to 10
4. Set Pos Y to -10

Also, since there is a bit too much space between the Title text and Value, let’s bring the **Width** of the container down to 220 and the **Height** to 40. You should notice that, as you resize the container, the text objects are snapping and repositioning relative to the side they’re aligned to. That’s the power of Anchors!

Now we’ll make the other two text containers for Coins and Lives:

1. Select the **CurHealthContainer** object and press Shift + Ctrl + D twice
2. Rename first duplicate object **CurCoinsContainer** and the subobjects **CurCoinsTitle** and **CurCoinsValue**
3. Set the **Text Input** for **CurCoinsTitle** to “Coins”
4. Set the **Text Input** for **CurCoinsValue** to 999

5. Rename second duplicate object **CurLivesContainer** and the subobjects **CurLivesTitle** and **CurLivesValue**
6. Set the **Text Input** for **CurLivesTitle** to “Lives”
7. Set the **Text Input** for **CurLivesValue** to 99
8. Select all the ‘Container’ objects and **Disable the Image** components on all of them (this will hide the Magic Pink rectangles we’ve been using for placement purposes)

Figure 4.28: The HUD with the UI elements hooked up.

The HUD Component

The last component we need to create in this Chapter is the Main Game HUD script.

Select the Canvas and use the **Add Component > New Script** option to add **MainGameHUD**. Because we’ll be setting display text, we’ll need to include the following **using** statement at the top of the file...

```
using TMPro;
```

Next, add these members...

```
[SerializeField, Tooltip("TMP object displaying our current health.")]
```

```
TextMeshProUGUI _healthValueText;
```

```
[SerializeField, Tooltip("TMP object displaying the # of collected coins.")]
```

```
TextMeshProUGUI _coinValueText;
```

```
[SerializeField, Tooltip("TMP object displaying lives remaining.")]
```

```
TextMeshProUGUI _livesValueText;  
  
[SerializeField, Tooltip("The Health Manager we're displaying  
data for.")]
```

```
HealthManager _healthManager;
```

Set the body of the **Update()** function to contain this code...

```
void Update()  
{  
  
    int curHealth =  
    Mathf.RoundToInt(_healthManager.GetHealthCur());  
  
    int maxHealth =  
    Mathf.RoundToInt(_healthManager.GetHealthMax());  
  
    _healthValueText.text = curHealth + "/" + maxHealth;  
  
    _coinValueText.text =  
    GameSessionManager.Instance.GetCoins().ToString();  
  
    _livesValueText.text =  
    GameSessionManager.Instance.GetLives().ToString();  
  
}
```

These lines of code take the values from our various systems and display that data to the player. One problem: the functions we're using don't exist yet, so let's add them.

In the **HealthManager** component, add these 'Get' functions...

```
public float GetHealthMax() { return _healthMax; }  
  
public float GetHealthCur() { return _healthCur; }
```

And in the GameSessionManager component, add these ...

```
public int GetCoins() { return PickUpItem.s_objectsCollected; }

public int GetLives() { return _playerLives; }
```

By setting these functions as **public**, any other script can now access these values. These are known as **Getter** functions, which simply give other systems a way to make use of data that's traditionally set as **private**.

Speaking of **public** vs. **private**, you should notice there's one more error we need to deal with...

Figure 4.29: The Objects Collected values we're trying to use is inaccessible (i.e., it's been set to **private**).

Since this is a temporary value, we're going to fix this by simply making the member **public**. Open the **PickUpItem** script, find where we define the Objects Collected member, and add the keyword **public** in front of it...

```
public static int s_objectsCollected = 0;
```

With the script ready, our last step is to assign objects to the member variables of MainGameHUD. Return to the UnityEditor, select the Canvas, and scroll down to the MainGameHUD component.

You'll see several parameters, all assigned to NONE. We need to fill these, otherwise our code will break.

Figure 4.30: Click the Small Circle in these entry fields to bring up a useful list of available objects to choose.

While we can drag objects into these parameters from the Hierarchy window, another option is to use the 'Object Picker' popup. Press one of the small circle icons to the right of the parameter field to bring up this list.

Make sure the **Scene** tab is selected, then choose the objects we want to assign...

1. **Health Value Text:** CurHealthText
2. **Coin Value Text:** CurCoinsText
3. **Lives Value Text:** CurLivesValue
4. **Health Manager:** PlayerObj_Sphere

Press **Play**, and you should now see proper values being displayed in our HUD. Collecting coins will update the ‘Coins’ count, and taking damage will change the health and eventually lose you lives.

However, once we die 3 times, we still have the **Soft Lock** that initiated the need for proper **Game Flow** and **UI** implementation.

We can’t let this stand.

Game Over

When the player dies, and the game is over, it’s vital that we communicate this with them. And now that we understand the basics of **Unity UI** and **Scene Management**, it’s time to hook up some **Game Over** text.

First, make sure you’re still in the SampleScene. Then right-click on the Canvas object and add a Text UI Element.

Rename this to **GameOverText** and select it. In its TextMeshPro component, set the following parameters...

- **Font Size:** 82
- **Font Style:** Bold
- **Text Input:** GAME OVER!
- **Vertex Color:** 236, 16, 16 (TV-Safe Red)

With our text made big and bold, we now have an issue with Text Wrapping. Specifically, the text is wrapping in such a way where it reads

“Gam E Ove R!”. Fix this by going to the Rect Transform component and setting **Width** to **600**.

And since we want this text hidden by default, let’s hide it by deselecting the toggle box next to its name.

Figure 4.31: Remember: Disabling an object is as easy as clicking this check mark toggle.

Now, return to Visual Studio and open the **GameSessionManager** script. Add two new members...

```
[SerializeField, Tooltip("Object to display when the game is over.")]
```

```
private GameObject _gameOverObj;
```

```
[SerializeField, Tooltip("Title Menu countdown after the game is over.")]
```

```
private float _returnToMenuCountdown = 0;
```

In the **onPlayerDeath()** function, add the following line when the player is out of lives...

```
_gameOverObj.SetActive(true);
```

```
_returnToMenuCountdown = 4;
```

In the **Update()** function, add a check that will decrement the countdown if necessary, and return to the TitleMenu when it reaches 0.

```
if (_returnToMenuCountdown > 0)
```

```
{
```

```
    _returnToMenuCountdown -= Time.deltaTime;
```

```
if (_returnToMenuCountdown < 0)  
  
SceneManager.LoadScene("TitleMenu");  
  
}
```

And since we're using the SceneManagement system, remember to add the corresponding **using** line to the top of the file...

```
using UnityEngine.SceneManagement;
```

With the code written, return to Unity and assign our 'Game Over' text object to our new **_gameOverObj** member. Select the **GameSessionManager**, and assign our **GameOverText** object to the **Game Over Obj** parameter.

Figure 4.32: You can either drag the object, or use 'Assign Object' popup via the circle button. It's completely up to you which method to use.

You can now press the Play button to put everything to the test. Once you've lost all your lives, the "Game Over!" text will pop up for several seconds, then return you to the Title Screen.

Figure 4.33: The Game may be Over - as well as this Chapter - but your game development adventures are just beginning.

Conclusion

We've learned a lot in this chapter, but when it comes to the **Unity UI**, we're only scratching the surface. There is a deep well of additional UI components that you can be making use of, but even with what we know, there's a lot we can do.

In fact, coming up next, we're going to use all our Unity knowledge to playtest, polish, and *perfect* our *prototype*.

Let's end these first few chapters with a *bang*, shall we?

Questions

1. What are some of the gameplay elements you'd show in a HUD (Heads Up Display)?
2. True or False: A Canvas is the required parent of any UI elements.
3. UI Anchors determine how UI elements are repositioned with the parent objects changes size. Why is this so important, especially on PC, Mac and Mobile?
4. True or False: UI draw order is directly related to the hierarchy position.
5. If a Stretched UI element is being scaled in a strange way, what is the most likely cause?
6. True or False: Scenes need to be added to the project's Build Settings before they can be loaded using the SceneManagement APIs.

Key Terms

- **UI:** User Interface – the text, buttons, and images used to present information and receive input from the player.
- **Game Flow:** The flow of screens, popups, and prompts that the player navigates as they play your game. You want the player to get to the core game loop in as few clicks as possible.
- **Screen Resolution:** The size of the screen a game is being played on. Unless you're making a game with a fixed resolution, your UI will need to work (as in *resize* and *reposition* properly) at various sizes.
- **Canvas:** The main object of the Unity UI system. All other UI elements need to be children of a Canvas object.
- **TextMeshPro:** The Text rendering system used by Unity to keep text crisp, no matter what size and resolution they're displayed at.
- **Image:** A UI element that displays a texture.
- **Button:** A UI element that can be pressed to trigger an Action.

- **Action:** An event that calls a function on a given object. Used for buttons to determine what functions to call, on what objects, when that button is pressed.
- **Rect Transform:** The transform data for UI objects. This is slightly different from traditional Transform data, since it has some additional data, like anchor positions and size information.
- **Anchors:** How UI objects snap to their parent object. Can also control the Stretching of objects when a parent object resizes. A critical component when making screens that work at multiple resolutions.
- **Scene Management:** The system that handles the Loading and Management of scenes at runtime.
- **HUD:** The Heads Up Display (HUD) is all the data shown to the player during the core gameplay loop. A HUD tends to show health, score, lives, and other vital gameplay information to the player.
- **TV-Safe Red:** A RGB value for Red (236, 16, 16) that doesn't blow out on Television screens.

CHAPTER 5

Mastering the Fundamentals

You could say that creating a video game requires the mastery of several disparate skill sets. That, however, would surely be an understatement. No other industry requires the *interdisciplinary talent stack* that Video Games rely on.

In this chapter, we're going to cover several fundamental skills required to make a great game. We'll be starting with a discussion on **Project Management** - something easy to overlook when working with a small team (and especially if going solo). We'll also be diving into **Level Design**, **Playtesting**, gathering **Critical Feedback**, and **Iteration**, where we use that feedback as a roadmap.

And rest assured, we'll make lots of things **Explode**.

You can't talk about game *development fundamentals* without *explosions*.

Structure

- The Importance of Milestones
- Playtesting and Critical Feedback
- Mastering the Camera
- Interesting Levels
- The Transform Tools
- Varied Gameplay Mechanics
- Better Baddies
- Bigger Booms
- Permanence in a Game World
- Hunting for Bugs

- Making ‘The Build’

Objective

As we wrap up these initial chapters, it's important to take a step back and cover the broad skills required to iterate on your game. Only through constant feedback and focused iteration, and this chapter will cover the basics of this vital cycle.

By the end, you'll have an understanding of gathering feedback, level design, fixing bugs, and creating a build of your game to for other players to enjoy.

There's a lot of ground to cover, so let's get started!

The Importance of Milestones

Of all the skills it takes to bring a video game to completion, the most underrated is **Project Management**. And it's completely understandable - nobody ever finished playing a game and thought, ‘I love how that game was managed’. It’s the visuals, music, gameplay and story that stick with us, so it's only natural that - when we have an idea for a game - we focus on these *impactful* aspects of the creation process.

But that's not to say *management* isn't equally impactful. In fact, without solid project management, it's possible for even veteran game teams to lose control of the process.

And if there's one management tool worth mastering, it's the practice of setting and hitting development **Milestones**.

A milestone is a scheduled point in development where you want to have a certain number of features completed. You can break your overall project into multiple milestones, which lets you break apart the *mammoth* task of game development, letting you tackle the project step by step.

We'll be treating in this chapter as your first major Milestone, where we playtest, gather feedback, iterate on that feedback, and eventually finalize a build that anyone can play.

Getting Critical

One fundamental skill required when creating your own game is critiquing that game. **Playtesting** to gain **Critical Feedback** - documenting the flaws of your game - is a critical step towards making something great.

This is a task, however, that is more difficult than you'd realize. It's tough to focus on flaws when your game isn't even complete. When there's a long to-do list ahead of you, stopping to critique an unfinished project can feel like a waste of time.

This is why we plan out our Milestones. Make it a priority to pause development and step away from your day-to-day tasks. This lets you see the project as a whole. When you're in the thick of coding systems, building levels, and tweaking numbers, it can be easy to lose sight of the bigger picture.

Taking the time to gather **Critical Feedback** ensures the game you release to players is the game you've imagined in your head.

So, with the goal of listing all the flaws in our demo game, let's open the **SampleScene** and press the **Play** button ([Figure 5.1](#)).

Figure 5.1: Collect some coins. Shoot some spikes. Die a few times.

Since bugs take priority during a Milestone playtesting session, we start with the question: "**What bugs can you find?**"

- **Bug:** The game soft-locks if you fall off the edge of the playing field.
- **Bug:** Sometimes bullets will move very slowly (if at all).
- **Bug:** Bullets never get destroyed. They'll keep moving in a direction forever.

Good - our code has some bugs to fix. Bugs should never come as a surprise. Even the most skilled developers introduce them to a project.

Now let's restart the session with gameplay improvements in mind: "**How does gameplay feel?**"

- **Gameplay:** Level is boring. A square isn't fun to explore.
- **Gameplay:** Static camera doesn't work for a 3rd-person game.
- **Gameplay:** There's no reason to jump. Why have a feature that doesn't have a payoff?

- **Gameplay:** ‘One shot’ spikes are lame.
- **Gameplay:** Static, unmoving enemies are lame.
- **Gameplay:** Completely Random coin placement is lame.
- **Gameplay:** Is there a way to win? If not - that’s lame.
- **Gameplay:** Sphere has no character. I want a hero with personality.
- **Gameplay:** Hate the White-Boxed levels. Needs better looking environment art.

Yikes! That’s quite a list.

It may hurt to get these faults listed, but it should also feel *empowering*. Own these issues, and prioritize them before moving onto the next milestone. You now have a road map to improve your game - ***use it!***

Also note that there are items that can be ignored. The last two, dealing with the look and feel of the Hero and Environment, are known issues. Remember, we’re still in our White-Boxing stage of development. For our Milestone, assume these are **Working as Designed** (or WAD). We’ll deal with them in a later chapter, but the other issues - we should consider fixing.

Let’s see how we can make our game better.

Mastering the Camera

It’s a completely valid criticism to say that our camera *stinks*. We’re prototyping a 3rd-person game, so one of the first systems we need to perfect is our **Camera**. And as of now, it’s far from perfect.

Before we can improve our game’s camera requires a better understanding of how this fundamental system of Unity actually works.

Every scene requires at least one **Main Camera** object. The placement and parameters of this camera are used to render the game to the player’s screen. Without it, there will be *literally* nothing to look at, as shown (or *not shown*) in [figure 5.2](#)

Figure 5.2: If your Game window ever looks like this, then you know you’re missing an active camera object.

1. Let's select the **Main Camera** in our game ([Figure 5.3](#)). It's currently tilted down to look down the playing field. From this angle, the user can see the player sphere, coins, and hazards present in the current level.

Figure 5.3: The Main Camera object.

Because the Camera is an extended GameObject, it gets its Position, Rotation, and Scale data from its Transform component. The actual rendering is handled by the **Camera** component, which holds several important parameters. Let's look at the values worth learning in [table 5.1...](#)

Clear Flags	Various ways to clear the background of your rendered scene. Skybox is the default, which displays the selected Skybox image (as set in the Lighting window). You can also clear to a Solid Color , Depth Only , or Don't Clear . We'll rarely switch from the default..
Projection	There are two projection modes. Perspective will render objects as they're normally seen: drawn towards a vanishing point, with closer objects looking bigger and distant objects looking smaller. Orthographic looks less natural, with all lines running parallel to one another and depth having no impact on the size of objects.
Clipping Plane - Near	The point nearest the camera that objects stop being drawn
Clipping Plane - Far	The point furthest from the camera that objects stop being drawn. This is very useful (to do)

Table 5.1: The main Camera Parameters.

If you look through the Camera component, you'll notice other options that we didn't dive into. Many of these will never need to be touched, so we'll only talk about those if we need them in a later chapter.

2. Back to our improved Camera, let's add a new component to the **Main Camera** called **Follow Cam**. Just like when we add a component to any other GameObject, select the Main Camera object, then in the Inspector window select **Add Component > New Script**, and call it "FollowCam".

Open the new script in Visual Studio and add the following members...

```
[SerializeField, Tooltip("The object to follow.")]  
private GameObject _camTarget;  
  
[SerializeField, Tooltip("Target offset.")]  
private Vector3 _targetOffset;  
  
[SerializeField, Tooltip("The height off the ground to  
follow from.")]  
private float _camHeight = 9;  
  
[SerializeField, Tooltip("The distance from the target to  
follow from.")]  
private float _camDistance = -16;
```

3. Then add this as your **Update()** function...

```
// make sure we have a valid target  
if (!_camTarget)  
    return;  
// use the target object and offsets to calculate our  
target position  
Vector3 targetPos = _camTarget.transform.position;  
targetPos += _targetOffset;  
targetPos.y += _camHeight;  
targetPos.z += _camDistance;  
  
// move camera towards target position  
Vector3 camPos = transform.position;  
transform.position = Vector3.Lerp(camPos, targetPos,  
Time.deltaTime * 5.0f );
```

With the class written, and ready to follow its target object, return to Unity. In the Inspector window, you'll want to set the Cam Target, Cam Height, and Cam Distance values to resemble those in [figure 5.4](#):

Figure 5.4: Set your *FollowCam* component to target the *Player Sphere*, and you should be ready to test!

Leveling Up

With the camera feeling more polished and responsive, it's time to look at the other major issue from playtesting: poor **Level Design**.

Our feedback contains complaints like "Boring", "Lack of Exploration", and "Bad Item Placement". These all point to a poorly made level. And level design doesn't get much worse than 'it's a big square'.

Before we start improving our level, it's good practice to get our Scene View organized and ready for building. There are two steps you'll want to take: switching to **Orthographic View** and creating our level's **Building Blocks**.

1. Change your view from Perspective to Orthographic by pressing the small grey cube in the upper right corner of your scene window ([figure 5.5](#))

Figure 5.5: Toggle between Perspective and Orthographic view by pressing the Cube Icon in the Top-Right corner of the Scene View window.

2. Switching to **Orthographic** view makes it considerably easier to judge the position and orientation of objects in 3D space. Our other prep-work is to create a handful of **Building Blocks**: shapes we can easily duplicate to quickly prototype our level.
3. The first shapes will be variants of the existing Ground cube. Select the ground object, duplicate it, and rename it **LevelObj_Bridge**. Set the following Transform values...

Table 5.2: Transform values for LevelObj_Bridge.

4. Now duplicate the Bridge, rename it to **LevelObj_Ramp**, and set these Transform values...

Table 5.3: Transform values for LevelObj_Ramp.

5. Last, right-click in the Hierarchy window and select **3D Object > Cylinder**. Rename the new object to **LevelObj_Cylinder** and set its Transform...

Table 5.4: Transform values for LevelObj_Cylinder.

6. If you look further down the Inspector window, you'll notice that the default Cylinder object seems to come with a 'Capsule Collider'. This isn't the shape we want to use for collision, so right-click the Capsule Collider component and select **Remove Component**. We'll instead want to add a **Mesh Collider** component, which will ensure the collision bounds of the Cylinder matches its shape.

Figure 5.6: The 'Building Blocks' for our updated level.

7. Your last step: create an empty GameObject to place all these objects into. Call the empty game object **Level1Root** and drag all the **Building Blocks** so they're children of Level1Root.

We've placed these pieces close enough together where you can easily test them out by jumping between them. Press **Play** and try to jump to the Cylinder.

Now let's take a moment to learn a few tools we'll be using when designing our level.

Tools of the Trade

Up until this point, whenever we've set the **Position**, **Rotation**, or **Scale** of an object, it's always by entering values directly into the Transform panel. This is fine when we know specifics, but far too slow for building levels, where quick iteration and rapid playtesting is required.

To assist us, let's learn about the **Tools** available for manipulating GameObjects in our scene, which are listed in the toolbar, shown in [figure 5.7](#).

Figure 5.7: The Toolbar, containing all the tools you'll need to move, scale, and resize objects.

Located in the upper left corner of the Scene View window, the **Toolbar** gives you quick access to several tools that we'll be making use of.

Table 5.5: The object-manipulation tools that you'll be using in the Unity Editor.

Mastering Hotkeys: One of the most important skills to master (across any editing software) are Hotkeys. Each of the above tools has a corresponding hotkey mapped to the QWERTY row of keys. As you design your level, make sure you're using your left hand to switch between tools quickly. The faster you can manipulate objects in the scene, the faster you can iterate, and the more playtesting you can do.

With a better understanding of our Tools, and with our Building Blocks ready to go, it's finally time to get creative.

Let's design our first level!

Designing our First Level

It's officially time to get serious about level design. Before starting, make sure your Scene View window is in **Orthographic** mode, and verify all your **Building Blocks** are properly nested within the **Level1Root** object ([figure 5.8](#)). If all these check out, then it's time to start building!

Figure 5.8: Nesting Building Block objects will let you collapse / organize a Hierarchy Window that's about to get VERY long.

At this point in the process, it's less about following a list of rules and more about having fun. the more you're enjoying yourself, the more likely you are to come up with unique creative ideas. feel free to take inspiration from games you played, or perhaps hikes you've enjoyed, or even movies you've watched. When it comes to getting creative, there's no one-size-fits-all solution.

Figure 5.9: A few example levels. Your level can be as simple or complex as you want..just make sure to have fun building it!

While creativity is a muscle you'll have to exercise on your own, there are a few level design tips in unity that you can make use of ([table 5.6](#)).

Table 5.6: Several Level Design rules that we can make use of.

By the time you're done, you should have a level that deals with many of the criticisms that came out of our critical feedback session. You have a dynamically moving camera, a level worth exploring, gaps worth jumping over, and gameplay that feels closer to something you'd see in a released video game.

One issue we haven't resolved however, are the static hazards that don't present much of a challenge. Let's see what we can do about that.

Better Baddies

Static spikes helped us Implement our health system, but in terms of an interesting enemy they're fairly lacking. However, we can use the prefab system to do some cool things with our basic hazard.

To improve our spikes, for in 2 the object spawning code that we originally wrote 4 coin placement. There's a lot of potential in how we spawn objects, so let's tap into some of that potential now.

1. First we're going to add a few members they can be used to specify how these spawned objects move.

```
[SerializeField, Tooltip("How objects be organized when
spawned?")]
private SpawnShape _spawnShape;
private enum SpawnShape
{
    Random,
    Circle,
    Grid,
    Count
}
[SerializeField, Tooltip("Speed that this group of objects
will rotate.")]
private Vector3 _rotationSpeed;
```

2. The spawning component now takes a shape option, we'll use to position the spawned objects in interesting ways. The **Circle** option is the easiest to implement, so let's make sure our **Start()** function looks like this...

```
void Start()
{
    // instantiate the objects according to the spawn shape
    // parameter
    if (_spawnShape == SpawnShape.Circle)
    {
        SpawnObjectsInCircle();
    }
    else
    {
        for (int i = 0; i < _itemCount; i++)
            SpawnItemAtRandomPosition();
    }
}
```

3. Then, add the **SpawnObjectsInCircle()** function...

```
/// <summary>
/// Go through all the objects and spawn them in a circle.
/// Radius is determined by the size of the spawn zone
/// collider.
/// </summary>
void SpawnObjectsInCircle()
{
    float radius = _spawnZone.bounds.size.x / 2;
    Transform parent = this.gameObject.transform;
    for (int i = 0; i < _itemCount; i++)
    {
        // get the position on the circle to spawn this object
        float angle = i * Mathf.PI * 2 / _itemCount;
        Vector3 pos = Vector3.zero();
        pos.x = Mathf.Cos(angle);
        pos.z = Mathf.Sin(angle);
        pos *= radius;
        pos += _spawnZone.bounds.center;
```

```

        // spawn as a child of the parent object
        GameObject newObj = Instantiate(_itemToSpawn, parent);
        newObj.transform.localPosition = pos;
    }
}

```

4. Last, make sure the **Update()** function performs the rotation of the parent object...

```

void Update()
{
    // calculate the new rotation
    // by taking the old rotation
    // and applying the speed parameter
    Vector3 newRot = transform.localEulerAngles;
    newRot += _rotationSpeed * Time.deltaTime;
    transform.localEulerAngles = newRot;
}

```

Figure 5.10: We can now have a circular, spinning group of Spikes (or Coins, if we'd like)!

Explosive Feedback

Now that our Spikes are putting up a fight, we need to spend some time on the **Visual Effects** (VFX) used when the player attacks. Since it's pretty awkward to simply have the spikes disappear when destroyed, let's implement something a bit more exciting: an **Explosion** effect.

Start by making two small ‘chunks’ to use as our explosion geometry. When the health manager reaches 0, it’ll spawn a bunch of these objects and send them flying outwards.

Both should be Cubes - a larger one with the material ‘matEnemyRed’ and a smaller one that uses ‘matWhite’. Keep their scales around the 0.2 range.

Figure 5.11: Our chunks.

You’ll also want to add a **RigidBody** component to both of these. Remember that a RigidBody handles the **Physics** logic, which is a critical

piece of a *satisfying explosion!*

Once they have all the necessary components and are colored and scaled properly, it's time to convert them to Prefabs. Drag the 'chunk' objects, one at a time, into the **Assets / Prefabs** folder, at which point you can delete them from the scene.

Staying in this folder, find your **EnemyObj_Spikes** prefab and double-click on it. This will open it in **Edit Prefab Mode**: a way to isolate and safely make adjustments to your prefabs. In the inspector window, Press Add Component > New Script and add a script called **VFXHandler**.

Figure 5.12: The Spike in 'Edit Prefab' mode. Note the Hierarchy window, which is now much shorter and has the name of the prefab at the top.

Open the **VFXHandler** component in Visual Studio and add the following code...

```
[SerializeField, Tooltip("Prefab to spawn when hit and destroyed.")]  
GameObject _mainExplosionChunk;  
  
[SerializeField, Tooltip("Less common prefab when hit and destroyed.")]  
GameObject _secondaryExplosionChunk;  
  
[SerializeField, Tooltip("Min explosion chunks to spawn.")]  
int _minChunks = 10;  
  
[SerializeField, Tooltip("Max amount to spawn.")]  
int _maxChunks = 20;  
  
[SerializeField, Tooltip("Force of explosion.")]  
float _explosionForce = 1500;  
public void SpawnExplosion()  
{  
    // spawn a random number of the main chunks  
    int rand = Random.Range(_minChunks, _maxChunks);  
    if (_mainExplosionChunk)  
        for ( int i = 0; i < rand; i++ )  
            SpawnSubObject( _mainExplosionChunk );
```

```

// now spawn the secondary object
// (but only half the amount)
rand /= 2;
if (_secondaryExplosionChunk)
    for (int i = 0; i < rand; i++)
        SpawnSubObject(_secondaryExplosionChunk);
}
void SpawnSubObject( GameObject prefab )
{
    // get a random point around our object
    // should prevent collision with parent
    Vector3 pos = transform.position;
    pos += Random.onUnitSphere * 0.8f;
    GameObject newObj = Instantiate(prefab, pos,
        Quaternion.identity);
    // give the chunk a random velocity
    Rigidbody rb = newObj.GetComponent<Rigidbody>();
    rb?.AddExplosionForce(_explosionForce, transform.position,
        1f);
}

```

The VFX Handler component takes two prefab objects, a handful of values for spawning, then creates a random number of objects, all with a randomized Explosion Force, where the Spike object used to be. Now we just need something to *trigger* the explosion.

To call our explosion spawning function, open the **HealthManager** component and add the following lines under the **IsDead** check...

```

if (_isDead)
{
    VFXHandler vfx = GetComponent<VFXHandler>();
    vfx?.SpawnExplosion();
}

```

Here we simply grab any VFX component that may be attached to the dying object. If that component is valid, we call the public **SpawnExplosion()** function.

The Question Mark: Note the ? symbol where we called `rb?.AddExplosionForce()` and `vfx?.SpawnExplosion()`. This says “Only call this function if a component was found.” Using a question

*mark in this way is shorthand for a **if (null)** check (which prevents run-time errors).*

Our final step is to make sure the VFXHandler component has its data properly set. Return to the Unity Editor and open the **EnemyObj_Spikes** object in **Edit Prefab** mode. Drag the RedChunk and WhiteChunk prefabs into the Main and Secondary parameters. Make sure the Min and Max range is 20-40, and give the explosion a force of 1500 units.

Figure 5.13: All the data needed for an epic explosion!

With the code in place and prefabs assigned to the VFX Component, it's time to *test* our explosions. Place a handful of spikes around your level and use the [Enter] key to shoot at them. A single bullet is all it takes to destroy a Spike, at which point you should see a spray of red and white cubes where the spike used to be.

It's a simple effect, but also a *convincing* one. By ensuring the colors in our explosion are the same as the colors of the spike, it really looks like our hazard was shattered into smaller pieces.

Figure 5.14: The satisfying aftermath of a successful attack. The spikes never stood a chance.

Game Design 101: Permanence

An important game design concept is the idea of **Permanence**: a game world that changes as you play.

In our last screenshot, you saw the floor covered in small red chunks. This is an organic way to tell the story of your battle against the Spike hazards. By keeping those objects on the ground, the player is reminded of what happened previously in the game.

You can also build permanence into a game design by letting the player *build* objects in the game world. Whether you're simply placing blocks (Minecraft), clearing garbage (Viva Pinata), or constructing an entire Village (Animal Crossing), it's always good game design to give the players multiple ways to leave their thumbprint on the game world.

Hunting For Bugs

As we playtested our game, we found several bugs we should fix before finalizing this milestone build.

The Endless Pit

So, as of right now, if you jump off the edge of a platform, you'll fall for eternity. This creates a Soft-Lock state, requiring the player to restart the game to continue. And since 'lock states' are considered one of the worst errors we can encounter, let's deal with this bug first.

1. Bring up your **HealthManager**, and add this code to the **Update()** function...

```
// insta-death when we're in an endless pit
float yBounds = -25f;
if (transform.position.y < yBounds)
    _isDead = true;
```

Then, let's make sure the re-spawn code resets our downward velocity (which prevents a possible issue with a respawned player clipping through the starting location geometry).

2. Open the **GameSessionManager** script, and in the **onPlayerDeath()** function, add this code where the player gets respawned (around line 74)...

```
// clear the velocity of this object
Rigidbody rb = player.transform.GetComponent<Rigidbody>();
if (rb)
    rb.velocity = Vector3.zero;
```

You can now jump off the edge of your level with confidence in a proper respawn after falling too far.

Buggy Bullets

One issue you probably experienced when attacking the spikes was general bugginess in bullet speed. They currently have a tendency to move at different speeds, mostly due to how we're storing the 'facing' value.

1. Let's clean up the input code in our **PlayerController** script. Go to the `Update()` function, and update the input `if()` checks to look like this...

```
// check to see if any of the keyboard arrows are being
pressed
// if so, adjust the speed of the player
// also store the facing based on the keys being pressed
if (Input.GetKey(KeyCode.RightArrow))
{
    curSpeed.x += (_movementAcceleration * Time.deltaTime);
    _curFacing.x = 1;
    _curFacing.z = 0;
}
if (Input.GetKey(KeyCode.LeftArrow))
{
    curSpeed.x -= (_movementAcceleration * Time.deltaTime);
    _curFacing.x = -1;
    _curFacing.z = 0;
}
if (Input.GetKey(KeyCode.UpArrow))
{
    curSpeed.z += (_movementAcceleration * Time.deltaTime);
    _curFacing.z = 1;
    _curFacing.x = 0;
}
if (Input.GetKey(KeyCode.DownArrow))
{
    curSpeed.z -= (_movementAcceleration * Time.deltaTime);
    _curFacing.z = -1;
    _curFacing.x = 0;
}
```

You'll notice that we're now explicitly setting the facing to be 1, -1 or 0 in the various axes. This ensures we are facing the last direction we got input for.

2. With the current facing being set above, you can delete this code, which sits right after those `if()` statements...

```
// store the current facing
// do this after speed is adjusted by arrow keys
```

```
// but before friction is applied  
if (curSpeed.x != 0 && curSpeed.z != 0)  
    _curFacing = curSpeed.normalized;
```

3. Our last step to fix our bullet bugs is to replace our declaration of the `_curFacing` member. At the top of the file, tweak the having variable to be facing ‘right’ by default.

```
Vector3 _curFacing = new Vector3(1, 0, 0);
```

4. Press play, and fire a few bullets to see them move as you’d expect, with no instances of them moving slow (or not at all).

Bullets Forever

It’s always a good habit to fix the most egregious bugs first. This means that, if you don’t have time to fix everything, at least the ‘showstoppers’ are taken care of.

With that in mind, our last bug is really more ‘sloppy’ than ‘buggy’. When a bullet is fired into the distance, it will live forever if it never hits something. This is more of a performance concern than anything - too many bullets being updated off-screen can slow down your game.

1. To fix this, you’ll want to double-click your **BulletObj_Basic** prefab, to open it in **Edit Prefab** mode. In the inspector window, select **Add Component** > **New Script**, and name the new script **SelfDestructTimer**.
2. The code for this new component is nice and simple...

```
[SerializeField, Tooltip("Seconds until this object self-  
destructs")]  
float _countdownTimer = 1.5f;  
void Update()  
{  
    _countdownTimer -= Time.deltaTime;  
    if (_countdownTimer <= 0)  
        Destroy(gameObject);  
}
```

3. Close the Prefab Editor by pressing the little arrow in the upper left of the Hierarchy window, and Save the changes, if Unity asks you to.

Now press play and use [Enter] to fire some bullets. You'll notice they now disappear before hitting the side of the screen.

4. And with that, our bugs are fixed and gameplay improvements are implemented. It's now time to make **The Build**.

Exporting ‘The Build’

We've reached the very last step in a milestone: making **The Build**.

A ‘Build’ is a standalone version of your game that anyone with the proper hardware can play. While Unity makes it easy to playtest directly in the editor, all serious Quality Assurance (QA) testing should be done using a standalone Build.

When we're talking about **The Build**, however, we're referring to a standalone version of the game that is tied *directly to a milestone*. This is a version of the game that you've been developing for weeks, months, or possibly years. **The Build** is often the punctuation to a lot of effort, and will always come with a sigh of relief.

With all your hard work from [Chapters 1-5](#) complete, it's officially time to make **The Build** for this milestone. Make sure your scene is saved, and from select File > Build Settings (or use the hotkey combo [Ctrl] + [Shift] + [B]).

Figure 5.15: You should recognize this window from [Chapter 4](#), where we added our scenes to the ‘Scenes in Build’ list.

This window has several options that we should understand before moving forward. Make sure the ‘Windows, Mac, Linux’ option is selected under **Platform**, then let's dig into these parameters...

Scenes in Build	A list of all the scenes that will be included in this Build. Remember that, if we ever want to load a scene from code (using the Scene Management system) that scene needs to be included in this list.
Platform	These are all the platforms that you can export to.
Asset Import Overrides	Some texture override settings, if you ever want to set a limit on size, or force a mode of compression.
Target Platform	Which OS you're targeting with this standalone build (Windows, Mac, or Linux).

Architecture	Do you want to make a 32-bit or 64-bit build?
---------------------	---

Table 5.7: Build parameters and settings.

All the other options listed are related to Debugging your Build, which we'll save for another Chapter.

There are also two buttons in the bottom right corner, **Build** and **Build And Run**. The first will simply create the build, while the second option will also run it once completed.

Select the **Build and Run** option, then navigate to your Projects folder, where you should create a new “Builds” folder (for organizational purposes). Within the new builds folder, make one more folder called “UnityMilestone1”. With this new folder selected, press the **Select Folder** button.

Figure 5.16: Create a Builds folder to store all the standalone files in.

Unity will take over from here, taking your projects files and making sure they're compatible with your platform of choice. When it's done, it should automatically start the game.

Figure 5.17: Your game - officially playable as a standalone Build!

Conclusion

You've reached the end of your first milestone - a huge *congratulations!*

If you remember at the beginning of the chapter, we stressed the importance of prioritizing **Milestones** as a way to reflect on your progress. Not only does it force you to play and enjoy your game, it also gives you a moment to appreciate how far you've come.

In the last five chapters, you've been able to master the basics of Unity. You've learned and utilized the fundamental systems of game creation: the Editor, adding components, spawning prefabs, building a UI, managing scenes. You've also used these tools to build a Player object, Health Management system, basic weapons and enemies, pickup items, and even your first level.

And while we've accomplished much, this milestone lets us also look towards the future. We still have some feedback items that need to be dealt with: from the lack of a 'Victory' condition to the *underwhelming appearance* of our environment and assets.

There's still much to learn, and more to do, but please take this moment to congratulate yourself on everything that's been accomplished.

You've *mastered the fundamentals* - now onto more advanced challenges!

Questions

1. Why is it difficult to give critical feedback about your own project?
Why is it important to have external testers providing feedback?
2. Why is it acceptable to mark unfinished aspects of your game a 'Working as Designed', even if they're unfinished?
3. True or False: Getting a long list of critical (possibly painful) feedback empowers you to make your game better.
4. When designing a level, what camera mode should you be in: Perspective or Orthographic?
5. True or False: Coins are a great way to draw the player down a specific path.

Key Terms

- **Milestones:** A stopping point in your project where you playtest, bug-fix, and polish the game and its current state. A great opportunity to take a step back from day-to-day tasks and critique the game as a whole.
- **Critical Feedback:** After play testing, you should have a list of critical feedback items gathered from that session. This list will include bugs and gameplay issues that should be prioritized and fixed.
- **Working As Designed (WAD):** It's always good to have a clear understanding of how a gameplay system should be working. As feedback comes in, take note of whether a complaint is, or isn't, valid. When playtesters criticize something that hasn't been fully

implemented yet, you can categorize these issues as WAD, with the expectation that they'll be dealt with in a later milestone.

- **Level Design:** The process of building environments that users will enjoy. Should start off as a rough ‘White-Boxed’ prototype, then be finalized once considered fun in their simplified form. Remember, if it’s fun with simple graphics, it’ll only get better with fancy graphics.
- **Building Blocks:** Basic shapes that you will use to design a rough version of your level. Blocks, ramps, cylinders are all great basic forms to make use of when prototyping a level.
- **Hotkeys:** Keyboard presses that will quickly change the tools being used in the editor. Mastering hotkeys will result in a huge boost to the speed of your design output.
- **Visual Effects (VFX):** Objects and effects used to improve gameplay feedback, but not necessarily alter the gameplay itself, are considered Visual Effects (VFX). Explosions let the player know an enemy has been destroyed. Wind wisps and dust mote particles can communicate the direction and strength of wind. And a weapon’s particle effects can have varying intensity, helping to communicate the strength of an attack.
- **Permanence:** Making sure the world changes as the player advances through your game. This can be fire, smoke, and debris from an epic battle. It can also be a growing village that the player is helping to rebuild. Use every opportunity you can to ensure the player notices their actions affecting the game world.
- **The Build:** A standalone version of your game that others can play (without needing the Unity Editor). Should correspond with a milestone, and be archived so you can look back on older builds to appreciate the progress made.

CHAPTER 6

The Physics of Fun

With Milestone #1 behind us, and the fundamentals of Unity under our belt, it's time to plan ahead to Milestone #2. What gameplay systems do we want to learn and implement? What Unity tools do we want to dive into? And what do we want our demo to look like by the end of this section?

While enemies, weapons, sound and animation all need to be discussed, our next big topic is going to be Unity's **Physics System**. This is an aspect of Unity that we've already utilized (since Colliders and RigidBodies are physics components), but given its importance, learning more about this powerful system is necessary before moving forward.

So get ready to accelerate, decelerate, collide, push, and pivot: we're jumping into Physics!

Overview

Some of the concepts we'll cover in [Chapter 6](#) are...

- Physics Components
- Collider Shapes
- Rigidbody Components
- Joints
- Adjusting Physics using Code
- Pushable Blocks
- Seesaws
- Springboards

Objective

By the end of this chapter, you'll have a much stronger understanding of Unity's Physics system. You'll learn about the available physics components and put your new knowledge to the test, creating physics-based gameplay elements for your demo game.

The Unity Physics System

Any game that has moving objects requires some sort of physics system. The complexity of these can vary, but at their core, a physics system uses real-world concepts - mass, velocity, gravity, etc - to simulate movement and collision in your virtual world.

Like most systems which we'll be learning, the **Unity Physics System** is component-based: you add physics components to GameObject to define how they move and react in the environment.

You've already made use of several physics components in previous chapters. RigidBody components were added to objects that need to respond to the physics simulation. Colliders were used to define the shape of an object within the simulation.

And while those two components types were important, we've only scratched the surface of what Unity's Physics System has to offer.

Physics Components

The best way to learn about the wide variety of physics components is to put them to the test. Boxes that can be pushed. Windy areas that apply a force to the player. Springboards that vault the player skyward. There are many ways we can use physics to make our game more interesting, with each new game mechanic teaching us a new Physics sub-system.

Let's start a fresh scene called **PhysicsSandbox**, which will be a clean workspace for learning and implementing sample versions of all these components.

Basic Collider Shapes

Collider components represent shapes within the physics simulation. All the 3D objects we've placed so far (Cubes, Spheres, Cylinders) have come with a collider pre-attached to them. Broadly speaking, every object that has a

gameplay purpose (enemies, level objects, bullets, pickup items, etc) will need a collider component attached to it.

One Idea, Many Names: Collider shapes have various names across different engines: “Hitbox”, “Collision Object”, “Bounding Box”. These all refer to the same concept: invisible geometry that represents the physical shape of a GameObject. You cannot see a Collider, but this is what dictates when an object lands on the ground, gets hit by a bullet, or picks up an item. While they don’t need to be exact, it’s important that loosely match the shape of the object they’re representing.

There are three basic collider shapes that you use whenever possible: Boxes, Spheres, and Capsules, as shown in [Figure 6.1](#)

Figure 6.1: The Box Collider, Sphere Collider, and Capsule collider.

These shapes have been optimized for the physics system, resulting in quick calculations across thousands of objects. Each component type has a handful of common parameters, as well as a handful of unique values that will adjust the size and shape of these colliders...

All Colliders	
Edit Collider	Toggle visual gizmos that can change the size and position of a collider.
Is Trigger	Is this a physical collider (blocks movement of other objects) or a trigger (allows other objects to pass-through it)?
Material	The Physics ‘Material’ of this object (mass, friction, etc).
Center	The center point of this collider.
Box Collider	
Size	The X, Y, and Z sizes of the cube.
Sphere Collider	
Radius	The size of the sphere.
Capsule Collider	
Radius	The radius of the capsule.
Height	How long / tall / wide the capsule is, in the specified direction.
Direction	The direction that the capsule extends on (based on the Height)

parameter).

Table 6.1: Parameters for the various Collider components you'll be using.

Of course, there are times where you will need a collision shape that doesn't fall into these three categories. In those situations, you can make use of the **MeshCollider**.

Mesh Collider

The mesh collider component will create a collision shape based on the geometry of a mesh. And while this is a flexible option for oddly shaped objects, it's also an *expensive* option, requiring considerably more calculations to handle collisions.

Here are a few helpful tips when determining whether to use a mesh collider for a game object:

- Is your object roughly the shape of a box, sphere, or capsule? During fast and frantic gameplay, performance is more important than precision, so a large, easy-to-hit collider of any shape may be preferable.
- If your object is too oddly-shaped, can you make a collider out of multiple components? Since Unity objects can have multiple collider's added to them, you may want to construct your shape out of Cubes, Spheres, and Capsules. A desk, for instance, could be made out of 3 Cubes, which would be preferable to a complex MeshCollider shape.
- If you DO decide a mesh collider is needed, make sure you create a low-poly mesh to assign as your collider. This has to be done in your authoring tool, but it gets you the precision of a MeshCollider without all the performance overhead.

Rigidbody

The Rigidbody is a vital Unity component: attaching one to your object gives the Physics System full control over its velocity, placement, and collision handling. Any object that needs to respond to gravity, trigger collision functions, or otherwise interact with other physics objects will need a Rigid Body.

Let's look over the RigidBody parameters we have access to in [Table 6.2](#)

RigidBody	
Mass	The mass of an object. Lower mass objects are easy to move, whereas higher mass objects are harder to move.
Drag	Drag used to slow down the movement of an object.
Angular Drag	Used to slow down the rotation of an object.
Use Gravity	Is this object pushed downward by the physics system?
Is Kinematic	If true, this object will no longer be affected by forces, collisions, or joints. Will still trigger callbacks when the object is collided with.
Interpolate	Smooths out the movement of certain physics objects. Unity suggests that you enable this for the Player and disable it for everything else.
Collision Detection	How often, and in what manner, collisions for this object are calculated. Faster objects will need to use Continuous detection, whereas slower objects will be fine using Discrete .
Freeze Position	Constrain the position of this object on the X, Y, or Z axis.
Freeze Rotation	Constrain the rotation of this object on the X, Y, or Z axis.
Info	Read-only parameters that show a detailed rundown of the current physics calculations.

Table 6.2: Parameters of the RigidBody component.

Joints

Physics **Joints** are a special way to attach Rigidbody objects to one another. Joint objects will rotate and swing as parent and child objects are moved and rotated.

1. Let's create a joint object by adding a 3D cube to the scene. Name this Cube **RootJoint** and give it the following transform values as shown in [table 6.3](#)

Table 6.3: Transform data for the new RootJoint cube.

2. Once **RootJoint** has its proper location and rotation set, you'll want to add a **CharacterJoint** component to our cube. This tells the physics

system that the object should pivot and rotate when a force acts upon it - never moving too far from its pivot point.

3. Now duplicate this object 3 times with [Ctrl] + [Shift] + [D], and name the 3 new cubes **Joint_1**, **Joint_2**, and **Joint_3**. Each object should have their Y Position set to **-1.1 units** below the previous object, with their finalized position values matching those in [table 6.4](#)

Table 6.4: Position data for the 3 new Joint cubes.

4. You should now have a column of cubes, with the RootJoint object on the top, and Joints 1, 2, and 3 below it. If you were to click on the **Game** tab, your view should look like [figure 6.2](#)

Figure 6.2: Our chain of joint objects.

All we have to do now is connect the joints in order.

5. Select **Joint_1**, and in the Inspector window locate the **CharacterJoint** component. The second parameter is named **Connected Body**, which lets you assign a parent Rigidbody object to this joint. The following steps will properly connect your joint objects in the proper order...

1. Select **Joint_1** and set its **ConnectedBody** to be **RootJoint**.
2. Select **Joint_2** and set its **ConnectedBody** to be **Joint_1**.
3. Select **Joint_3** and set its **ConnectedBody** to be **Joint_2**.

All of the joint objects should have physics components as shown in [Figure 6.3](#).

Figure 6.3: The Joint object and Character Joint component.

6. Before we test there's one last parameter to set. Select the RootJoint and enable the **Is Kinematic** option. This tells the physics system to stop applying forces to this object, allowing the RootJoint to affect other physics objects - like its child joints - but not be affected itself.

With our objects created and component parameters set, it's time to test our chain of joints.

7. Press the **Play** button, go to the Scene view, and use the **Movement Tool** to drag around the RootJoint. As the RootJoint object changes position, you should notice all the other joints dragging behind it, much like a chain or rope.

Another way to test our joints is to place a sphere object in the scene. Give this new sphere object a rigid body component, enable **Is Kinematic**, and disable **Use Gravity**.

8. Press the **Play** button again, but this time drag around our new sphere. Notice that, as it collides with our chain of joint objects, it pushes them around as expected.

Figure 6.4: Our sphere object easily pushes around our chain of joint objects.

Tweaking Physics in Code

Unity's 'out of the box' Physics components will certainly speed up game creation, but some design ideas are bound to require custom scripts. In these situations, Unity provides access to all the functions and members needed to tweak physics objects directly through code.

Accessing Parameters: Need to change a value in code? Use the function GetComponent() to grab the script you want to modify, then use intellisense to search for the member you're looking for. If a parameter is displayed in a Component's Inspector panel, chances are it'll be accessible in code.

Much like our previous experience with rigid bodies and colliders, we've already used code to affect physics objects. In the **PlayerController** script, we take the input from the keyboard and adjust the velocity of our player object accordingly. Specifically, we adjust the player's velocity using the line...

```
_rigidBody.velocity = curSpeed;
```

This overrides the current velocity of our player with the curSpeed X, Y, and Z values, which we've tweaked based on input.

Now let's consider the gameplay concept of 'Wind'. By creating an area where a directional force is applied to objects, level designers get to push the player around an environment. Windy areas force the user to think fast before they fall off a cliff or get pushed into hazards - a simple but effective use of the physics system.

Start this new hazard by adding a 3D Capsule to our physics sandbox. Name it **WindZone** and set its transform component to match the values in [Table 6.5](#).

Table 6.5: The transform values of our new WindZone object.

This will place a large gray capsule to the right of our Joint Objects. Since this object represents an Area of Effect, and not a physical object, go to its Rigidbody component and set **Is Trigger**. This will ensure that the player can move through, and be properly affected by, our WindZone. You don't want the WindZone to be something you can jump onto.

Since this is a trigger area, which the player can now move *through*, let's create and apply a **Transparent Material** to it. This allows objects within the zone to still be visible - something players will expect.

In the Assets > Materials folder, create a new material called **matTransBlue**. In the inspector window, set its **Rendering Mode** to **Fade**, and set its **Albedo** color to light blue (0, 150, 255). Make sure to also set the alpha value to 100. By using a transparent color with the Fade rendering mode enabled, you're indicating that you want this material to show up as transparent. Once done, your material's parameters should match those in [Figure 6.5](#)

Figure 6.5: Our new Transparent Blue material. Make sure Rendering Mode is set to Fade and the Albedo color has an Alpha value set.

Once created, drag the matTransBlue material from the assets folder and directly onto the **WindZone** object. You should instantly see it take on this new transparent color, allowing objects behind it to retain some visibility.

Next, select the WindZone and add a new script named **PhysicsForceZone**. This will be the script that applies a change in velocity to any objects within the collider area.

Figure 6.6: Press Add Component > New Script and add a PhysicsForceZone component to our transparent WindZone object.

Open the script in Visual Studio and add a single member to the top of the script...

```
[SerializeField, Tooltip("Force applied to any hit RigidBody  
object.")]  
float _forceToApply = 1;
```

This parameter will be the amount of force applied to any objects within the Collision capsule.

Next, add the following awake function to the PhysicsForceZone script...

```
private void Awake()  
{  
    CapsuleCollider c = GetComponent<CapsuleCollider>();  
    if (c)  
    {  
        c.isTrigger = true;  
    }  
    Rigidbody rb = GetComponent<Rigidbody>();  
    if (rb)  
    {  
        rb.isKinematic = true;  
        rb.useGravity = false;  
    }  
}
```

While all these values could have been set from the Unity Editor, this **Awake()** function ensures the properties are set correctly in the off-chance that step was forgotten. In general, if you have a script that's going to be used across your project, it's good to make your code as 'mistake-proof' as possible.

The last bit of code we need to add is an **OnTriggerStay()** function, which gets called on every object within a trigger collider, for as long as that object remains within that Collider.

```
void OnTriggerStay(Collider collider)  
{
```

```

GameObject hitObj = collider.gameObject;
if ( hitObj != null )
{
    Rigidbody rb = hitObj.GetComponent<Rigidbody>();
    // get the direction of the Y axis
    Vector3 dir = transform.up;
    // apply directional force to this object
    rb.AddForce( dir * _forceToApply );
}
}

```

What this code does is take the object within the Collider, grab their rigid body components, and apply an amount of force in the direction the **WindZone** is facing (ie. the direction the Y-axis is pointing).

You can now press the **Play** button. Bring up the Scene view window, select the **WindZone** object, and press the **[W]** hotkey to switch to the Movement Tool. Drag the WindZone around until it intersects with the chain objects, where it will push the linked joints upwards ever-so-slightly. Now rotate your WindZone object so that it's blowing the chains in another Direction. Use the inspector window to increase the **Force to Apply** to **50**. Notice how the increased force causes the chain to be pushed even more, as in [Figure 6.7?](#)

Figure 6.7: Our WindZone applies physics force to the dangling chain.

We've successfully tested some of the Unity's most useful physics components - utilizing rigid bodies, collider shapes, joints, and applying forces.

With all that new knowledge, it's time to turn our physics systems into Gameplay systems.

Gameplay: Pushable Blocks

One of the more common gameplay tropes is the **Pushable Block**. The player is placed in an environment with a switch that needs to be pressed, or a ledge that's too high to reach. Somewhere in that environment is a block that has to be pushed to solve this simple puzzle.

Let's do a little level design work to create an area where a **Pushable Block** is needed.

1. Return to our **SimpleScene**. Find an open area of your level and make a long corridor. At one end, place a ledge that's just out of reach (and preferably filled with treasure). On the other end, create a new 3D Cube named **LevelObj_PushableCube**. Make it small enough to jump on, but large enough where It can be used to reach our Treasure-filled ledge.
2. Our Cube already has a collider component attached to it, so all we need to do here is add a RigidBody component. Select the Cube, and press **Add Component > Physics > RigidBody**.

Figure 6.8: Our Pushable Block. You may want to make it a unique color - like orange - to stand out as a unique environmental object.

3. You can now press **Play** to see how your hero will interact with the Cube's default physics parameters. You'll notice the object is really hard to push and far too unwieldy, making precision movement difficult.

Figure 6.9: The default RigidBody settings make a pushable block that's...not very pushable.

4. To fix this, we can make use of a series of Constraint parameters in Rigidbody called **Freeze Rotation**. With the cube selected, go to the RigidBody parameters and find the Freeze Rotation toggles. Enable these across all 3 axes (X, Y, Z).

Figure 6.10: Freezing the rotation will keep our cube from turning as we push it.

5. You should also reduce the **Mass** of the cube from **1** down to **0.5**. Once all these parameters are set, press the **Play** button. Pushing the cube should be much easier, and you should be able to get into a position that lets you jump onto it, then onto your previously unreachable ledge.

Figure 6.11: We now have a Pushable Block!

Some important notes about designing a level with Pushable Blocks in mind...

- Have a raised border of walkable space around the area that the block can be pushed within. If you allow blocks to be pushed all the way to the wall, you can have a situation where the player can't get behind it to push it further.
- If you design a level where the block can fall, or be destroyed, create a system that will respawn the block after a few seconds. You can't have a 'Pushable Block' puzzle without the block.
- Get creative with the object you're pushing. Is it a pushable set of stairs? Perhaps it's a cauldron of fire that has to be pushed under something flammable? Or perhaps you're pushing an object that needs to work as a shield against dangerous projectiles. Always be thinking of clever twists on common ideas.
- Now that we have our Pushable Block working, let's officially make it one of our Level Design prefabs. Drag the **LevelObj_PushableCube** object from the hierarchy window into the **Assets > Prefabs** folder.

And, with that, we've created our first physics-based gameplay element. Now let's move from pushable blocks to another common video game concept: **Moving Platforms**.

Gameplay: Seesaw Platform

Moving Platforms are another physics-based staple of video game design. By requiring precise jumps and quick reflexes, the **Moving Platform** is a great addition to our growing palette of level design elements.

For example, let's make an angled platform that raises on one end when lowered on the other. Specifically, let's make a **Seesaw**.

1. Start by making an empty object named **Seesaw**. This will only be used as a parent object for our physics objects, and not actually have physics components attached to it. Next, make a 3D Cylinder and name it **Seesaw_Pivot**. Add a **RigidBody** component to it, with the

parameter **Is Kinematic** activated, and set the scale to the values mentioned in [table 6.6](#)

Table 6.6: Scale values of the Seasaw Pivot object

2. Our last object is the plank. Make a 3D Cube, name it **Seesaw_Plank**, assign the matOrange material to it, and set the transform values to match those in [table 6.7](#)

Table 6.7: Transform values for the Seasaw Plank object.

3. Give **Seesaw_Plank** a **CharacterJoint** component. Set its **ConnectedBody** parameter to **Seesaw_Pivot**. Make sure both of these are children of the original **Seesaw** object.
4. Now build a section of the level where the SeeSaw platform spans across a gap, where at the end of the gap there's a ledge. You can use [Figure 6.12](#) as an example to build from.

Figure 6.12: Our current Moving Platform, complete with distant ledge. While not necessary, it's preferable to top this ledge with coins, because coins are awesome.

5. We now have the basic objects and components set up for our Seesaw. Press **Play** to test it.
6. Jumping on the plank it kind of works - the long cube certainly moves around when you collide with it. It's also fairly broken - tilting on all axes and, in general, not acting very 'Seesaw-like'.
7. This is as expected. When you set up a physics object, it often needs testing and tweaking to work as intended.
8. Press **Stop**. We need to first add some constraints to the plank. In the Rigidbody component, **Freeze Position** on all axes, and **Freeze Rotation** on the X and Z axis as shown in [figure 6.13](#)

Figure 6.13: The SeaSaw_Plank Rigidbody constraints should look like this.

9. Press **Play** again. Jumping onto the Seesaw should now feel better, but now you'll notice two major issues, which are mentioned below:

- The angle of the plank changes way too fast, with the platform swinging downward before you can respond.
- You can't jump off the seesaw! Literally, pressing [Spacebar] while standing on the plank does nothing.

10. The first problem is easy to fix. Go to the Rigidbody of Seesaw_Pivot and set the Mass to 1 and Angular Drag to 20. This will slow the tilting of the plank and make it feel better.

11. The second issue requires a slight rework of our Jump logic. Open PlayerController and find where we look for the Spacebar input (look for the comment "Does the player want to jump?").

Instead of looking at the current downwards velocity, we need to check a new value: **Is Grounded**. Checking if an object 'Is Grounded' is a common bit of logic you'll need to write across your GameObjects. There are many actions that simply can't be performed if in the air (jumping, crouching, opening doors, etc). These actions will need to make a similar check, but for now, we only need it for our **Jump** input.

12. Change the spacebar input **if()** statement to look like this...

```
if (Input.GetKeyDown(KeyCode.Space) && CalcIsGrounded())
```

13. Back at the top of the file, Add a **bool _isGrounded** member, as well as a member to store our collider component for quick access.

```
[SerializeField, Tooltip("Are we on the ground?")]
private bool _isGrounded = false;
[SerializeField, Tooltip("The player's main collision
shape.")]
Collider _myCollider = null;
```

14. Next, in **Start()**, right under where we grab the rigidbody component, also get and store the collider object...

```
_myCollider = GetComponent<Collider>();
```

15. Our last step is to add this function towards the bottom of the file....

```
/// <summary>
/// Check below the player object.
```

```

    /// If they're standing on a solid object, they can Jump
    /// and perform other actions not available in mid-air.
    /// </summary>
bool CalcIsGrounded()
{
    float offset = 0.1f;
    Vector3 pos = _myCollider.bounds.center;
    pos.y = _myCollider.bounds.min.y - offset;
    _isGrounded = Physics.CheckSphere(pos, offset);
    return _isGrounded;
}

```

The IsGrounded check simply looks below our collider (roughly where feet would be) to see if the physics system detects a collision. If so, it sets **_isGrounded** to **True**, which we can then use to allow or block the player from jumping in mid-air. Once these changes are made, you should be able to jump onto the Seasaw and have if function as seen in [Figure 6.14](#).

Figure 6.14: Hooray - we have a working seesaw!

Gameplay: Springboards

Another common physics-based object you may recognize is the **Springboard**: part of the environment that can be jumped onto, which then launches the player skyward.

Our Springboard is going to be a simple, flattened Cube. We'll place it on the ground so it's easy to step onto, then attach a script that bounces the player up into the air.

1. First, make the 3D Cube, then give it the following Scale values, as shown in [table 6.8](#)

Table 6.8: The Scale of our new Springboard cube.

2. Next, build a tall ledge, with the Springboard object at the bottom. You can also use a trail of coins to telegraph the purpose and trajectory of that Springboard as shown in [figure 6.15](#).

Figure 6.15: The Springboard - created, placed, and ready for its component.

3. Your last step is to select the Springboard object and use **Add Component > New Script** to attach a new **Springboard** component. Open this in Visual Studio and add the following code...

```
[SerializeField, Tooltip("Velocity change on the Y axis.")]
float _upwardsForce = 2000f;
private void OnCollisionEnter(Collision collision)
{
    GameObject hitObj = collision.gameObject;
    if (hitObj != null)
    {
        Rigidbody rb = hitObj.GetComponent<Rigidbody>();
        rb?.AddForce(0, _upwardsForce, 0);
    }
}
```

And that's it! The Springboard component lets you set a force that is applied to any object that steps on it.

4. Press **Play** and move your player onto our new object. Once they make contact, the **Springboard** script uses the `AddForce()` function to launch the player into the sky.
5. The last step is to drag our object into the **Assets > Prefabs** folder so we can use it whenever we're building a level.

Gameplay: Wind Zones Strike Again

At the beginning of this chapter, we created a new component that would apply a physical force to objects that step into a trigger collider - WindZones. Let's revisit this idea to create some blustery areas with timed wind gusts. Players will have to plan their movement carefully to avoid getting knocked off a platform.

1. Let's start by making a 3D cube named **WindZoneSpawner**. Set the scale of this object to the values shown in [table 6.9](#)

Table 6.9: The scale values of our WindZoneSpawner object.

2. Now make a 3D Capsule as a child of the WindZoneSpawner. Create and assign a new Black material to the capsule, and set its transform data to values indicated in [table 6.10](#)

Table 6.10: The transform values of our new 3D Capsule.

You should now have a white box with a little black circle on one side. While not hazardous itself, this object **Telegraphs** the dangerous gusts that can appear at any moment. It's important to communicate dangers to your player, so deaths never feel cheap.

Telegraphing Invisible Dangers: It may seem counterintuitive to make a visible object to represent a windy area (something inherently invisible), but even unseen hazards need to be communicated to the player. At all times the player should understand the dangers ahead of them. This is important! Telegraphing Dangers = Good Game Design. Blindsiding Players = Bad Game Design.

3. Next, spawn a 3D Capsule object as a child of the **WindZoneSpawner**. Name this new capsule object **WindZone**. Set its material to be the **matTransBlue** we made earlier in the chapter. Add a **PhysicsForceZone** component to it, and set the **Force** parameter to **200**. Set the transform values to match those in [table 6.11](#)

Table 6.11: Transform values for the WindZone Capsule object.

We now have a group of WindZone objects that will push the player in a given direction. If everything was set properly, your WindZone should look like [Figure 6.16](#)

Figure 6.16: The WindZone object.

4. If you press **Play** and run into the wind zone, it should push your player sphere in the direction of the Y-Axis. If that direction isn't what you expect, make sure the local Y-Axis of the WindZone object is

facing the direction you want. Make sure your tools are set to use the ‘Local’ orientation, which can be toggled in the upper-left corner of the Scene view window as shown in [figure 6.17](#)

Figure 6.17: Use the Local pivot setting to make sure the green Y-Axis is facing the direction you want it to.

5. The last script we need to make is a **ToggleTimer** component. This will enable and disable the wind zone at regular intervals. Select the **WindZoneSpawner**, add a new script called **ToggleTimer**, and add this code in Visual Studio...

```
[SerializeField, Tooltip("Current timer.")]
float _curTimer = 0;
[SerializeField, Tooltip("Seconds between toggling of
objects.")]
float _timerGoal = 3;
[SerializeField, Tooltip("Objects to toggle on/off.")]
List<GameObject> _toggleObjs;
void Update()
{
    // if there are no objects to toggle
    // then don't bother with the countdown logic
    if (_toggleObjs == null)
        return;
    // increment timer
    _curTimer += Time.deltaTime;
    // have we met our goal?
    if ( _curTimer > _timerGoal )
    {
        // reset timer
        _curTimer = 0;
        // go through objects and toggle on/off
        for ( int i = 0; i < _toggleObjs.Count; i++ )
        {
            bool newVal = !_toggleObjs[i].activeSelf;
            _toggleObjs[i].SetActive(newVal);
        }
    }
}
```

```
    }  
}
```

6. The component has a timer that will be incremented every time **Update()** is hit. Once the timer hits its goal, it will go through a list of **GameObjects** and set them to enabled, or disabled, based on their current state.
7. Drag the **WindZone** object from the hierarchy and into the list of **Toggle Obs** in the component panel. If your Toggle Timer component looks like [Figure 6.18](#), then you're ready to test!

Figure 6.18: We've designed the *Toggle Timer* to contain a list of objects that it will turn on, then off, every X seconds.

8. Press the **Play** button. The wind zone should now toggle on and off every 3 seconds. When enabled, the wind zone will push the player as expected. When disabled, the player can make it past the hazard safely.
9. Drag the parent **WindZoneSpawner** into the **Prefabs** folder, and you now have a fun new hazard to use when designing levels!

Figure 6.19: One idea is to line up the zones and place coins between them, which are a great way to telegraph safe areas when using timed hazards in your level.

Utilizing Shadows

As we've built our level and created new objects to jump onto, you may have noticed a general *lack of precision* in those jumps. It's easy enough to move around, and jumps give ample hang-time to correct a mistimed leap, but determining *where you're about to land* can be tricky.

Why is that?

The issue isn't with code, or physics. Instead, the problem is in the **Lighting** of our scene. Specifically, poor *shadow placement*.

That's not to say the Shadows are placed *incorrectly*: The default lighting simply has everything lit from an angle. This results in shadows appearing

to the left and behind an object, making it tricky to read where the player, blocks, ad coins are placed in the 3D scene as shown in [figure 6.20](#)

Figure 6.20: The angle of default lights makes our scene look good, but also makes it hard to determine where objects are in 3D space (which makes precision jumping difficult).

Luckily, the solution is straightforward - rotate the light to get those shadows appearing directly below objects. In the hierarchy window, find and select the **Directional Light** object, and set its rotation value to be straight down as shown in [table 6.12](#)

Table 6.12: The rotation of our main Directional Light.

Then, in the Light component, set the **Intensity** to **0.5**. This controls both the brightness of the light, and the intensity of any shadows cast by it.

While shadows are now more useful, the problem now is that our geometry now lacks definition (both the front and sides have the same gray color). Since we want all sides of our geometry to be uniquely readable, let's add a second **Directional Light** to make our object look more defined.

Figure 6.21: The box on the Left has defined sides (good), while the box on the Right does not (bad). Unfortunately, without a secondary light, our scene now looks closer to the Right image.

With the original light selected, press **[Ctrl] + [Shift] + [D]** to duplicate it. Now select this new light and give it an angular rotation in its **Transform** component as shown in [table 6.13](#)

Table 6.13: The rotation of our second light.

Notice how the shadows are now cast directly below an object. This substantially increases the player's ability to navigate in 3D-Space, especially after a jump as shown in [figure 6.22](#). Adding the second light ensures objects always have a 'dark side' and a 'bright top' - keeping our level geometry easy to read.

Figure 6.22: Proper lighting for any 3D game with jumping as a mechanic.

Gameplay Palette

When designing your game, one useful metaphor is to think of your gameplay elements as a **Palette**. Much like the colors used by a painter, or the chords and tempo used by a musician, game designers have various gameplay elements to build the game from.

Figure 6.23: New gameplay elements = new level design possibilities!

Conclusion

In this chapter, we've used Unity's Physics System to expand our 'gameplay palette' considerably. Pushable blocks, wind zones, springboards, and moving platforms can be used to build a wide variety of levels from.

As we created those assets, we also learned about the intricacies of physics components. Collider types, physics parameters that we can access through the editor or through our scripts. We even touched on the importance of Lighting when lining up a jump. There was a lot to cover, but when it comes to physics – *the literal movement of objects through space and time* – it was bound to be a *robust read*.

With our new knowledge of this *scientific* method of movement, it's time to consider a Unity's tools for *hand-crafting* the movement of objects: **Animation!**

Key Terms

- **Sandbox:** An empty scene that lets us learn and play with systems without the risk of breaking our project.
- **Collider Shapes:** Components that define the shape of an object, as used within the Physics simulation. Box, Sphere, and Capsule shapes are the fastest options. Mesh Colliders can be used for oddly shaped objects, but require more computational power.

- **Mass:** The mass of a Rigidbody object controls how ‘light’ or ‘heavy’ the object feels. Larger mass makes an object difficult to move. Smaller mass makes it easy to move.
- **Velocity:** The speed that a Rigidbody is moving at.
- **Joint:** Physics components that link two Rigidbody objects together. Joint objects will rotate on a pivot location. Should be used for parenting physics objects.
- **Kinematic Objects:** Physics objects that don’t respond to the forces and collisions of the simulation. Useful for when position and rotation needs to be driven from input or from an animation system.
- **Use Gravity:** A boolean that tells a Rigidbody object if it should be affected by the force of gravity.
- **Constraints:** You can Freeze Rotation or Freeze Movement of a physics object. A vital parameter to keep GameObjects from moving in unwanted ways.
- **Is Grounded:** A common (and useful) check for GameObjects that can jump. By performing a downwards raycast, we can determine if a player is on the ground or not, restricting certain actions as necessary.
- **Telegraphing:** As the game designer, it’s important that you’re always guiding the player along and communicating possible dangers. This concept, known as Telegraphing, ensures that the player always knows where to go and when they’re in a dangerous situation.

Multiple Choice Questions

1. Calculating the collision of objects can be quick if you use the right Collider shapes. Which of these collider shapes is the most expensive (hardest to calculate)?
 - A. Box Collider
 - B. Sphere Collider
 - C. Capsule Collider
 - D. Mesh Collider
2. If you want an object to be Easier to Push, You Should...

- A. Increase its Mass.
 - B. Increase its Mass.
3. The following gameplay is traditionally built off a physics system – except one. Which of these does NOT make use of physics?
- A. Combat
 - B. Character Movement
 - C. Dialog
 - D. Grabbing Items
 - E. Enemy AI
4. If you Freeze the Rotation or Position of a physics object, then that object can NEVER be moved.
- A. True: You froze the object. Now it's frozen. What did you expect?
 - B. False: You can still adjust the position or rotation directly in script. Or disable the Constraint in script.
5. You've created an object and placed a Collider On it, but of your onCollision() calls are being hit. You Probably forgot to Add...
- A. The Transform Data.
 - B. The RigidBody Component.
 - C. The Input System.
6. A RigidBody object set to be ‘Kinematic’ will still be moved by the Physics system.
- A. True: It's still a physics object, so of course it will move.
 - B. False: Kinematic objects will still collide with other physics object, but all movement will have to be driven from external sources (player input, animation systems, etc).
7. Using a Box Collider, you've created an area where the player can walk into it to recharge their shield. The problem is, when the player walks to it, there's an invisible wall keeping them from moving into this ‘Shield Recharge’ zone. What the probable cause?

- A. Forgot to add a Rigid Body.
- B. Forgot to set the Collider's 'Is Trigger' parameter.
- C. Your input system is broken.

CHAPTER 7

The Joy of Animation

Just as video game development has been up-ended over the last 20 years, so too has the animation industry. Making an animated movie or television show was once a laborious process - requiring animators to draw, frame by frame, the characters, objects, and special effects that a story required.

The rise of 3D animation, however, changed all that. Cutting edge technology fueled movies such as Toy Story and Monsters Inc. - technologies that required powerful animation tools to tell these engaging stories. Concepts that were once specific to 2D animation - **Run Cycles, Keyframes, Tweening, Squash and Stretch**, etc - were now built directly into these new digital toolsets.

The Unity animation system is built with these concepts in mind, so the tools you'll be learning closely resemble those of leading animation studios. And while this chapter will primarily focus on the animation systems in Unity, we'll also be covering the basics of animation theory, ensuring a well-rounded understanding of this powerful art form.

Structure

- Principles of animation
- Building a character
- The Animation Editor
- Placing Keyframes
- Tweening
- Idle Poses
- Run Cycles
- Coding your animations

Objective

In this chapter, you'll be infusing objects with personality through the mastery of the animation editor and animation graph systems. You'll learn the principles of animation and utilize components that easily and automatically apply these principles to the objects in your game.

By the end of the chapter, you will finally have a hero that feels alive: running, jumping, and moving through the world in a way that exudes character.

Principles of Animation

Animation, as we know it today, is a relatively new art form. It means ‘to give life’ and is a process that tricks the brain into thinking something inherently lifeless is anything but. Early 2D animators would breathe life into a series of still images, while modern 3D animators infuse digital geometry with life.

There are 10 key **Principles of Animation** that can be applied across both 2D and 3D mediums. By making use of these rules, you can ensure your heroes, enemies, NPCs, and even environments exude their intended personality.

1. Squash and Stretch

When an object collides with something, or quickly accelerates in a given direction, Its shape will change in a concept known as **Squash and Stretch**. It has to do with the inherent elasticity, or malleability, of most objects.

Picture a balloon being pushed against a wall. It will be **Squashed** - a flattening of its shape against the surface being collided against. Now think about that same balloon going from standstill to quickly moving in a given direction. It's going to **Stretch**, with the rear end of the object attempting to catch up with the point of acceleration.

We'll be using the **Scale Tool** to make appealing animations that utilize this concept.

2. Anticipation

The time spent ‘building up’ to a given action is known as **Anticipation**. It could be a weightlifter, struggling with a dumbbell before eventually lifting it. This could be a lumberjack pulling back an axe before swinging it. This could be a snake coiling up before striking its prey. The longer the anticipation, the bigger the expected payoff.

Game Design Parallels: While we’ll be primarily using anticipation for creating appealing animations, it’s actually a concept that’s applicable throughout game design. Any game with melee combat will use a **Telegraph** (anticipatory buildup) before a big attack. A game’s **Difficulty Curve** will often have an anticipatory lull before a boss battle. And writers will use NPCs to **Foreshadow** future areas and events, creating anticipation in the player for the wider game world.

Anticipation isn’t just useful for great animations - it’s also a powerful game design tool!

3. Staging

Making sure the viewer (or player) understands what’s important in a scene is the primary goal of **Staging**. This can be achieved through camera positioning, scene lighting, or even character posing. Think back to the previous chapter, when we determined the angle of the light was making it difficult to land jumps. This was a staging issue, where the vital data of “Where am I going to land?” wasn’t easy to understand. A quick change to the lighting fixed that.

If you’re watching somebody play your game, and they keep getting stuck or frustrated, it can often be fixed with **Staging**.

4. Exaggeration

Subtlety is not your friend when trying to convey information quickly. Always push your animation to be as readable as possible. This often means you’ll need to **Exaggerate** shapes and sizes in a way that isn’t realistic. Strange proportions, even if only seen for 1 frame, are a quick way to visually signify something important. A powerful kick may result in a foot that’s 2x its normal size. Being electrocuted can make the skeleton visible for a few seconds. Being poisoned can turn a character completely green.

Don't be afraid to Exaggerate things when animating - especially in video games.

5. Slow In / Slow Out

Every movement requires acceleration or deceleration. In animation, This is a concept known as **Slow In / Slow Out**, where it takes time for an object to start, or stop, moving. This is something modern animation tools do automatically - mathematically easing the speed of motion between two points to be instantly pleasing.

*Coding parallels: Programmers may recognize this concept as **Easing**, interpolating an object between two points using a variety of satisfying methods (Ease In, Ease Out, Bounce In, Bounce Out, etc). Easing is like math magic: a way to instantly make any movement more appealing.*

6. Arc

Similar to the concept of Slow In / Slow Out, **Arcs** in animation mimic the smooth movement curves caused by inertia. While linear (ie. straight & direct) movement between two points feels mechanical, smooth Arcs result in natural, life-like motion.

7. Timing

The number of frames between two poses, or **Timing**, has a massive effect on how an animation *feels*. Imagine a boxer throwing a punch. If it takes 5 seconds to reach its target, that punch is going to feel slow, laborious, and ineffective. Now imagine that same punch, but with only 1/2 seconds between the wind-up and contact. That punch is going to FEEL more powerful, simply by changing the Timing.

The Unity editor makes use of a **Timeline** to allow us to easily manage the number of frames spent in a given pose.

8. Secondary Action

You can add extra appeal to an animation by introducing a **Secondary Action**, an extra bit of movement that infuses your main action with extra personality. A monstrous Ogre may have an idle pose where they occasionally roar into the sky. Perhaps your protagonist is a toddler that likes to pretend they're a super hero. As they run, a good

secondary action would be to stick their arms out and move their shoulders like they're flying.

Secondary Actions let you polish a character's animations, turning a *good* animation into something truly *memorable*.

9. Follow Through / Overlapping Action

When an object is attached to another, and follows behind the main action at a slower rate, this is known as **Secondary Action**. Perhaps its hair, flowing behind a character as they run. Perhaps it's a cape, trailing a superhero. It could even be a hat that bounces on a character's head as they move.

Design your characters with clothing, long hair styles, and even magical auras if you want to accentuate their animations with Secondary Action.

A Note on Physics: If you design your characters correctly, you can often make use of the physics system to handle Secondary Motion. By using physics, you won't have to hand-animate these objects, instead generating real-time movement from the forces of momentum, gravity, and wind. This process can save you time AND it looks awesome!

10. Appeal

While all the above concepts are concerned with *movement* in animation, **Appeal** focuses on a character's design. It's important that the personality of a character is bold: readable at a glance. This means using interesting shapes, unique colors, and strong poses to instantly sell this character's personality.

Is your character tough? Design them using big, thick shapes.

Is your character frail? Design them with small, thin shapes.

Is your character an 'everyday hero'? Give them a friendly, bushy mustache!

Is your character a Bard? Give them a big, obvious *Lute*.

Is your character a Knight? They probably shouldn't have a *Lute*!

Is your character a *Lute*? Awesome!

Spend serious time thinking through the design of your characters. The more Appealing you can make them, the more players will enjoy playing your game, and the more likely you'll see people cosplay as them (the ultimate test of an Appealing character).

*A Note on Pre-Production: Designing an appealing character starts in a phase called **Pre-Production**. This is a dedicated period of brainstorming spent away from your development tools. Don't squander this opportunity to get creative. Make sure you're 100% happy with a design before moving onto the modeling, rigging, and animation part of the process (ie. the expensive part). Design is cheap, fun, and a great way to get unique ideas onto paper.*

Making A Basic Hero

The best way to learn a new system is to have a specific goal in mind. Our goal, as we learn Unity's Animation System, will be to update our boring 'Sphere' player object with a character that has some personality.

Since we'll be editing the player object, let's start by converting the Player Sphere into a Prefab. In the Unity Editor, open the scene named 'SimpleScene' and find **PlayerObj_Sphere**. Drag it from the Hierarchy window into the **Assets > Prefabs** folder.

Next, create a new scene called **AnimationSandbox**. Much like the PhysicsSandbox scene from the last chapter, an empty sandbox allows us to focus on learning, utilizing, and mastering the animation system without existing assets getting in the way.

Once the AnimationSandbox scene has been created, place a new **3D Object > Plane** at Position 0,0,0. This will serve as our 'ground' object - useful when determining where our feet should be placed

Next, ensure the PlayerObj_Sphere object is in your prefab's folder. Once the player object is an official prefab, drag it into the scene, also at the 0,0,0 position. Remove the **Mesh Renderer** component and set the SphereCollider's **Center** and **Radius** values to match those in [table 7.1](#)

Table 7.1: Tweaking our player's Sphere Collider.

With the player object selected, you should now only see the green wireframe sphere of our collider. By adjusting the Center of our collider upwards, it moves the Pivot Point from the center of the object to the bottom - at the hero's feet. Whenever making a character, it's a good practice to keep the pivot point where the feet touch the ground.

We've also decreased the radius of the hero's collider sphere, which will allow them to navigate some of the tighter spaces areas we'll be testing in later chapters.

Now, we need to build a hero that can be animated. We'll use spheres to expand on our current prototype player object, creating a fluffy, cloud-like character that's perfect for learning the animation tools. Once complete, our hero will resemble the image in [figure 7.1](#)

Figure 7.1: From simple shapes to a spherical hero! Once complete, our upgraded player object will look like this.

The beauty of this kind of hero is that they can be made entirely out of spheres. And since 3d Spheres come pre-packaged in unity, it's just a matter of creating and arranging them. Follow these steps to start with the head and feet...

1. In the hierarchy window, right-click on the PlayerObj_Sphere. Select 3D Object > Sphere to add the body sphere.
2. Name this new sphere **Hero1_Body**.
3. Disable the Sphere Collider component on the body.
4. Again, right-click the PlayerObj_Sphere and use 3D Object > Sphere twice to add two new feet spheres.
5. Name these two new spheres **Foot_L** and **Foot_R**.
6. Disable the Sphere Collider components on the feet.

With these three new objects made, use [table 7.2](#) to set their transform data...

Table 7.2: The position, rotation, and scale values for our new hero sub-objects.

With the body made, we'll make some children spheres as the Arms using the following steps...

1. In the hierarchy window, right-click on Hero1_Body. Select 3D Object > Sphere twice to add two Arm spheres.
2. Name one sphere **Arm_L** and the other **Arm_R**.
3. Remove the Sphere Collider script from both of these new objects.

The parent-child relationship of these objects is important. Double-check to make sure these Arms are children of the Body (they will be indented under the Body sphere in the Hierarchy window). You can also verify by moving the Body object. If the arms move with it, then they are properly set up as children of the body.

Use the values in [table 7.3](#) to set the necessary transform data for our Arm objects...

Table 7.3: the transform data for our Hero's arms.

At this point, our hero has gone from a single sphere to five spheres (body, two feet, and two arms). If the transform data has been set properly, your hero should now look similar to the image in [figure 7.2](#)

Figure 7.2: If the Body, Feet, and Arms were set up correctly, your hero should currently look like this.

They say that eyes are the ‘window of the soul’, so any character you create should have eyes that reflect their personality. For our hero – a cute, fluff-ball character – we’ll give them simple, appealing eyes (which can also be made from spheres). These steps will get you started...

1. In the hierarchy window, right-click on Hero1_Body. Select 3D Object > Sphere twice to add two spheres for our Eyes.
2. Name one sphere **Eye_R**, and the other **Eye_L**.
3. Remove the Sphere Collider script from both of these.
4. Right-Click on each of these eye objects and add a sphere child to each.

5. Name both child spheres **WhiteGlare**.

With these four eye objects made, use [table 7.4](#) to set their transform data...

Table 7.4: The position, rotation, and scale values for our eye sub-objects.

Our last step is to create a tuft of hair to use when applying secondary motion to the character. Large or small, it's always good design practice to add something that can trail your hero: long hair, a scarf, a cape, or perhaps simply a magical aura. It's a subtle effect, but a secondary motion – one that works as a trail - helps the hero to stand out in a hectic scene.

Let's add our trailing tuft of hair by following these steps...

1. In the hierarchy window, right-click on Hero1_Body. Select 3D Object > Sphere to create our first Hair object.
2. Name this object **Hair_1** and remove the Sphere Collider component.
3. Right-click on Hair_1 and add a child sphere named **Hair_2**.
4. Right-click on Hair_2 and add a child sphere named **Hair_3**.
5. Right-click on Hair_3 and add a child sphere named **Hair_4**.
6. Make sure all these hair objects have their Sphere Collider scripts disabled or removed.

Again, the parent-child setup here is important, so check the Hierarchy window to make sure all the hair objects are children of the preceding objects (4 is a child of 3, 3 is a child of 2, and so on). Once all this has been verified, use [table 7.5](#) to set the position, rotation, and scale of these tufts of hair...

Table 7.5: The transform values to set for the hair objects.

If everything was entered correctly, you should now have a round, colorless hero with arms, feet, eyes, and a flowing mane of poofy hair. Feel free to color the hero how you want.

The important part – setting up a character to animate – has been done. Now it's time to dive into the tools.

The ‘Idle’ Pose

The default animation used by a character is known as the **Idle** pose. This should never be too flashy, or have too much movement, since the character is simply waiting for you to press the button. Sonic the Hedgehog famously taps his foot and gives you an impatient glare when he’s in his Idle pose for too long.

In this way, the idle animation can be a fun way to display the personality of your video game character.

The Idle pose of our hero, however, is going to be simple. We’ll rotate their head to the right, rotate it to the left, then back to the center position. Add in some ‘blinking’ and it’ll be ready for action - another ‘simple yet effective’ solution.

To get started, we’ll need to first get acquainted with Unity’s main animation tool: the **Animation View** window.

The Animation View

As with most of the ‘advanced’ systems in Unity, you’ll notice that the animation tools are hidden by default. Let’s unhide the **Animation View** now: go to the top menu bar to select **Window > Animation > Animation** as shown in [figure 7.3](#)

Figure 7.3: Open the Animation Editor by selecting Window > Animation > Animation.

Once open, drag this new window down into the bottom panel, where it will become an additional tab next to the ‘**Project**’ and ‘**Console**’ options ([figure 7.4](#))

Figure 7.4: Dock the Animation View window in the bottom panel of the editor. Note the Create button, which we’ll use shortly.

The next step is to select the root object that you’ll be animating. In this case, that would be **PlayerObj_Sphere**. Select that object in the Hierarchy window, then select the **Create** Button In the middle of the Animation View window.

This performs several actions. First, it attaches a new **Animator** component to your object. This handles the various animations we'll be making. We'll be accessing the Animator script, using code to swap between those animations.

It will also create our first animation file, which we'll name **SphereHero_Idle**. These files (which use the filename extension **.anim**) hold all the data for a specific animation. Eventually, we'll also need a unique **.anim** file for Running, Jumping, and Falling.

With our empty Idle animation created, it's time to specify the sub-objects we'll be moving around. These will be listed on the left side of the Animation View, similar to how the Hierarchy Window lists objects.

Go to the left side of this window and press the **Add Property** button, which displays a list of all the parameters we can affect. Select **Hero1_Body > Transform > Rotation** and press the + button, as shown in [figure 7.5](#). Do this again, but now add the **Hero1_Body > Transform > Position** parameter. What we're doing here is notifying Unity that we'll be animating both the *Position* and *Rotation* of the 'Body' object. These two parameters should now be the only two listed.

Figure 7.5: If an object isn't added, then it won't be animated. Use the Add Property button to add objects (and other parameters) to the Animation View window.

Animating Script Parameters: An incredibly useful aspect of this system is that it allows you to affect the components of an object. You'll notice that the PlayerController and HealthManager scripts (the ones created in [Chapter 1](#) and [2](#)) are both listed here. This means that any serialized members of those scripts can be modified by our animations. This makes the animation system useful for purposes well beyond 'making the character run and jump'.

Keyframes and Inbetweening

Now that we've specified the objects and properties we'll be animating, you should notice a diamond shaped dot to the right of each parameter. These are called **Keyframes** - a specified value at a specified time. These small bits of data are the *cornerstone* of the digital animation process. A

Keyframe stores the position, rotation, scale, (or any other parameter) at a given frame. When there are two keyframes present, the computer performs a process known as Inbetweening, or **Tweening** for short. Tweening is the interpolation of values between one keyframe and the next.

All of this is managed on the **Dopesheet** - the grid on the right side of the Animation View window ([figure 7.6](#)).

Figure 7.6: The Dopesheet is where you'll be managing the keyframes of your animation.

Using both **Keyframes** and **Tweening**, the animation system allows us to create complex, multi-frame animations by simply setting poses that get interpolated at runtime. It's powerful, quick, and easy, but like most concepts in this chapter, it's something you really need to see in action.

Let's start our idle animation by defining its **Length**, which we'll make 6 seconds long. Position your cursor over the Dopesheet, and use the Mouse-Wheel to expand the timeline. If the timeline gets too large, you may need to hold the middle-mouse button to drag the dopesheet back to the starting keyframes.

Keep scrolling the mouse-wheel until you see **6:00** appear at the top, then do the following:

1. Click and drag to select all keyframes at the **0:00** mark.
2. Use **[Ctrl] + [C]** to copy them.
3. Go to the **6:00** mark and click on it.
4. Use **[Ctrl] + [V]** to paste those keyframes ([figure 7.7](#))
5. Optional: There may be keyframes at the 1:00 mark. If so, click and drag to select them, then press the Delete key to remove them.

When making a looping animation, you'll always want the first frame and last frame to have the same keyframes. Without them, you'll get weird popping of objects when the animation loops.

Figure 7.7: The length of an animation is determined by the timestamp of the last keyframe.

Looping Idle Poses: Most of the animations we make will be **Looping**: they'll return to frame 1 after the final frame. In most cases a shorter

cycle is ok, but for an Idle pose, a short cycle will result in a very twitchy, very unappealing character. Always make your Idle animation several seconds long to keep the loop hidden and the character appealing.

Since keyframes are necessary when making an animation, we should cover the various ways you can add them to your timeline. Using the Animation View, there are three preferred ways to add keyframes to your animation:

1. Copy and Paste (which we performed above)
2. Right Click on the Dopesheet, then select “Add Keyframe”
3. Automatically adding keyframes with **Recording Mode**

For our purposes, we'll be using the Keyframe **Recording Mode** for animating our hero. When it's active, a keyframe will be added whenever the transform of an object changes (ie. when you adjust its position, scale, or rotation). This is a fast way to animate - just be careful that you don't accidentally add keyframes where you don't intend to.

To start recording, press the small Red Circle button in the upper left corner of the Animation View window as shown in [figure 7.8](#)

Figure 7.8: When enabled, Recording Mode will automatically add keyframes to an animation.

Our Idle animation will simply be our hero looking around. This should only require us to add two keyframes, so let's do that now...

1. Select the **Hero1_Body** object, either from the Animation View or the Hierarchy Window.
2. Switch to the **Rotation Tool** by pressing the [E] hotkey.
3. Go to the **2:00** mark on the timeline.
4. Rotate the head to look towards the right, or use the Transform panel to set the rotation to (30, -40, -5).
5. Go to the **4:00** mark on the timeline.
6. Rotate the head to look towards the left or use the Transform panel to set the rotation to (25, 20, 3).

And that's it! You should now have two new keyframes - one at 2:00, the other at 4:00. Click anywhere on the Dopesheet, then press **[Spacebar]** to **Play** our animation. You should see the hero look slowly to the right, then to the left, as shown in [figure 7.9](#). Since we copied the starting keyframes to match the last keyframes, this will continue in a seamless loop. Press **[Spacebar]** again to **Stop** the animation.

Figure 7.9: A simple ‘looking around’ Idle animation, and it only required a few keyframes!

Another useful way to test your animation is by **Scrubbing**: manually setting the current frame by dragging the mouse along the timeline. Try this now: move your cursor over the timeline (the bar that lists times between 0:00 and 6:00). Left click on the timeline bar and drag the ‘current frame’ forwards and backwards.

Our Idle animation looks *good*, but we can do *better*. Let’s place some additional keyframes to add even more life and personality into our hero. We’ll start with the eyes, adding some quick Press blinking movements (an easy way to imply cognition in a character).

1. Click on the timeline and Scrub to frame **130** (Shown in a small white box to the right of the Fast Forward button)
2. Use the Add Property button to add **Eye_L** and **Eye_R** Scale properties
3. Right click on these entries and select **Add Key**
4. Scrub to the frame 136
5. Again, Right click on the eye entries and select **Add Keys** ([figure 7.10](#))
6. Scrub to frame 133
7. Use the [R] keyframe to switch to the Scale Tool
8. Drag the Scale tool’s green bar (Y-Axis) to make the hero’s eyes nice and squinty. Because you’re in Record mode, you will see new keyframes automatically added.

Figure 7.10: Add Keyframes for a specific property using the Right-Click menu’s ‘Add Key’ option.

Press the **[Spacebar]** to see our blinking animation in action. The outer keyframes (at frames 130 and 136) store the ‘normal’ size of the eyes, ensuring the scale of the eyes return to their proper non-blinking values. The inside keyframe (at frame 133) handles the squashing of the eye, with tweening filling the frames between. While not the most exciting animation trick, a blinking character exudes much more life than a non-blinking one.

Figure 7.11: Our blinking animation - six keyframes well spent!

Additional Polish: When it comes to animation, you can spend a lot of time in the ‘polishing’ phase. Right now we have a decent Idle pose, but there are several ways to improve it. You could copy and paste our blinking keyframes to make the hero blink a few more times. You could set a few keyframes on the hair objects to give them some secondary movement as the body moves around. While we won’t dive into the specifics of these, it’s always good to be looking for ways to push towards animation perfection.

We now have a hero with a subtle Idle pose, looking around as they wait for the player’s input. By using **Keyframes** and **Tweening**, we were able to create a 6-second animation with only a handful of keyframes. In the days of 2D, this movement would have taken *days* to draw by hand. A testament to the power of computer generated animation.

Now that you know the basics, let’s get a bit more advanced with our next animation: the **Run Cycle**!

A Warning about Recording Mode: While it does speed up the animation process, Recording Mode can lead to extra keyframes that you may not expect. In most cases, this will simply result in strange animation pops. But if you accidentally set a keyframe for the ROOT object (the main parent for the objects being animated), it may completely break the movement of this object, since its transform is now being set via a keyframe. If you ever run into an issue where your movement breaks, check to see if a stray root-object keyframe made it into one of your animations. And be careful when that red ‘Recording’ button is pressed!

Run Cycles

The Idle pose ensures that our hero has personality when no input is received, but what happens when the player starts pressing buttons? A hero sliding around the level in their Idle pose would look *fairly silly*, so let's talk about the next important animation to hook up.

A **Run Cycle** is the animation clip used when the player is *moving* and *grounded*. It tends to be short: normally a 30 frame looping animation. Since our hero is bipedal (ie. only has two feet) the cycle is broken up where 15 frames are spent on the Right foot, and 15 frames spent on the Left foot.

Let's start by creating a new clip. In the upper left corner of the Animation View, you will see a dropdown menu with the name of the current Animation clip (look for "SphereHero_Idle"). Click this area and select **Create New Clip** from the drop down menu. Name the new animation **SphereHero_Run**.

Figure 7.12: Create new animation clips for your character from this dropdown menu.

We now have an empty animation clip, but a Run Cycle is considerably more complex than an Idle Pose, so where do we start? This 'blank canvas' moment is often the hardest part of the process, so let's begin by listing the requirements:

1. **Body:** We'll need to move the body up and down with each step. It should rise and fall 2 times within the animation - once for every time a foot touches the ground,
2. **Feet:** We need to place and rotate the feet, so that one foot is hitting the ground when the opposite foot is raised.
3. **Arms:** We need to swing the arms, such that its movement is the *inverse* of the foot (ie. when the right foot is facing forward, the right arm should be pulled back.)
4. **Hair:** We should add some secondary animation to the fluffy hair objects. A way to ensure your animation more appealing is to ensure dangly bits are given the proper attention.

Figure 7.13: The frames of our ‘Running’ animation, where we alternate between landing on the hero’s Left and Right feet.

Since the Body object is first on our list, let’s start by adding parameters for the **Position**, **Rotation**, and **Scale** of **Hero1_Body** as shown in [figure 7.14](#)

Figure 7.14: Since we’re going to animating the Body, we’ll need to add it’s transform properties to the Animation View window

Now we need to do some keyframe management. Use the following steps to set up our initial Running keyframes…

1. Move the keyframes for each of the properties from the 1:00 mark to the 0:30 mark. Because Unity determines animation length by the timestamp of the last keyframe, we’re telling Unity that our run cycle is 30 seconds long.
2. Enable **Keyframe Recording Mode** (the little red dot next to the rewind button).
3. Scrub to frame 8 (timestamp 0:8) and move the body upwards (Y position should be around 0.9).
4. Scrub to frame 15 and move the body back down (Y position around 0.6).
5. Scrub to frame 23 and move the body upwards (Y position around 0.9).

Press the play button (or use the [**Spacebar**] hotkey). The body of our hero should now be bobbing up and down, with the eyes, hands, and hair following it. We can add a bit more appeal by adding some rotation and squash-and-stretch at the keyframes listed in [table 7.6](#) (and note that we’re only changing Rotation and Scale here – don’t accidentally change position values).

Reminder: If you hand-type these values, remember to either add Rotation and Scale keyframes, or have Recording Mode enabled. If you forget to add keyframes and adjust these numbers, you’ll only end up messing up the values for the Frame 1 keyframe.

Table 7.6: The Rotation and Scale keyframes to add to the Body object.

At this point the body should be bouncing nicely. Play your animation, or scrub the timeline, to see how it looks. You should notice some appealing *squash-and-stretch* now that we've added the Scale and Rotation keyframes, with frames 8 and 23 stretching the hero vertically, and frame 15 squishing the hero horizontally.

It's a subtle effect, but important when making a satisfying movement.

Now we'll move onto the feet. Use the Add Property button to add **Position** and **Rotation** parameters for **Foot_R** and **Foot_L**.

Assign the following keyframe data for the Foot objects as laid out [Table 7.7](#)

Table 7.7: The Position and Rotation keyframes for the feet objects.

Notice there is no data listed for Frame 30. This is because, whenever you have a looping animation, the last frames should always match the first frames. Simple copy and paste the keyframes from Frame 1 to the 0:30 mark - this will ensure a seamless loop.

Now, if you play the animation, you will see an acceptable foundation for a **Run Cycle**. The arms may not be moving, and the tuft of hair feels a bit stiff, but in general it's a great start.

Let's polish our Run by adding some keyframes to the arms. Use **Add Property** to add the **Position** of **Arm_L** and **Arm_R** (nested under the **Hero1_Body** object). Once those properties have been added, hook up the frames shown in [table 7.8](#)

Table 7.8: The Position keyframes for the arm objects. Again, remember to use the Frame 1 values for Frame 30, to ensure the animation loops properly.

For the final bit of polish, we'll be adding some **Secondary Action** to the hero's 'Hair'. However, unlike the steps taken above, we're going to rely solely on the Keyframe **Recording Mode** to add our parameters and keyframes.

With Recording Mode enabled, find the object named **Hair_3** in the Hierarchy window (you may need to expand several child objects within the hero). With Hair_3 selected, perform the following steps...

1. Enable the Rotate Tool using the hotkey **[E]**.
2. Scrub the timeline to Frame 0 (timestamp 0:00).
3. Rotate the Hair object EVER SO SLIGHTLY. This little movement will register as a ‘Change’ and add the object’s rotation to the Animation Property list. It will also add a keyframe at the 0:00 mark.
4. Scrub the timeline to Frame 30 (timestamp 0:30).
5. Again, rotate the object very slightly, which causes the Keyframe Recorder to add a keyframe at the end of the timeline.
6. Scrub the timeline to Frame 15 (timestamp 0:15).
7. Rotate the Hair object to the Left approximately 50 degrees. Make sure the tool is set to rotate around the Pivot (opposed to Center).

Turn off Recording Mode and play the animation. You will now see the end of the hero’s ‘tuft of hair’ following a few steps behind the larger tufts of hair. This is known as Secondary Action, one of the Principles of Animation that we talked about earlier.

And just like that, we now have a run cycle for our hero! Great work!

Animations in Action

You’ve spent all that time getting those animations hooked up, and now it’s time to put your hard work to the test. Press the **Play** button to start the game and use the **Arrow Keys** to move your updated hero around the scene ([figure 7.15](#)).

Figure 7.15: Ummmmmmmm. What?

This is definitely NOT what we expected. Instead of running around like a proper hero, our object seems to be rolling around in their Idle pose. What went wrong?

Well, nothing went *wrong*, technically. Our hero is still moving using on how we set up the code and physics of the original sphere. We simply have a few script tweaks to make to get our hero moving around as expected.

First, let’s deal with the whole ‘rolling around’ situation. When we made our temporary hero object, we constructed it out of a 3D Sphere object and

let the physics engine do the rest. While there's nothing inherently wrong with this, it did mask the fact that we've been rolling around all this time.

The problem is, we don't want the physics system having that much power over the transform of our object. We want it to drive movement, not rotation. And when we want to freeze the rotation of a Rigid Body, all we have to do is toggle the appropriate boxes.

Figure 7.16: Freeze rotation on all axes of our player object.

Physics will no longer change our players rotation, but we DO want to face in the direction that our hero is moving. Open the **PlayerController** script and add the following code at the bottom of the **Update()** function:

```
// set rotation based on facing  
transform.LookAt(transform.position + new  
Vector3(_curFacing.x, 0f, _curFacing.z));
```

Next, we need to add some code to make our player swap between the Idle and Run animation states. Select **PlayerObj_Sphere**, locate the **Animator** component's **Controller** parameter, and double-click the field to the right of it. The Animator Controller window will be unhidden, showing the animation states associated with the selected object ([figure 7.17](#)).

Figure 7.17: The Animator Controller window.

All we want to do here is make sure our Idle and Run animations are named correctly. Select each entry in the graph, and rename the **ShereHero_Idle** state to **Idle** and the **SphereHero_Run** state to be **Run**.

Now you can return to the **PlayerController** in Visual Studio. Add two new members at the top of the script:

```
// store whether or not input was received this frame  
bool _moveInput = false;  
// the animator controller for this object  
Animator _myAnimator;
```

Then update the **Start()** function to find and store the Animator component:

```
void Start()  
{
```

```
_rigidBody = GetComponent<Rigidbody>();  
_myCollider = GetComponent<Collider>();  
_myAnimator = GetComponent<Animator>();  
}
```

At the top of the **Update()** function, you'll want to reset the **_moveInput** before we start calculating key presses:

```
// reset move input  
_moveInput = false;
```

You'll then need to set this **_moveInput** value to **true** if input has been received this frame. Add this code in any of the **if (Input.GetKey())** statements to register that a key has been pressed:

```
_moveInput = true;
```

At the bottom of the **Update()** function, call new a function with the line:
`UpdateAnimation();`

Then define that function with the following block of code:

```
void UpdateAnimation()  
{  
    if (_myAnimator == null)  
        return;  
    if (_moveInput)  
        _myAnimator.Play("Run");  
    else  
        _myAnimator.Play("Idle");  
}
```

This function first checks to see if we have a valid Animator component. If so, it will set either the Run or Idle animation based on whether movement keys are being pressed by the player.

Back in the Unity Editor, press the **Play** button to test our new code. Our hero will no longer roll around, and when buttons are pressed you'll see the running animation played as expected.

Our final step is to update the player prefab. We've made several changes to this object, and we want those changes to be applied to other instances of the prefab (in our SampleScene, for instance).

To do this, select **PlayerObj_Sphere** and find the **Overrides** dropdown in the Inspector. Click **Overrides** and select **Apply All** (in the lower right

corner) as shown in [figure 7.18](#)

Figure 7.18: You'll need to Apply your prefab changes for them to show up across in other instances of the player object.

Cleaning Up

Unity does a great job of automatically generating files for your animations. One unfortunate side-effect, however, is that it's easy to generate assets in a spot you don't intend.

A good practice is to end your day with a quick bit of *clean-up*. Hunt down the assets that have been automatically generated and make sure they're properly organized in your project.

Let's start our clean-up process by creating a new folder in your Assets directory. Name this folder "Animations". Next, use the Project window to search for the newly created animation files. Once you find them, drag them into your empty Animations folder as shown as follows:

Figure 7.19: Cleanliness and organization is important as your project grows.

Unity doesn't have any requirements as to where these files should be placed, but once your game project it's large enough, you'll be happy you spent extra time keeping everything organized.

Conclusion

It's truly surprising how much power **Animation** has. We were able to take a dozen spheres and turn them into a cute little hero. By using the animation systems provided by Unity - as well the basic concepts from the **Principles of Animation** - you can take almost any object you can think of and bring it to life. Towering Heroes, timid NPCs, terrifying enemies.

And speaking of Terrifying Enemies, it's finally time to get our enemies feeling fierce. It's time to learn about **Enemy AI**.

Questions

1. The principles of animation were developed to help breathe life into static objects. Can you name some of these principles?
2. If you want to apply Squash-and-Stretch to an object, what tool should you use?
3. When is an Idle pose used? Why is it important to make an Idle pose longer than other animation states?
4. How does Unity determine the length of an animation?
5. What information does a keyframe hold? How do keyframes and interpolation speed up the animation process?
6. Why do looping animations use the same keyframe for both the first and last frames?

Key Terms

1. **Animation:** The process of moving an image, or set of objects, in a way that makes it feel ‘alive’.
2. **Keyframes:** A data value at a given frame. Used for storing changes in position, rotation, and scale over time. A vital component of any animation system.
3. **Tween:** The process of taking two keyframes and interpolating the frames between.
4. **Principles of Animation:** Rules you can follow to add personality and appeal to the movement of your characters.
5. **Squash and Stretch:** The changing of an objects shape as it hits something, or accelerates in a given direction. Use the Scale tool to add this elasticity to your 3D animations.
6. **Idle Pose:** The default pose of a character as they wait to be told where to move.
7. **Run Cycle:** A short looping animation that plays when a character is running in a given direction.
8. **Dopesheet:** The grid on the right side of the Animation View window. Holds all the keyframes for a given animation.
9. **Timeline:** A bar of time signatures at the top of the dopesheet. Use this to determine the length of your animation and to adjust the timing

of your movements.

10. **Scrubbing:** Moving the current animation frame by clicking on the timeline and dragging the mouse left and right.
11. Recording Mode: A mode that automatically generates keyframes and animation parameters as you adjust the transform of objects.
12. Prefab Overrides: Changes made to a prefab object. Apply them to ensure those changes are made to all instances of the prefab.

CHAPTER 8

The Mind of the Enemy

A hero is only as good as their villain. This is a famous guiding principle of storytellers, one that ensures the strengths of a hero are matched against an equally powerful foe.

It's also a principle that holds true in Video Game design. Without great enemies to test the player's skill, a game - much like a story - will feel *woefully unsatisfying*.

That's why, in this chapter, we're going to dive into the *Mind of the Enemy*: creating components and logic that can drive the intelligent movement of new bad guys.

Let's learn how Unity provides all the tools needed to make enemies that are *great* at being *bad*.

Structure

- Artificial Intelligence
- The Basic Enemy
- AI Components
- Unity Events and Actions
- Hunting Down the Player
- Data-driving your AI
- Navigation Meshes
- NavMesh Agents

The Thought Process

Enemies in video games take all shapes and sizes. You have 'single-hit' cannon fodder foes, agile enemies, stationary enemies that launch

projectiles from afar, and your epic boss-battle enemies.

With gamers expecting this large variety of unique enemies, it can be difficult for a new developer to know where to start.

First, it's important to realize that, programmatically speaking, **Enemy** objects aren't that much different than the **Player** object. They both have *health* associated with them. They both components that dictate how fast they can run, how high they can jump, and what weapons and items they can use. The key area of differentiation lies in how decisions are made.

For the player object, all decisions are made by the user - the gamer pressing the buttons on the keyboard or controller.

Enemy objects, on the other hand, are driven by **Artificial Intelligence (AI)** - code that tells them what to do and how to do it. We'll be creating a component called **AI Brain** which will observe the game world (location of the player, obstacles in the area, etc) and make decisions based on that data.

Our Basic Enemy

Before we start coding the AI Brain component, let's make a little enemy to attach it to. A simple, generic, one-eyed bad guy will do the job nicely!

While we're still using prototype art, we now have enough objects in our game where quick differentiation is important. And luckily, this easily identifiable foe can be made from 3 cubes and 2 spheres.

Make an empty GameObject named **EnemyObj_BasicFoe**, then add the following child objects:

- 3D Object > Cube named **CubeHead**
- 3D Object > Sphere named **YellowEye**
- 3D Object > Sphere named **EyePupil**
- 3D Object > Cube named **Eyebrow_L**
- 3D Object > Cube named **Eyebrow_R**

Once the child objects are created, use the values in [table 8.1](#) to place the objects and create our *basic enemy*.

Table 8.1: The transform data for our Basic Foe's subobjects.

Like with the Spikes we made previously, properly coloring our enemy is a great way to identify it as something dangerous. Apply the **matColorRed** material to **CubeHead** and **matBlack** to **EyePupil**, **Eyebrow_L**, and **Eyebrow_R**. Then make a new **matYellow** material and apply it to the **YellowEye** object.

Once done, your enemy should resemble the one-eyed creature shown in [figure 8.1](#)

Figure 8.1: A nice, red, generic enemy. The V-shaped eyebrow sells it.

New Component: AIBrain

With our basic foe built, It's time to infuse it with a little artificial intelligence. To do that, we will be creating a new component called **AIBrain**. This script will manage *what* the enemy wants to do, also known as the **AI State**. Each possible **State** will have a series of actions that can be assigned to it, making full use of the Unity Editor to **Data-Drive** how our enemy acts (and reacts) to the player.

Select the **BasicFoe** object and select Add Component > New Script to add a component called **AIBrain**. Open this Script in Visual Studio and add the following members:

```
#region ** members **  
  
// the current set of AI actions  
  
UnityEvent _curAIDirective;
```

```

[SerializeField, Tooltip("Default events for this AI.")]

UnityEvent _defaultActions;

[SerializeField, Tooltip("Events to trigger when alerted.")]

UnityEvent _alertedActions;

[SerializeField, Tooltip("Events to trigger when hunting the
player.")]

UnityEvent _huntActions;

[SerializeField, Tooltip("Misc Patterns of AI movement.")]

public UnityEvent _miscPattern1Actions;

public UnityEvent _miscPattern2Actions;

public UnityEvent _miscPattern3Actions;

// timer for pausing AI logic

float _pauseTimer = 0;

// we need quick access to the player object

PlayerController _playerObject = null;

#endregion

```

First off, you probably noticed the use of a new set of tags: **#region** and **#endregion**. These lines of code specify a block of data that can be collapsed within Visual Studio. You should notice a small box with a minus sign with in it next to the **#region** tag ([figure 8.2](#)). Clicking this will condense all the members into a single visible line.

Regions are an important tool for keeping long scripts organized.

Figure 8.2: Use the [-] icon next to a #region tag to consolidate a block of code. This will make your longer scripts considerably more readable.

Second, you probably noticed several **UnityEvent** members being added. These will hold the Actions associated with the various AI states, and we'll be talking about them in the next section.

Thirdly, the last few members dealt with the pausing of AI and the storing of a player object. The member **_pauseTimer** can be set to pause the movements of any enemy, in an important way to give the player time to react to a situation.

The **_playerObject** allows us to cache the player's information for quick retrieval. This is because almost every AI decision will be based on the location and status of the player. Which direction to face, whether to patrol or chase, and when to fire a weapon are all based on the player's location in relation to the enemy's. Retrieving the player every frame, across all enemy AI Scripts, would be computation time better spent on making smart decisions.

Now that the members are added, let's hook up a handful of necessary functions:

```
private void Start()
{
    // find the player object in the scene
    _playerObject = GameObject.FindObjectOfType<PlayerController>();
    // set default actions
    _curAIDirective = _defaultActions;
}
```

```

void Update()

{
    if (UpdatePausedAI())
        return;

    _curAIDirective.Invoke();
}

bool UpdatePausedAI()

{
    if (_pauseTimer > 0)
    {
        _pauseTimer -= Time.deltaTime;
        _pauseTimer = Mathf.Max(_pauseTimer, 0f);
    }

    return (bool)(_pauseTimer > 0f);
}

```

The **Start()** functions find the player object and initialize the current AI directive to use the default actions. The **UpdatePausedAI()** function determines if the AI should be running this frame. And, if it SHOULD be run, the **Update()** function will **Invoke** the current UnityEvents (call all the functions associated with the current State of the AI).

Since so much of our AIBrain relies on these UnityEvent objects, let's take a moment to learn more about these powerful data types.

UnityEvents and AI Actions

When hooking up our members, you will have noticed that several of them use a new data type: **UnityEvents**. This Is a Unity-Specific type that lets you queue up function calls directly from the editor. This leverages power and flexibility in a way that's perfect for our AI system.

A UnityEvent is a List of function calls associated with a specific object in the scene. When hooking up our UI, for instance, Are buttons all made use of unity events to trigger functions when they were pressed.

Our AIBrain script will Handle States in a similar way: If an enemy is in a given state, it will trigger a set of unity events assigned to that state.

Let's start by creating a handful of public functions that can be called to specify what state the AI should be in:

```
#region *** AI State ***

public void SetState_Default()

{
    _curAIDirective = _defaultActions;
}

public void SetState_Hunt()

{
    _curAIDirective = _huntActions;
}

public void SetState_MiscPattern(int pattern)

{
```

```
/* Todo: Ch14 */
```

```
}
```

```
#endregion
```

These functions should look fairly straight-forward, with each SetState function assigning a new group of actions as our **_curAIDirective** (the UnityEvents that get **Invoked** every Update call).

Below, you'll find the second batch of code that our UnityEvent system relies upon - AI events that can be assigned to the various AI states:

```
#region *** AI Events ***

public void Jump( float force )

{

    GetComponent<Rigidbody>()?.AddForce( new Vector3(0, force, 0)
);

}

public void AlertIfPlayerNearby(float distance)

{

    if (CalcDistanceToPlayer() < distance)

        _alertedActions?.Invoke();

}

public void PauseAI( float timeInMS )

{

    _pauseTimer = timeInMS;
```

```
}
```

```
public void UseWeapon()
```

```
{
```

```
/* Todo: Ch10 */
```

```
}
```

```
#endregion
```

AI Event function should always be short and sweet, only doing one specific task at a time. The more functions you can assign, the more robust and dynamic your AI can be. In fact, you can see where, in [Chapter 10](#), we'll be hooking up a new AI function to handle the use of weapons.

But for now, we're going to focus on one specific, and important, enemy directive: *hunting down the player*.

Hunting Down the Player

In most games, the primary goal of an enemy object is to serve as a mobile hazard that the player has to deal with. It's something to jump over, attack, or simply avoid before moving onto the next area.

There are certain enemies, however, that strike considerably more fear in the player. By adding some additional code, we can make our enemy *hunt the player* - chasing them around the level, forcing the player to make a move. By putting the player on the defensive, this style of enemy feels considerably more dangerous than your typical *mindless grunt*.

With 'murdering the player' as this enemy's top priority, it's vital that the AIBrain keeps track of the players location. This information can be used to aim weapons, chase the player, or even retreat if a situation turns hopeless on an overzealous AI.

Here are the player-related functions we'll be using in our AIBrain component:

```
#region *** Player Hunting ***

float CalcDistanceToPlayer()

{
    return Vector3.Distance( transform.position,
    _playerObject.transform.position );
}

Vector3 CalcPlayerPos( bool ignoreY = false )

{
    Vector3 playerPos =
    _playerObject.gameObject.transform.position;

    if (ignoreY)

        playerPos.y = transform.position.y;

    return playerPos;
}

public void LookAtPlayer()

{
    transform.LookAt(CalcPlayerPos(true));
}

public void MoveTowardsPlayer( float speed )
```

```

{

    // move towards the player

    Vector3 playerPos = CalcPlayerPos(true);

    Vector3 newPos = transform.position;

    playerPos.y = transform.position.y;

    newPos += (playerPos - transform.position).normalized * (speed
        * Time.deltaTime);

    transform.position = newPos;

    // look ahead towards the player

    transform.LookAt(playerPos);

}

#endif

```

These functions not only give us quick access to the location of the player, but we now have a simple **MoveTowardsPlayer()** function that will turn our foe towards the player and hunt them down.

Keeping Things Tidy: Like the other #regions of code, you can now use the [-] to collapse these functions. If you've consolidated all the code you've written, the AIBrain script should now look nice and tidy, with 140 lines of code condensed down to 8 lines, as shown in [figure 8.3](#).

Figure 8.3: If you've been using the [-] buttons in Visual Studio to consolidate your code, this is what your AIBrain script should look like.

As proud as you should be of both the AI script and your commitment to organization, we have one more step before testing our new Enemy: *data-*

driving the AI.

Setting up the AI Component

Return to the Unity Editor so we can start filling out the **AIBrain** component. When you select the enemy object, the inspector window will show an empty AIBrain that resembles [figure 8.4](#)

Figure 8.4: By default, there's *literally* nothing going on inside this enemy's brain. Fill it with intelligence by filling these empty lists with UnityEvent data.

Each of these lists represents a different **State** that the AI will manage. We'll start by filling the *default actions* of this enemy. Do this by clicking on the small plus button in the lower right corner of the **Default Actions ()** panel as shown in [figure 8.5](#).

Figure 8.5: Add a new Event to the DefaultActions list by pressing the + button.

Each event in this list is composed of three parts, each of which we'll need to set. First, we need to specify the game object that this event will be acting upon. Second, we'll need to set the Component and Function being called by this event. And third, we'll need to specify any parameters required by the selected function.

Setting the object associated with this event is as simple as dragging that object from the Hierarchy window to where it says **None (Object)** in the event panel ([figure 8.6](#))

Figure 8.6: Drag our 'Basic Foe' object into the event panel (where it says None).

Once an object has been assigned, Unity will search through all the possible components and functions we can call upon that subject. Click on the drop down menu that currently says ‘*No Function*’ to see all the possible functions the AI can trigger ([figure 8.7](#))

As a game designer, this list of available functions should get you excited about all the gameplay possibilities! By data-driving the actions performed during an AI State, Unity automatically gives us access to hundreds of existing functions that can be modified by our AI. Want to set an object to be enabled or disabled? What about changing the scale of an object with the Transform component? Or perhaps you need to lock an object’s movement through it’s Rigidbody component. By controlling the logic of AIBrain using **UnityEvents**, all that functionality is instantly at your fingertips.

Figure 8.7: Our Actions can trigger ANY public function on any of the scripts attached to this object. This makes UnityEvents - and thus, our AIBrain - both versatile and powerful.

Now - back to our Basic Foe. The default action of this enemy will be to sit and wait until the player gets too close. To accomplish this, we’ll use the function **AlertIfPlayerNearby()** that we added to AIBrain. All we have to do now is assign that function in our default action list as shown in [figure 8.8](#).

Figure 8.8: Our Default action is to simply look out for the player object.

The alert function we’re calling takes a distance value, which serves as our Alert radius. Once the player gets this close the the enemy, the AIBrain will enter its Alerted state. In this case, we’ve set this Distance to 5, meaning the player needs to keep their distance if they don’t want to alert the enemy.

Once alerted, however, our Basic Enemy should perform the following functions:

- Turn to LOOK at the player.
- Perform a short, startled JUMP (using a force of 200).
- PAUSE to consider their response (for around 1.5 seconds).
- Begin to HUNT the player down.

Luckily for us, we've written corresponding functions for all these actions. Add four new events to the alerted actions panel, and set the functions to be triggered, as shown in [figure 8.9](#)

Figure 8.9: When Alerted, our foe will Look at the player, Jump, Pause for a 1.5 seconds, then start Hunting down the player.

The last function that we trigger that's the AI state to be in 'Hunt the player' mode. This logic has only one required action: keep moving towards the player. In the hunt actions, add event shown in [figure 8.10](#)

Figure 8.10: When Hunting the player, the only thing on the mind of the Basic Foe is moving towards the player at a speed of 5 units per second.

And that's it! You can now press the play button to put this new enemy to the test. If everything was implemented correctly, our Basic Foe will sit there until the player gets close. Once they're alerted, the enemy will jump, turn to the player, then start chasing them after a brief pause.

While this implementation of 'Hunt the Player' logic is fairly barebones, it's *probably* good enough for most of your enemies. When a player is enjoying your game (and attempting to survive) they're not going to spend too much time critiquing the simplicity of your AI.

There will be situations, however, where complex level design will require a more sophisticated AI. When you need more *intelligence* in your AI, Unity has the perfect tool for the job: the **Navigation Mesh**.

NavMesh

In most cases, our simple implementation of ‘Hunt the Player’ will be good enough. However, if your level is a maze of intricate corridors, or has added complexity for other reasons, this basic movement logic will *not* be enough.

This is where Unity’s **Navigation Mesh** becomes necessary. The Navigation Mesh (or NavMesh) Is a generated piece of geometry that can tell your enemy where it can and cannot move. Couple this with integrated Pathfinding - code that figures out how to get from point A to point B - And suddenly your enemy’s hunting skills will receive a considerable upgrade.

Bring up the navigation window by selecting **Window > AI > Navigation**. It should automatically dock itself as a tab within the Inspector window.

The navigation window has 4 categories to work with as mentioned below:

1. Agents

1. This tab lets you specify various character sizes that will be used your game. By default there’s only one - Humanoid. Even if you don’t specify a new character category, all the settings available here can also be tweaked later, directly in the **NavMesh Agent** component.

2. Areas

1. This tab presents a list of the various movement areas cut your levels will consist of. By default there are only three: **Walkable**, **Blocked**, and **Jump**. In most cases, these three will be enough, though adding more is as easy as entering a new name into the list.

3. Bake

1. This is the most important tab in the navigation window. With all your level geometry selected, generating a NavMesh is as easy as pressing the **Bake** button in the lower right corner. *Forgetting this step will result in broken AI Behavior, so always remember to bake your NavMesh after editing a level.*

4. Object

1. The Object tab gives you NavMesh related information about the selected piece of geometry. Use this to specify what Area

category is assigned to each part of your level.

To generate a NavMesh for our enemy to make use of, simply go to the **Bake** tab in the Navigation window and press the Bake button ([figure 8.11](#)). You'll know it worked if a semi-transparent blue square appears, covering a majority of the white ground plane. This blue area is the walkable space that the **NavMesh** will use when **Pathfinding**.

Figure 8.11: The generated NavMesh will appear as a blue, semi-transparent plane over your level geometry.

NavMesh Agents

Before an object will make use of our generated NavMesh, it'll first need to be assigned as a **NavMesh Agent**. Like most systems in Unity, this is done by assigning a component to the object that needs to perform the AI Pathfinding. Select the **EnemyObj_BasicFoe** object and press **Add Component** > **Navigation** > **NavMesh Agent** to apply the necessary component ([figure 8.12](#))

Figure 8.12: The NavMesh Agent component.

While most of these default values are fine, let's make sure the size of the agent matches our enemy. Set the Base Offset to **-0.2** and the height to **0.98**. This will make the agent less 'Humanoid' and more 'Squat, Square Bad Guy' in proportions.

And while using NavMesh is *mostly* automated, we *do* need to add a special AIBrain function to hunt the player using the navigation system.

Open the AIBrain script in Visual Studio, and add this line to the top of the file:

```
using UnityEngine.AI;
```

Anytime you're working with the NavMesh in a script, this **using** call is required. Then, at the bottom of the file, add a new **#region** for NavMesh specific logic, as well as this new function:

```
#region *** NavMesh ***

public void MoveTowardsPlayerUsingNavMesh()

{

    NavMeshAgent agent = GetComponent<NavMeshAgent>();

    if ( agent )

        agent.SetDestination( _playerObject.transform.position );

}

#endregion
```

The function is only three lines long, but they're a powerful three lines. First we grab the Agent component, which we just added to our enemy. If that's valid, we then tell the agent to **SetDestination()** to the player's position. No math. No adjusting positions. The NavMesh does *all* the heavy lifting!

Once the function is ready, go into the AI component and assign Hunt Actions to use our new Nav Mesh logic ([figure 8.13](#))

Figure 8.13: Make sure you assign this new function in the ‘Hunt the Actions’ list.

As always, our last step is to press the **PLAY** button to see how it all works. The enemy should sit there patiently, then become startled when the player gets too close, then start chasing the player as before.

Great to see things are still working, but this test-case doesn't do a great job showcasing the power of the NavMesh. Let's make things a bit harder for our Basic Foe.

Testing our Enemy

New systems often require new test-cases to ensure they're working properly. This process is called **Stress Testing**: creating various scenarios of escalating complexity to make sure your system can handle them.

Since the whole purpose of the NavMesh is to create Enemies that can intelligently **Pathfind** their way around a complex level, we're going to make our level more complex to properly test.

The easiest way to do this is to surround the enemy with Impassable objects, where rushing straight towards the player will find them permanently stuck.

Start by placing (and scaling) three black Cube objects around the Enemy to make a U shape, as in [figure 8.14](#).

Figure 8.14: This U shape will test the intelligence of our NavMesh-enabled enemy.

Now go to the Navigation window and press the Bake button. The NavMesh will update, showing a transparent blue plane over the areas of the scene that are walkable. Note that there is a small white outline around the impassable black cubes. When Pathfinding, the NavMesh system will now avoid those areas and attempt to find a way around the blockade.

Press **Play** to put the Enemy AI to the test. If everything was hooked up correctly, you will see our little red Enemy turn, make its way around the U shape, and slam directly into the player object.

It works! This isn't good news for the player, but it's certainly good news for us. We now have an AI system that leverages power and flexibility.

Conclusion

Our hazards have officially leveled up - tapping into various Unity systems to data-drive intelligent enemies, as well as generating Navigation Meshes to locate and hunt down the player.

But something is missing.

It's not enough to simply have heroes and enemies running around. Combat requires ways for the player to dish out damage, and for enemies to reciprocate. AI is important, but great battles require some great offensive capabilities: swords, guns, magic spells, explosives, and more! We need to get serious about combat, and for that, we'll need new prefabs, components, and design concepts.

It's time to fire up our *Weapon Forge*.

Questions

1. There is considerable overlap in the systems used by Hero and Enemy objects. Which component can you use for both of these object types? What systems will need to be different?
2. What does the term AI stand for?
3. Our AIBrain script manages several AI states: Idle/Default, Alerted, and Hunting for Player. What other states would make for an interesting enemy AI?
4. Think back to your favorite games. Do any of them have an enemy type you enjoyed fighting? With one in mind, take a piece of paper and sketch out the logic states of this enemy. How would you use our current system to implement this enemy logic?
5. Our enemy will give a little startled 'hop' when it sees the player. What does this animation communicate to the player, and why is it important?

6. What is the purpose of Unity's NavMesh system?
7. Currently, when the enemy collides with the player, nothing happens. How could you use a HealthModifier component on the Enemy to fix this?

Key Terms

- **Cannon Fodder:** Enemies that are easy to kill, normally in one shot. These foes aren't much of a challenge, but will make the player feel *strong*.
- **Boss Battles:** Fights that test the various gameplay skills learned by the player. A good Boss Fight will make the player feel strong AND smart.
- **Artificial Intelligence:** A system that analyzes the data of a scene and makes smart decisions about what to do.
- **States:** The current directive of an AI object. Is it patrolling an area? Hunting down the player? Fleeing from danger? The logic of the current State is what drives the actions of an AI object.
- **Data-Driven AI:** Using the Unity Editor to tell the AI how to act, and react, to the player. Our implementation of 'Data-Driven AI' will make use of the data type *UnityEvent*.
- **UnityEvents:** A list of function calls, assigned to be triggered on specific objects, that can be data-driven through the editor and invoked in code. A great way to create flexible systems in Unity.
- **Invoke:** A function call to UnityEvents that triggers the assigned functions of the associated object with the appropriate parameters (all of which has been data-driven using the Unity Editor).
- **NavMesh:** A generated 'Navigation Mesh' that Unity can use to help objects that need to make use of Pathfinding.
- **Pathfinding:** When the AI uses a mesh, tree, or other data structure to intelligently find its way around an environment.
- **#region / #endregion:** Tags in your scripts that define a block of code. These blocks can then be consolidated to keep your code tidy.

CHAPTER 9

Forging Your Weapon System

When creating a Video Game, designers will often use of **Verbs** as a way to express what the player can do. *Running, Jumping, Talking, Planning, Searching and Building* each express a gameplay system that shapes the player's experience.

Combat-focused design verbs are a great way to get a player's blood pumping. *Shooting, Dodging, Reloading, Stabbing, Slashing, Blocking*. Creating a game with a deep and engaging combat system immediately raises the player's adrenalin levels and hooks them into your game in a visceral way. There are no bigger stakes than 'kill or be killed'.

And while 'jumping on enemies heads' works for platformers, we're going to look at the other way to dispatch your foes: **Weapons**. With a flexible Weapon system, we can create a wide variety of cool weapons to use when fighting our way to victory.

Structure

- Weapon Categories
- Weapon Sandbox
- Melee Combat
- Equipping the Hero
- Projectile Weapons
- Knockback Effect
- Attack Animations
- Weapon Trails
- Game Design: Advanced Combat

Objectives

After studying this unit, you will have built several new battle-related objects and scripts, each teaching a different aspect of video-game combat. You have a melee weapon that will hit enemies close to the player, and a projectile weapon that can target enemies from a distance.

You'll also have a new system for triggering specific animations for your attacks - changing your animations based on the type of weapon being held.

By the end, you'll have both practical training in combat systems, and several advanced design concepts to use when planning combat in your own projects.

Sticks and Stones

Ever since our ancestors could swing a stick or throw a rock, weapons could be divided into two major categories: **Melee** and **Projectile**.

The same is true in video games - most weapons fall into these two categories, each with their own specific pros and cons:

	Pros	Cons
Melee	<ul style="list-style-type: none">Easier to land a hit.Physical Strength and Dexterity can improve Dealt Damage	<ul style="list-style-type: none">Closer to danger.
Projectile	<ul style="list-style-type: none">Safer to Attack from a DistancePhysical Strength not typically a Factor (can be used Universally)	<ul style="list-style-type: none">Accuracy is more important (easier to miss the target).Limited Ammunition

Table 9.1: The Pros and Cons of Melee vs. Projectile combat.

Most games have some mixture of both. First Person Shooters tend to be projectile-heavy, but will give the player a single melee option if they ever run out of ammo. Fantasy games will have a large assortment of Melee weapons (Swords, Spears, Maces, Hammers, etc) but will include Bow-and-Arrow weapons and magic to give players plenty of combat options.

Weapon Sandbox

To keep our project organized, let's make another sandbox scene - this time for making our Weapons. Create a new scene and name it **WeaponSandbox**.

Use the Place a new 3D Object > Plane to act as our ground. Assign the *matBlack* material to it, so our objects are clearly visible. Set its position and scale as listed in [table 9.2](#)

Plane						
Position	X	0	Y	0	Z	0
Scale	X	3	Y	3	Z	3

Table 9.2: The position and scale of our Plane “Ground” object.

You'll also want to drag a Sphere Hero prefab into the scene. By having the hero visible, we can build weapons that fit within the scale we've set. Otherwise, it's easy to build objects that are too large, or too small, for the characters that will be holding them. Once places, set the position of the player to match [table 9.3](#)

PlayerObj_Sphere						
Position	X	-3	Y	0.5	Z	0

Table 9.3: The position of our hero sphere.

Like all sandboxes, our WeaponSandbox will be an empty scene that we can build our weapons in. Once the area resembles [figure 9.1](#), you're ready to start forging some weapons!

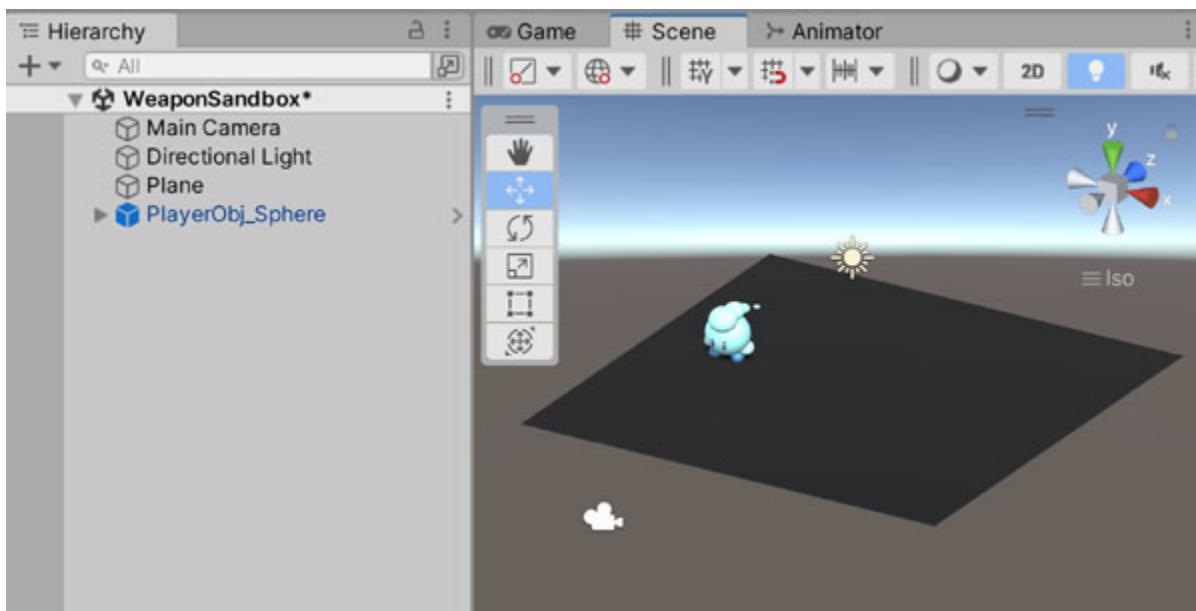


Figure 9.1: Your WeaponSandbox should look like this.

Let's start by creating some prototype weapons models. We'll start by using the 3D Cube objects to create our Melee weapon: the *Sword*.

Make an empty GameObject named **WeaponObj_Sword**, add the following child objects, and set their transforms according to the following values ([Table 9.4](#)).

- 3D Object > Cube named **SwordBlade**
- 3D Object > Cube named **SwordPoint**
- 3D Object > Cube named **SwordGuard**
- 3D Object > Cube named **SwordHilt**

SwordBlade						
Position	X	0	Y	0.6	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	0.2	Y	1	Z	0.1

SwordPoint						
Position	X	0	Y	1.1	Z	0
Rotation	X	0	Y	0	Z	45
Scale	X	0.2	Y	0.2	Z	0.1

SwordGuard						
Position	X	0	Y	0.15	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	0.5	Y	0.1	Z	0.15

SwordHilt						
Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	0.15	Y	0.4	Z	0.15

Table 9.4: The transform data for our Sword Prototype.

At this point you can have some fun with coloring the sword. I personally made the hilt blue and blade silver, but you can create your own materials to color it as you see fit ([figure 9.2](#))

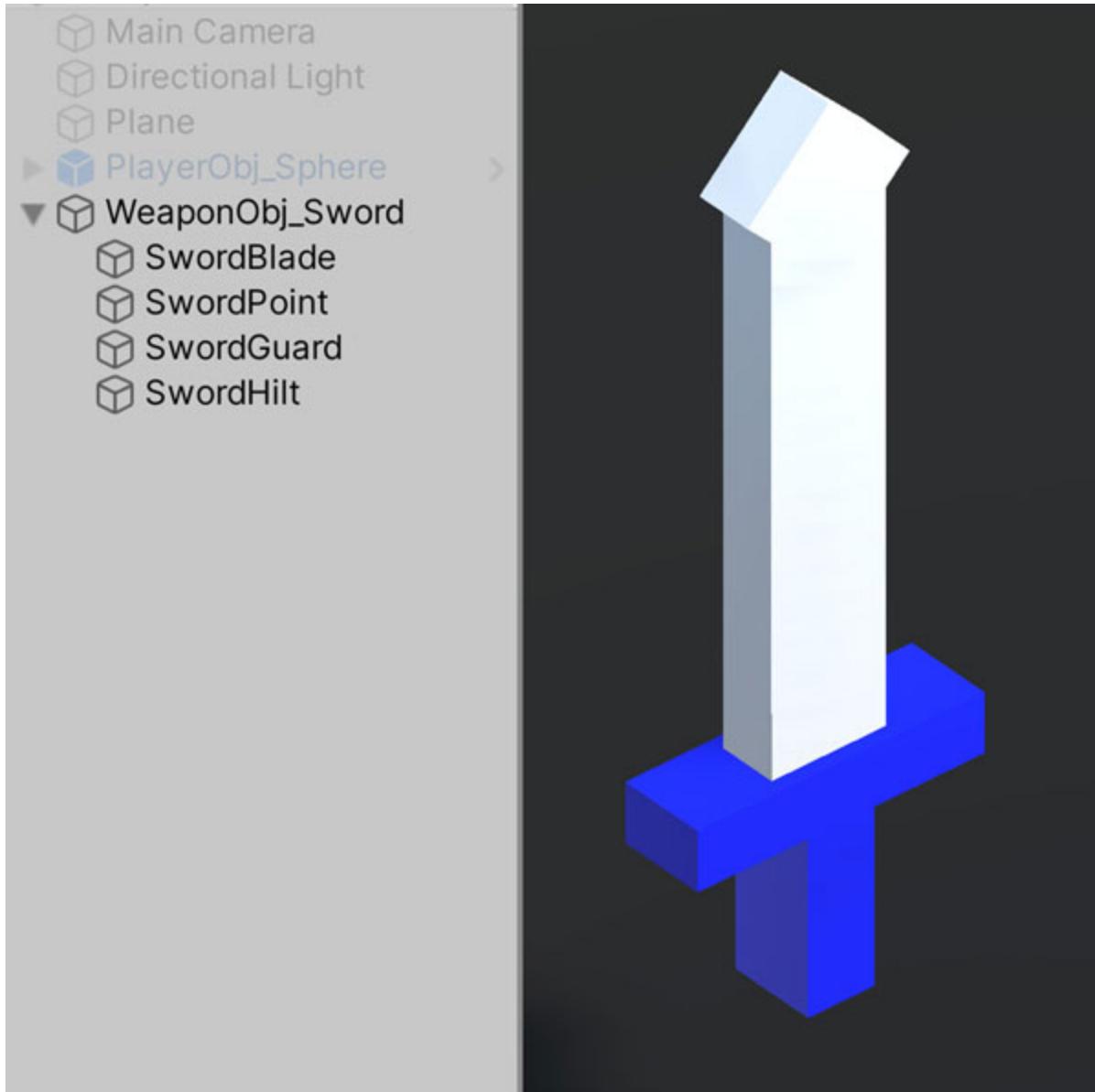


Figure 9.2: Your Prototype Sword object!

When using the included 3D Objects (Cubes, Spheres, Cylinders, etc) it's often good practice to disable the pre-existing colliders on those objects. When stacked together, these colliders will cause unexpected physics oddities. Select all the cubes you've placed, and in the Inspector window, right click on the Box Collider entries and select Remove Component for all of them, as shown in [figure 9.3](#)

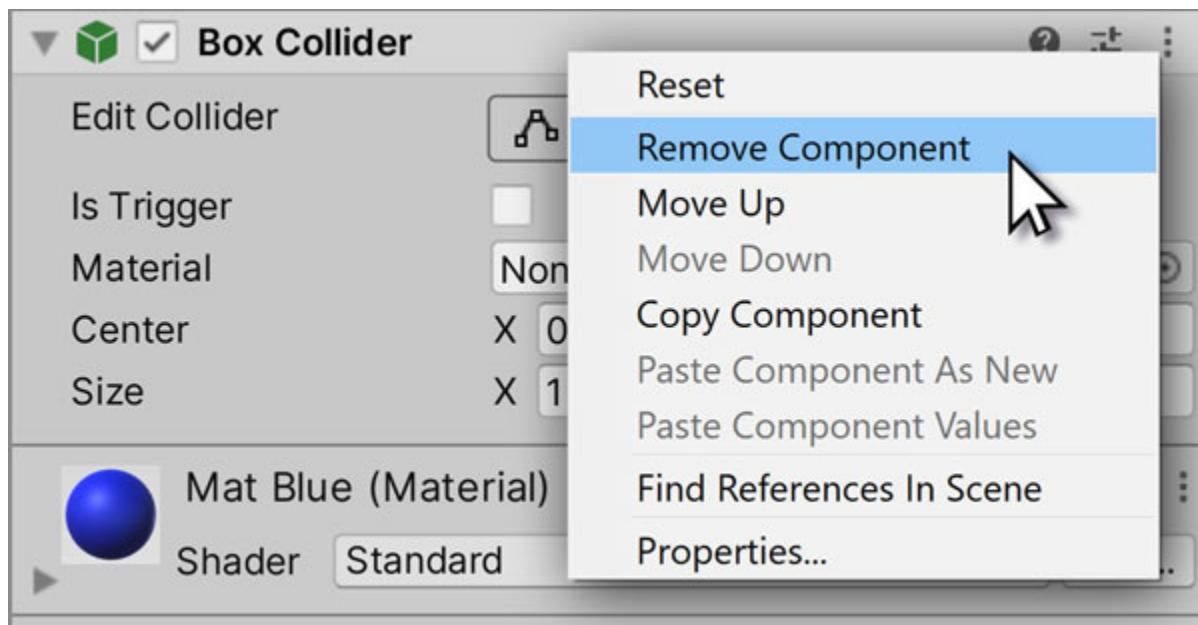


Figure 9.3: When using 3d Primitives to make prototype objects, you'll often need to remove the collider components.

We'll replace all those separate colliders with a single Capsule Collider on the root object. Once added, set the Capsule Collider data to match what is shown in [figure 9.4](#)

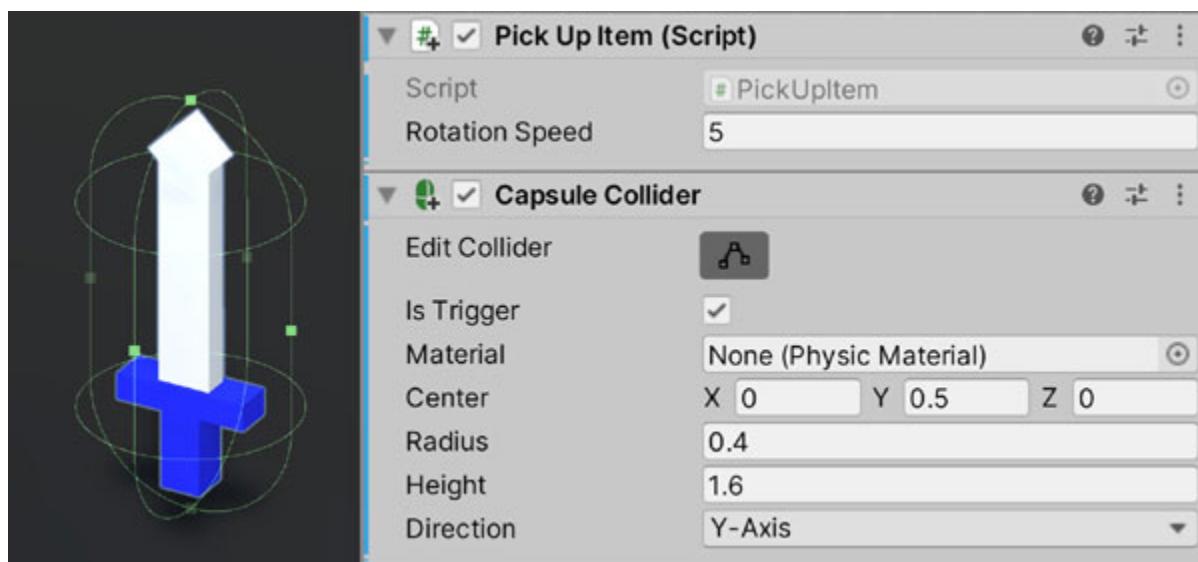


Figure 9.4: Our Sword will also need a Capsule Collider, Pick Up Item, and Rigid Body script.

Besides the Capsule Collider, you'll also need to add the following scripts to the Sword...

- **Rigid Body:** Since the sword will be colliding with enemies, it will need to have the proper physics scripts in place to fire off the `onTriggerEnter()` functions. Note: Make sure Gravity is disabled!
- **Pick-Up Item:** This is the script we made for coins, but will be expanded to allow us to pick up, and equip, the Weapons we make.

Another component that we'll want to add is a new Weapon script ([figure 9.5](#)). You can attach this to the sword using a component **Stub**: an empty class (or asset) that is simply there as a placeholder. Again, don't worry about adding functionality. We'll be doing that later in the chapter.

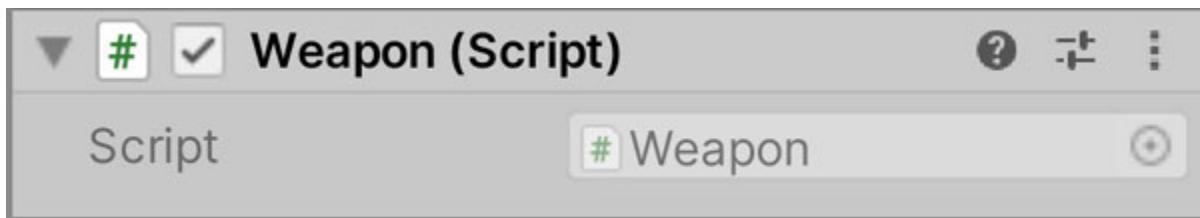


Figure 9.5: A stubbed-in Weapon script.

At this point, you may want to perform a quick sanity check on the pivot point of our Sword. When creating Melee weapons, you'll want its point of rotation to be wherever it would be held by the player object (or enemies). In this case, we want to make sure the pivot point is on the hilt. Select **WeaponObj_Sword** in the Hierarchy window, select the Rotation tool, and make sure it rotates around its hilt ([figure 9.6](#))

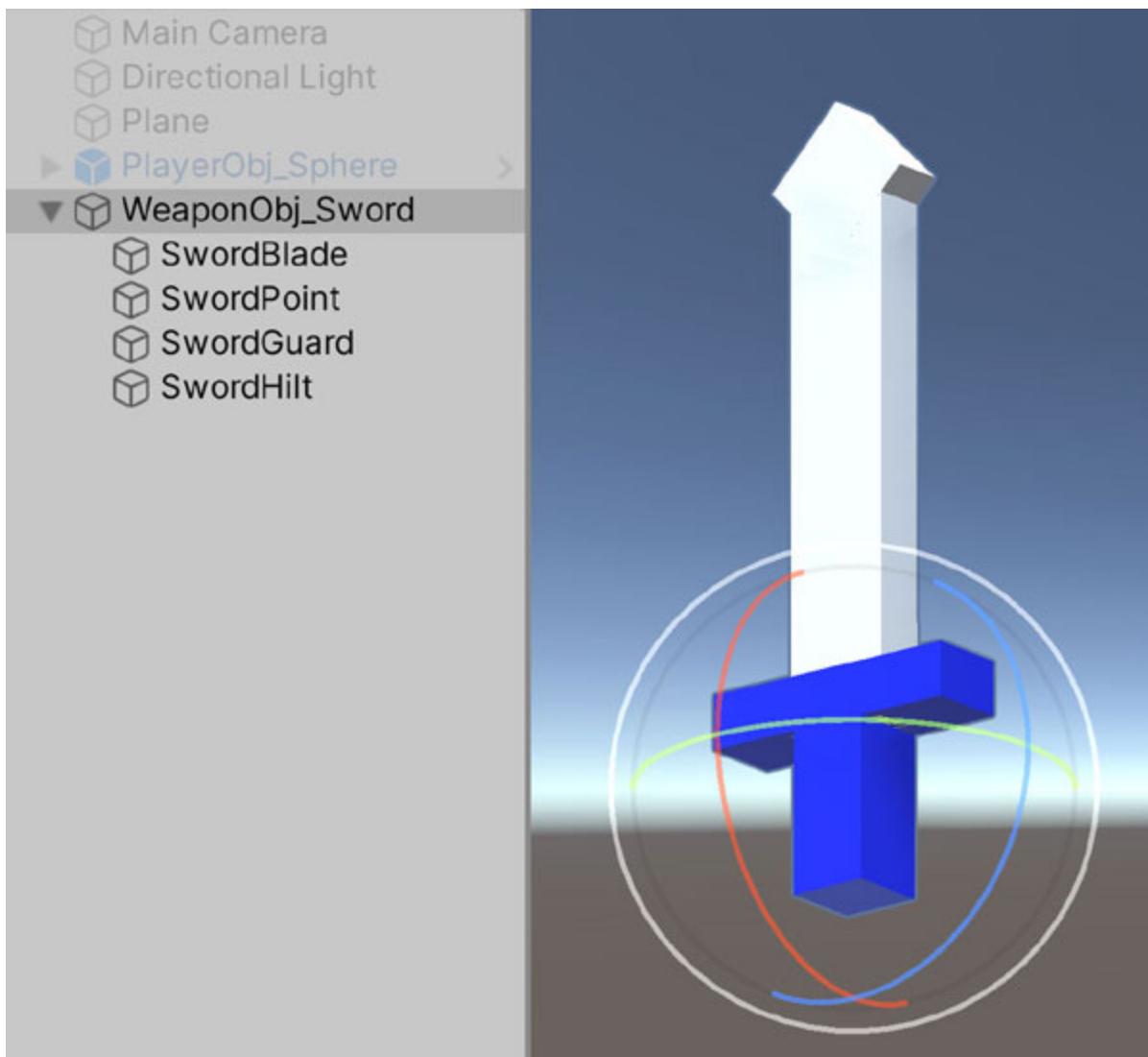


Figure 9.6: Make sure your melee weapon rotates around the proper pivot point. For our sword object, that should be the hilt.

Our last step is to drag our sword into the prefabs folder. With that, we now have our Melee weapon prototype made!

Now it's time to make our Projectile weapon.

Create an empty GameObject named **WeaponObj_Blaster**, add these following child objects, then set their transform data as laid out in [Table 9.5](#).

- 3D Object > Cube named **Grip**
- 3D Object > Cube named **Body**
- 3D Object > Cube named **Barrel**
- 3D Object > Cube named **PlasmaCoil**

- 3D Object > Cylinder named **Core**

WeaponObj_Blaster						
Position	X	0	Y	0.5	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	0.1
Grip						
Position	X	0	Y	0.1	Z	0
Rotation	X	-30	Y	0	Z	0
Scale	X	0.15	Y	0.4	Z	0.2
Body						
Position	X	0	Y	0.3	Z	-0.2
Rotation	X	0	Y	0	Z	0
Scale	X	0.22	Y	0.32	Z	0.8
Barrel						
Position	X	0	Y	0.3	Z	-0.7
Rotation	X	0	Y	0	Z	0
Scale	X	0.1	Y	0.4	Z	1.3
PlasmaCoil						
Position	X	0	Y	0.32	Z	-0.7
Rotation	X	0	Y	0	Z	0
Scale	X	0.17	Y	0.1	Z	1.4
Core						
Position	X	0	Y	0.3	Z	0
Rotation	X	0	Y	0	Z	90
Scale	X	0.3	Y	0.15	Z	0.3

Table 9.5: The transform data for our ‘Blaster’ Prototype.

Once all the objects are added and positioned, you should have an object that looks like the model in [figure 9.7](#). Use different materials to color your blaster how you want.

Just like with the Sword, remove all the **Box Collider** components and instead add a **Capsule Collider** that envelops the blaster object. You’ll also need to add a **Pick-Up Item** script and a **Weapon** component.

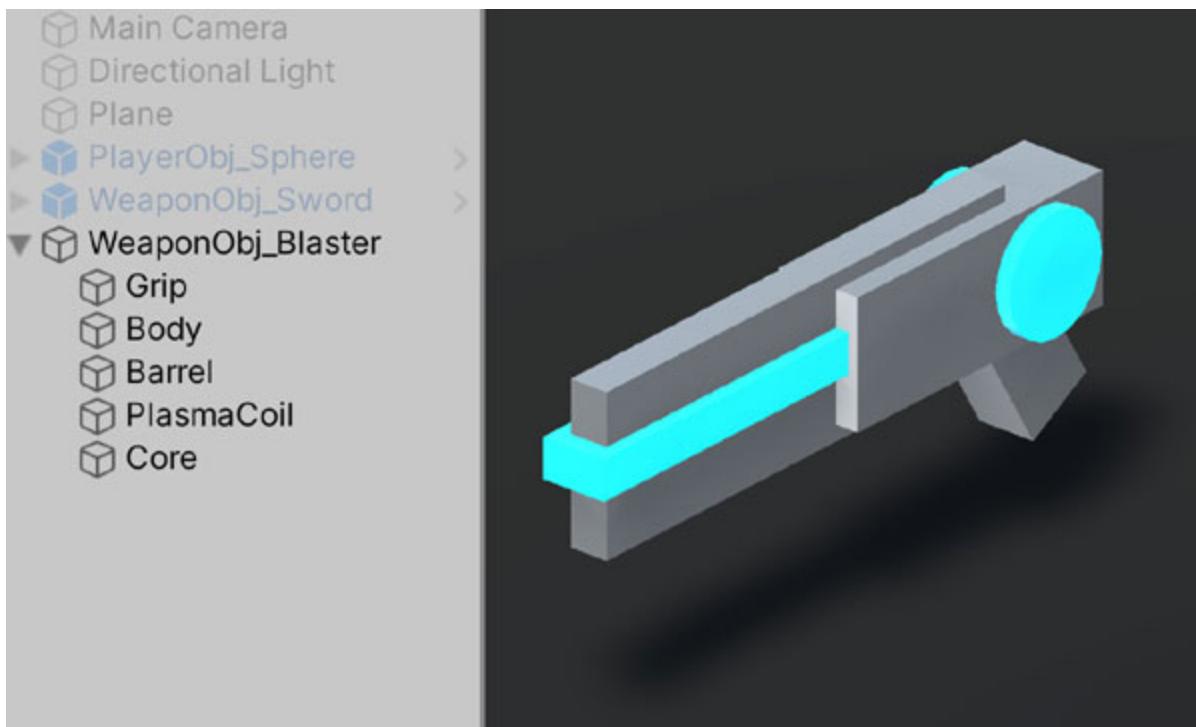


Figure 9.7: Our finalized Blaster Prototype.

Drag our projectile prototype into the Prefabs folder to complete our first step, with our second step being the actual *equipping* of the weapons by our hero.

Equipping Your Weapon

In modern games, there's the expectation that the player will be able to see the equipped weapon in the hands of their character. This visually informs the player what to expect when they press the 'attack' button, and gives them the opportunity to swap out weapons as necessary (if Inventory Management is a part of the gameplay loop).

For us, this means that we need to add an empty game object on our hero to use as a **Weapon Locator**. A Locator lets us know where on an object an equipped object should be attached.

Open the hero prefab in the Prefab editor. Find the Arm_R object in the hierarchy. Create an empty GameObject as a child of Arm_R. Name it "WEAPON_LOC" and set its position and rotation (see [Table 9.6](#)).

WEAPON_LOC						
Position	X	-3	Y	0.5	Z	0
Rotation	X	-90	Y	0	Z	0

Table 9.6: The position and rotation of the WEAPON_LOC object. This is where we'll be attaching the Sword and Blaster when they're equipped by the hero.

If you set all the data correctly, the empty Locator object should appear in the hand of our hero, with the position gizmo visible similar to how it's shown in [figure 9.8](#)

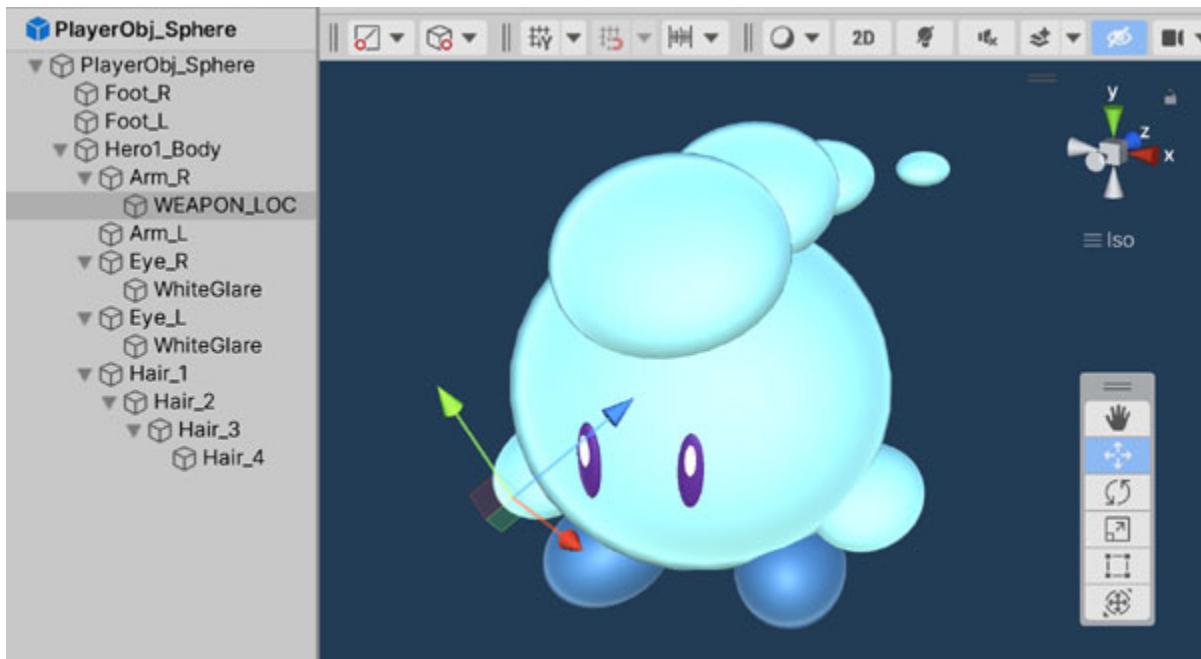


Figure 9.8: The locator sub-object of Arm_R. This is where the Sword and Blaster will be placed when we equip them.

We'll now have to add some code to make our weapons equip properly. Open the **Pick Up Item** script and add this code to the top of the `onPickedUp()` function...

```
if ( GetComponent<Weapon>() != null )
{
    PlayerController player =
        whoPickedUp.GetComponent<PlayerController>();
    if ( player != null )
    {
        // player has picked up a weapon
```

```

player.EquipWeapon(GetComponent<Weapon>());
// disable this 'pickup' script
enabled = false;
}
return;
}

```

This code performs the following logic...

- Did the player just collide with a PickUp Item?
- If so, and that item a ‘Weapon’?
- If we DID collide with a ‘Weapon’ that can be picked up, then tell the player object to equip this item as it’s current weapon.
- We also disable this Pick Up Item script, since you can’t pick up an item that you’re already carrying.

We now bring up the **Player Controller** script, so we can add this Equip Weapon functionality. First, add a member to store the weapon that’s currently equipped.

```
[SerializeField, Tooltip("This player's equipped Weapon.")]
private Weapon _weaponEquipped = null;
```

Then, at the bottom of this script, add this new region of code...

```
#region *** Weapons ***
public void EquipWeapon( Weapon weapon )
{
    _weaponEquipped = weapon;
    weapon.SetParent(GameObject.Find("WEAPON_LOC"));
}
#endregion
```

This will store the incoming weapon as our equipped weapon, while also attaching the weapon to the player object.

The Weapon Component

While the above code dealt with the equipping of weapons, the Weapon component is where the real work is done. Open **Weapon.cs** and add the following code...

```
public class Weapon : MonoBehaviour
{
    GameObject _attachmentParent;

    [SerializeField, Tooltip("Pause movement after an attack?")]
    float _pauseMovementMax = 1.0f;
    float _pauseMovementTimer = 0.0f;

    // Update is called once per frame
    void Update()
    {
        if (_pauseMovementTimer > 0f)
        {
            _pauseMovementTimer -= Time.deltaTime;
            Return; // temp
        }
        if (_attachmentParent)
        {
            // reposition the weapon gfx in relation to whoever
            // has this weapon equipped
            Transform tr = _attachmentParent.transform;
            transform.position = tr.position;
            transform.localEulerAngles = tr.eulerAngles;
        }
    }

    public void SetAttachmentParent( GameObject parentObj )
    {
        _attachmentParent = parentObj;
    }

    public bool IsMovementPaused()
    {
        return (bool)( _pauseMovementTimer > 0 );
    }

    public void onAttack( Vector3 facing )
    {
        // todo: handle the 'swing sword' logic
        // todo: handle the 'fire blaster' logic
    }
}
```

```
    }  
}
```

There are currently two main features of this code.

1. There is logic that handles the ‘pausing’ of movement. Most weapons, when used, will result in a period where the weapon cannot be fired again. This logic will take care of that functionality.
2. The update call will make sure that a weapon will always follow the position of it’s parent object. This ensures the weapon appears in the hand of the character that has equipped it.

Now press **Play**. Grab the sword, and it should now attach itself to the hero ([figure 9.9](#))!

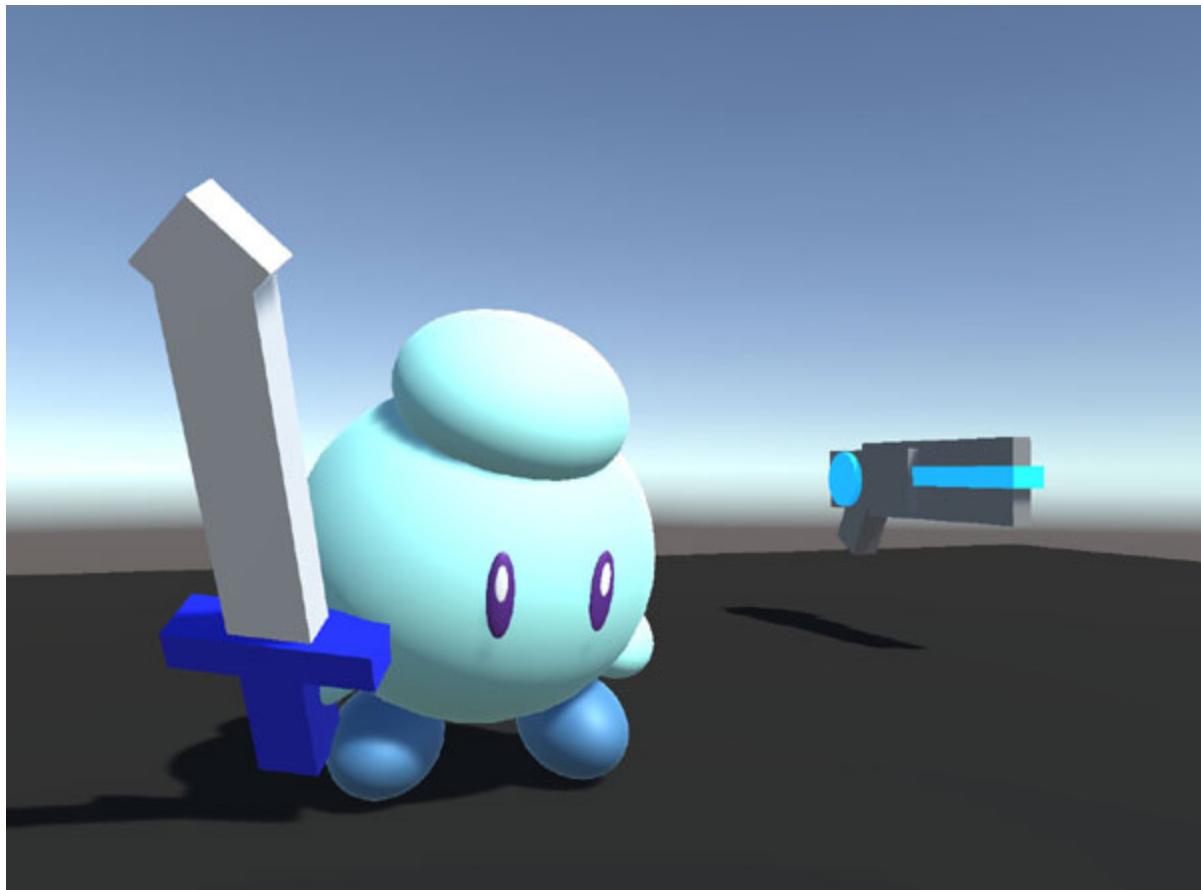


Figure 9.9: Weapon in hand and ready to fight!

With our weapon in hand, it’s time to move onto the fun stuff - combat!

Basics of Melee Combat

Melee combat puts the player within arm's reach of their opponent, forcing them to strike fast and move quick. When they press the attack button, the response needs to almost be instantaneous. Lag between input and on-screen action is a quick way to kill the thrill of the fight.

With that in mind, our first step should be to grab that input and trigger an attack. Open the Player Controller script and update the 'fire weapon' code to look like this...

```
// fire the weapon?  
if ( Input.GetKeyDown(KeyCode.Return) )  
{  
    if (_weaponEquipped)  
        _weaponEquipped.onAttack(_curFacing);
```

The two lines we're adding will check to see if a weapon is equipped. If so, let that weapon know that it should perform its 'attack' functionality.

Go to the **Weapon** script next, and in the **onAttack()** function, add this logic...

```
// handle the 'swing sword' logic  
transform.position = transform.position + facing;  
transform.Rotate(new Vector3(45, -90f, 90f));  
_pauseMovementTimer = _pauseMovementMax;
```

This code moves and rotates the sword in front of the player object and pauses movement for a small amount of time.

Speaking of pausing movement, we'll need to add these lines to the top of the **Update()** function back in Player Controller.

```
void Update()  
{  
    // pause movement if we've recently attacked  
    if (_weaponEquipped && _weaponEquipped.IsMovementPaused())  
    {  
        _rigidBody.velocity = Vector3.zero;  
        return;  
    }
```

The player will be locked in position when they attack. Now press **Play**, grab a Sword, and use the **Enter** key to attack. As you can see, our attack

is...underwhelming. Such a pathetic thrust will have to be overhauled by the end of the chapter, but for now, it gets the job done.

The next important step is to make the Sword deal damage to enemies!

Open the Sword prefab and add a **Health Modifier** component ([figure 9.10](#)). Set **Change Health** to **-10**, and make it target **Enemies**.

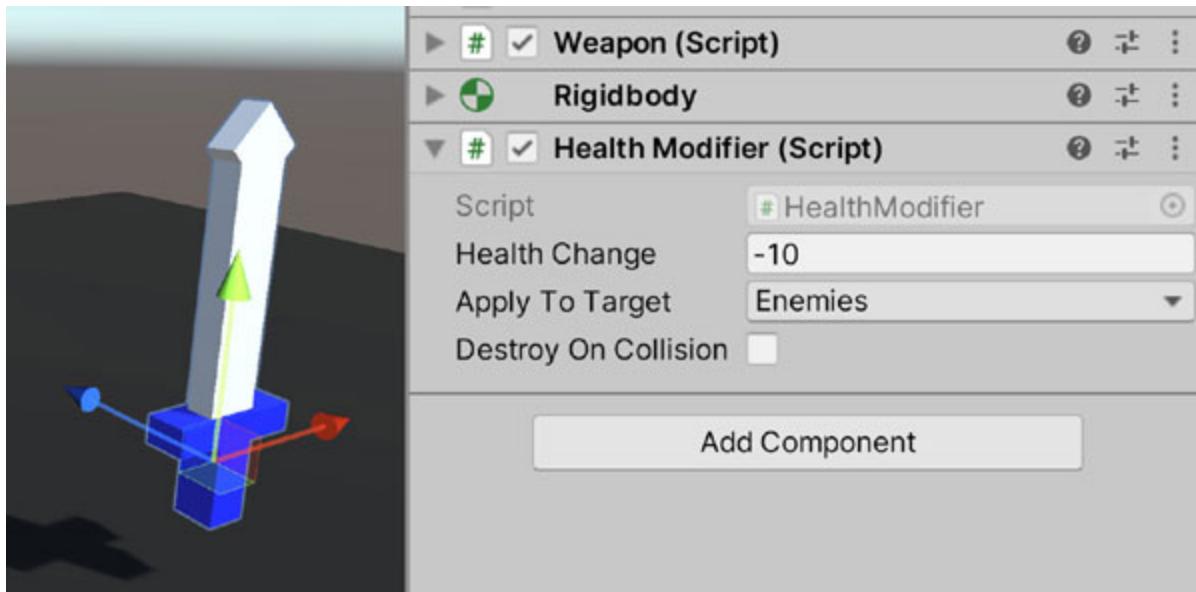


Figure 9.10: Dealing damage is as easy as attaching a ‘Health Modifier’ component to our Sword!

Place some Spikes in your Scene and attack them. Note how they die when attacked with the sword.

With the Melee weapon functioning properly, it’s time to set up our Projectile Weapon!

Basics of Projectile Weaponry

Projectile weapons let the player fight from afar. To do this, they’ll need to be assigned a projectile that gets spawned when the player presses the attack button.

Let’s do that now by adding a ‘bullet to spawn’ member to the Weapon component.

```
[SerializeField, Tooltip("The bullet projectile to fire.")]  
private GameObject _bulletToSpawn;
```

Our weapons can now have a prefab assigned to it that can be spawned on attack. While still in the Weapon script, go to the `onAttack()` function and

have the attack spawn the assigned bullet.

```
// handle the 'projectile weapon' logic
if (_bulletToSpawn)
{
    GameObject newBullet = Instantiate( _bulletToSpawn,
    transform.position,
    Quaternion.identity);

    Bullet bullet = newBullet.GetComponent<Bullet>();
    if (bullet)
    {
        bullet.SetDirection(new Vector3(facing.x, 0f, facing.z));
    }
}
```

Now go back into the Unity Editor, open the Blaster prefab, and assign a bullet to spawn ([figure 9.11](#)).

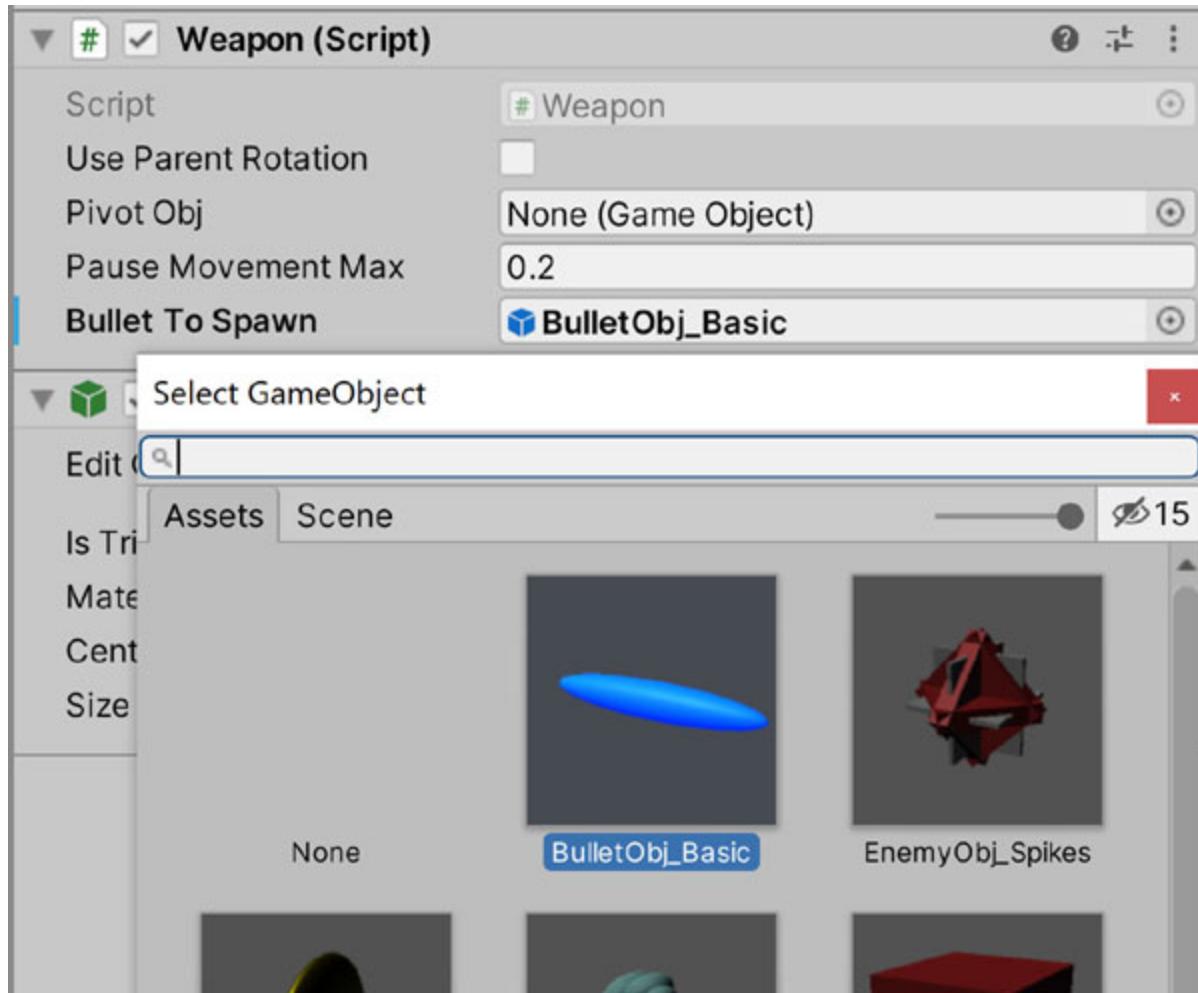


Figure 9.11: Assign the bullet prefab we want to spawn when firing this weapon.

Now, when you pick up our ‘Blaster’ weapon and fire it, the bullet we’ve assigned will be spawned in the direction the player is facing!

The ‘Knockback’ Effect

It’s important to show the player when an object has taken damage. One way to make attacks feel better is by using our physics system to ‘knock-back’ a damaged enemy. This helps to sell the impact of an attack, and is fairly easy within our physics system.

The best place to add this is directly in the Health Modifier - giving some extra umph when we lower the health of our target.

Start by adding a Knockback Force member to the top of our Health Modifier component.

```
[SerializeField, Tooltip("Knockback force when this damage is applied.")]  
float _knockbackForce = 0f;
```

Next, we need to add a physics force to targets that have a Rigid Body component. Place this code in the Health Modifier script, within the `onTriggerStay()` function, after damage has been applied.

```
// apply knockback when damage is dealt  
if (_healthChange < 0 && _knockbackForce != 0)  
{  
    Rigidbody rb = hitObj.GetComponent<Rigidbody>();  
    rb?.AddExplosionForce(_knockbackForce, transform.position,  
    10f);  
}
```

Now open the Bullet prefab and change the knockback to 400. Place enemy prefabs and watch them fly when attacked! Very satisfying!

Animating our Attacks

Let's revisit our Melee attack for a moment. One of the most important aspects of melee combat is the Motion of the attack. If a player equipped a Dagger, Sword, or Giant Hammer, they would expect these weapons to move in different ways and at different speeds.

Some games will drive this movement with code. Our current implementation takes this route - moving the sword forward and rotating it for half a second. This *can* be acceptable, but we should also consider the second, more robust method of making your weapons move, and that is through *Animation*.

Luckily, you've already learned about Unity's animation system, so you know how you can make different animation clips that can be played on an object. For attacks, all we need to do is create a new 'Swing Sword' animation, then play that animation when `onAttack` is called.

It's easy. It's flexible. And it looks considerably better than the code-drive alternative.

Begin by adding some temporary weapons as children of the hero's `WEAPON_LOC` object. These will allow us to preview the rotation and position of these weapons over the course of the animation ([figure 9.12](#)).

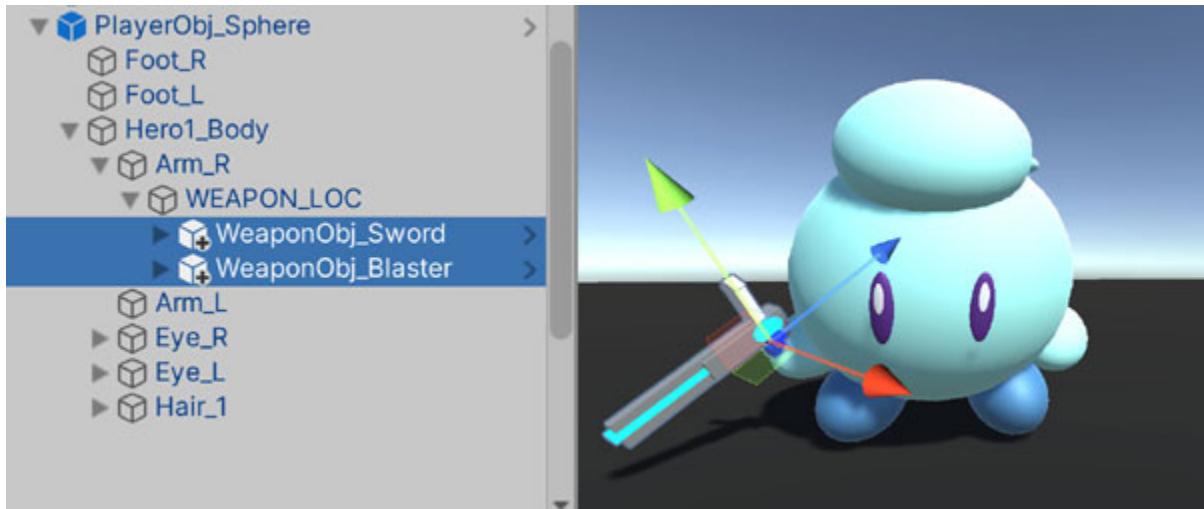


Figure 9.12: Placing temporary weapons into the locator will help us make more impactful attack animations.

Next, bring up the Animation window and create a new clip called “SphereHero_SwingSword_1”. Also, use the small red circle button to enable ‘Keyframe Recording Mode’, then set the following object keyframes:

Hero1_Body							
Frame 0				Frame 10			
Pos	X	0	Y	0.1	Z	0.2	
Rot	X	10	Y	60	Z	15	
Arm_R							
Frame 0				Frame 20			
Rot	X	18	Y	100	Z	0	
				Rot	X	7	Y
						50	Z
						-20	
Foot_L							
Frame 0				Frame 10			
Pos	X	0.2	Y	-0.34	Z	-0.18	
Rot	X	-12	Y	0	Z	-22	
Foot_R							

Frame 0						
Rot	X	-0.18	Y	0.34	Z	0.2
Rot	X	-18	Y	40	Z	-6
Frame 10						
Rot	X	-0.1	Y	-0.35	Z	-0.32
Rot	X	-17	Y	-10	Z	10

Table 9.7: We only need two keyframes to get the effect we're looking for.

When you use the ‘Play Animation’ button (in the Animation Panel), you should now see our Spherical Hero swing his sword around and forward, looking considerably more polished than our initial code-based attempt at a ‘Sword Swing’ ([figure 9.13](#)).

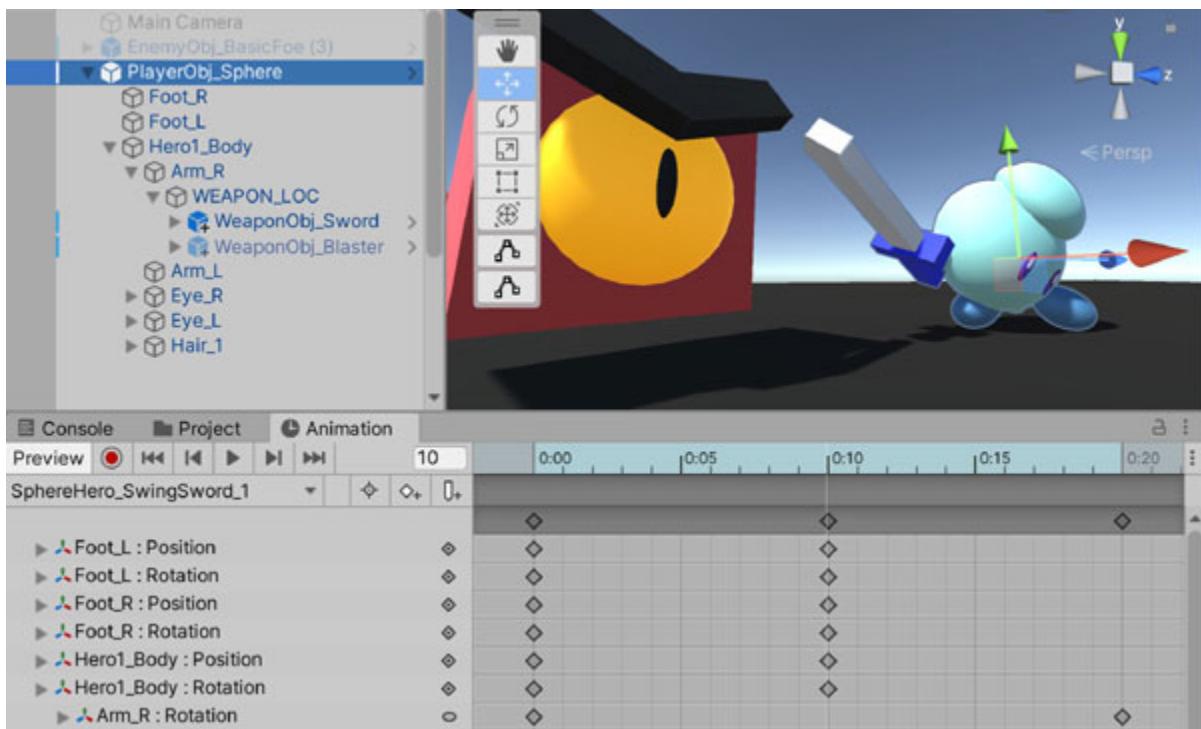


Figure 9.13: By animating our Sword Swing, we can ensure the movement of the character matches the movement of the equipped weapon.

Now, open the ‘Animator’ panel ([figure 9.14](#)), and make let’s rename the clip to something a bit more friendly: **SwordAttack01**.

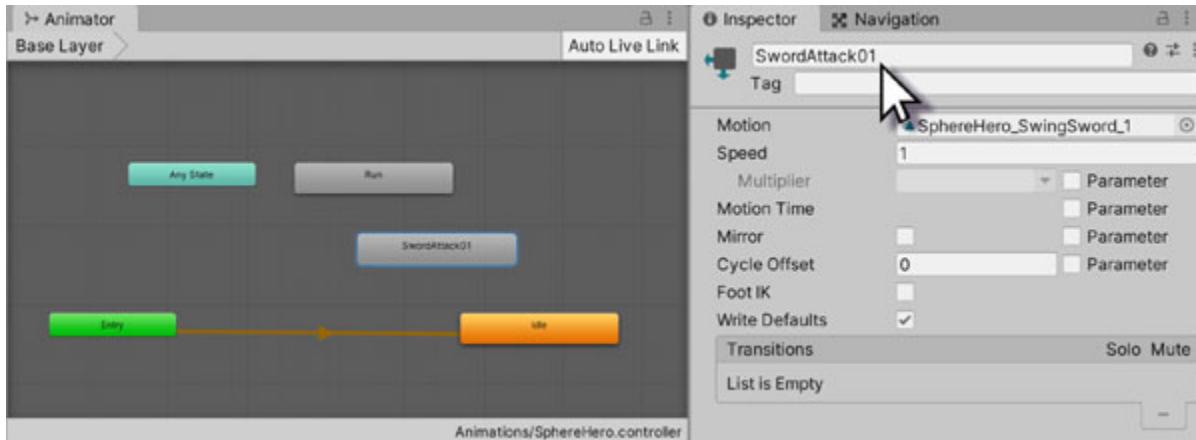


Figure 9.14: Rename our new animation *SwordAttack01*. Our goal: any object that equips a sword, and has a ‘*SwordAttack01*’ animation clip, will play the proper animation when the attack button is pressed.

With the animation complete, we now need to go into the Weapon script and add an ‘Attack Animation’ member...

```
[SerializeField, Tooltip("Animation to play when attacking.")]
public string _attackAnim = "SwordAttack01";
```

This string will let us data-drive the animation that gets used when a weapon is used. This simple system would allow us to create the Dagger, Sword, and Giant Hammer weapon variety we talked about earlier.

Staying in the Weapon script, let’s also remove some code. In `Update()`, you’ll want to comment out the `RETURN` line after we adjust the pause movement timer.

```
_pauseMovementTimer -= Time.deltaTime;
// return;
```

This will allow the weapon attachment code to keep the sword aligned with our animated hero.

We’ll also want to clean up the `onAttack()` function. Find the comment about ‘handling the Swing Sword logic’. Remove those 4 lines, and replace them with this...

```
// pause movement
_pauseMovementTimer = _pauseMovementMax;
```

The previous code moved the sword object using code. This new solution - using animations - handles the positioning directly in the animation clip.

Our final step is to open the Player Controller script. In the `Update()` function, find where we call `onAttack()`. Right after that, add these lines to play the proper animation...

```
// assign animation  
if ( _weaponEquipped._attackAnim != "" )  
    _myAnimator.Play( _weaponEquipped._attackAnim );
```

Our hero will now swing his sword as you'd expect from a modern Video Game hero! Press Play, grab the sword, and start swinging!

Weapon Trails

We talked earlier about the importance of visually informing the player when an attack is successful. We added a Knockback Effect to help with this, but there are countless other ways to present combat information. Sounds and particle effects we'll get to in another chapter, but right now, we should take a moment to quickly implement Trails.

While they don't necessarily show the impact of a weapon, they can show the shape and movement of the attack, which is arguably just as important as an attack's strength.

Bring up your Sword's prefab, find the `SwordBlade` sub-object, and add a new component called **Trail Renderer** ([figure 9.15](#))

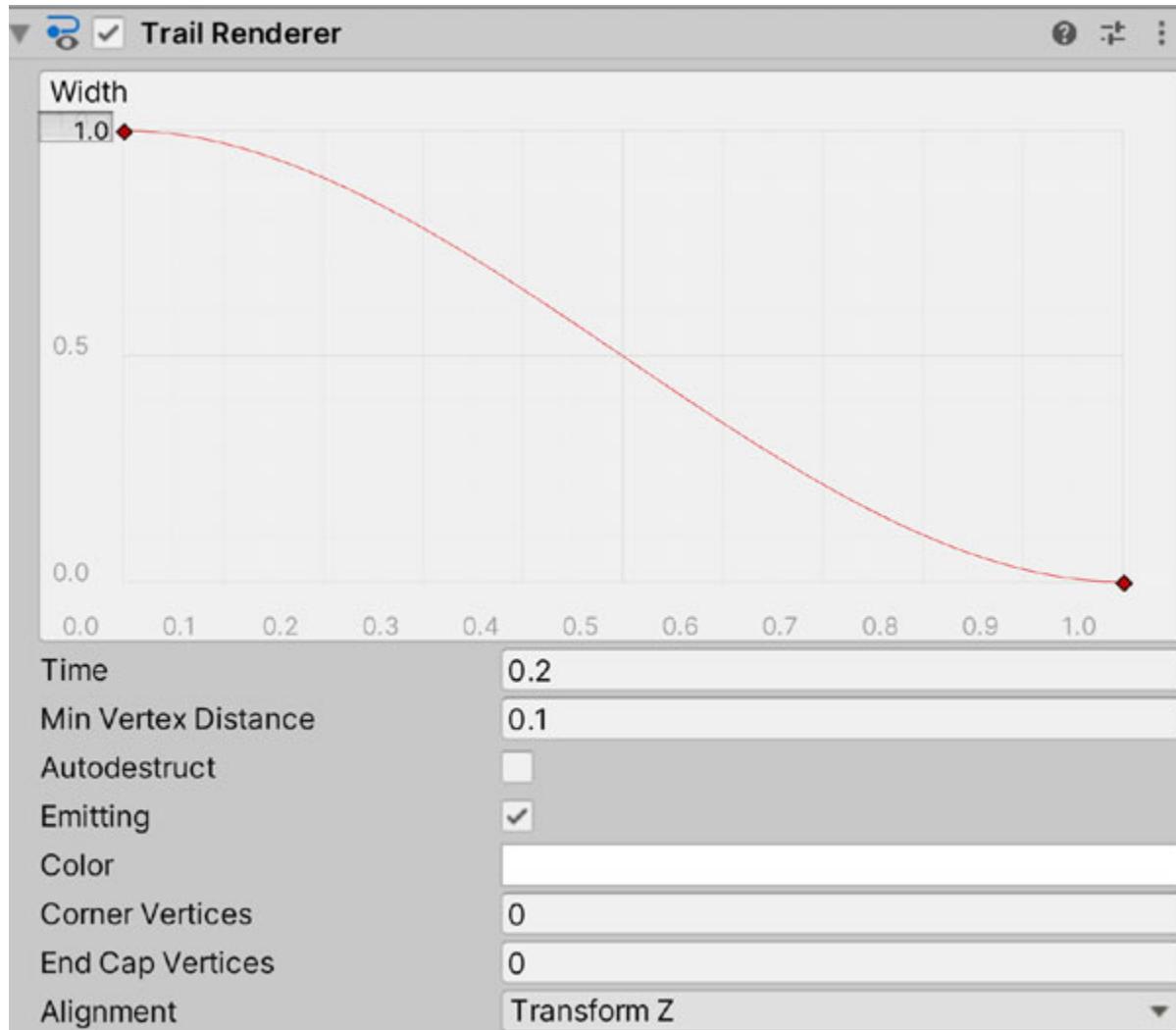


Figure 9.15: Our Sword's Trail Renderer component.

On the ‘Width’ grid, double-click the red line to add another point on the curve. Drag that point down into the lower right corner (it should look like the curve in [Figure 9.15](#)).

Next, set the **Time** to **0.2** seconds. If you press play and grab the sword, you will now have a pink trail following your sword. When you attack, a trail will show the Area of Attack for that weapon.

And don’t worry - in a later chapter we’ll fix that *Pink* color.

Game Design: Advanced Combat

Combat is one area where you can create some fun and intricate systems for making battles feel *great*. While the above components and prefabs will get

you started, lets dive into advanced combat mechanics that you may want to incorporate into your own game design...

- **Ammo Systems**

- Many games with projectile weapons will offset their key advantage (damaging enemies from a safe distance) with one key weakness: limited **Ammo**. By limiting the use of Guns, Bow+Arrows, and Explosives, designers create a great risk/reward situation where players must strategize before planning an attack. Ammo can also come in the form of mana, if you're making a fantasy game.

- **Damage Types**

- You can push the ‘strategic’ angle of combat by including **Damage Types**. These are categories of attack – Cutting, Bashing, Fire, Lightning, etc. – where players, enemies, and equipment have strengths and weaknesses against specific types.

- **Charge Attacks**

- Holding a button to unleash a stronger attack is a great Risk/Reward setup – especially in a game where there are many enemies to dodge. Do you attack as much as possible to thin the enemies’ ranks, or do you hold the ‘fire’ button to build up (and unleash) a devastating **Charge Attack**?

- **Loadouts**

- Another way to add some strategy to your game is by using weapon **Loadouts** – a phase where players must pick the weapons they’ll be using for a battle. This is often done through an inventory system, where players can only hold a limited number of weapons and have to manage the best one for a given situation.

- **Dropped Weapons**

- Players love having *tons* of weapons to choose from. Determining the best weapon to use becomes a puzzle of juggling the available equipment with the danger you’re up against.

Building a system where enemies provide **Dropped Weapons** gives the player even more choices to consider. This also becomes a great reward for beating a larger enemy: you get to wield that amazing weapon that you were just dodging.

- **Breakable Weapons**

- A hotly contested concept in modern combat systems is the idea of **Breakable Weapons** – equipment with a set lifespan, where using it too much will cause them to shatter. There are pros and cons to this approach, but much like an Ammo system, Breakable Weapons force players to strategize about which weapons to use, and which ones to save for a later battle.

- **Weapon Crafting**

- Gathering resources and letting the player **Craft** their own weapons can result in some enjoyable adventuring with player-driven objectives (“I need to find 2 more Iron so I can forge that sword!”). This system pairs well with Breakable Weapons and Weapon Loadout.

- **Aim Assist**

- To make battles a bit more fun, you may want to consider an **Aim Assist** system to help players hit targets with a higher rate of success. This will increase fun but decrease skill/difficulty, but if you’re getting player feedback that combat is “too frustrating”, you may want to consider adding this feature.

- **Air Juggling**

- A fun addition to melee combat – **Air Juggling** is where you knock enemies into the air and continue to attack them while airborne. Makes you feel like a melee-master, so it’s a great skill to upgrade to.

- **Stats and Leveling Up**

- In many games, the strength of an attack is determined by a combination of the weapon strength + physical strength. In this way, the player can amass experience and level up their own

strength to boost the damage being dealt. Leveling systems also open the door for special abilities to be learned, creating new play styles and unlocking awesome attacks as the game unfolds.

A unique combat system with clever intricacies is a great way to hook users and keep them playing. These are only a handful of combat mechanics that you can use in your game, but always be thinking about ideas that haven't been done before.

A game design with a unique combination of combat ideas is a powerful way to set your game apart from the rest!

Conclusion

Combat, and the weapons associated with them, give the designer lots of options when creating their game. Is this a game where the player needs to stay back and attack from afar, managing their guns and ammo? Is combat an 'up close and personal' experience, with a large variety of cool Melee weapons to use in hand-to-hand battle? Is it somewhere in between, where the player needs to manage the pros and cons of their inventory against a given situation?

No matter the game you're designing, weapons are a great way to add excitement to the player's experience!

Another important design tool for adding excitement is through the often-overlooked areas of Music and Audio. Let's dive into those subjects now and give them the attention they deserve!

Questions

1. Verbs give the designer a way to simplify gameplay down to a set of high-level actions. Think back to your favorite games - what are some of the verbs (actions) that hooked you in those titles?
2. What are the pros and cons of a player using a Melee weapon? What about a Projectile weapon?
3. The Weapon component is set up in a way where even melee weapons can have 'bullets' that they launch. Can you think of any games that have melee weapons that can also fire a projectile? What are the

gameplay prerequisites tied to these special attacks (full health, charge-up time)?

4. What would we need to do to get Enemies equipping weapons? What AIBrain functions would we need to add to make enemies attack with them?
5. Our hero and enemies are roughly the same height, ensuring the Collision Bounds of the sword is always within striking distance. Where would we encounter problems if the hero was taller than the enemy?
6. Limited ammo is a common gameplay feature in games with projectile weapons. Can you think of a way to expand our Weapon component to allow for limited ammunition before needing to reload?
7. Besides Ammo Systems, we listed some advanced combat mechanics toward the end of this chapter. Which of those sounded like something you'd want in your game? Using our current system, how would *you* implement this feature?

Key Terms

- **Verbs:** Game design adjectives to describe what a players *does* – what actions they perform while playing your game. Jumping, Talking, Hiding, Collecting, Building, and Fighting are common verbs that you'll experience.
- **Melee:** Combat where the player has to get close to enemies to attack. Typically requires a weapon, but could be hand-to-hand.
- **Projectile Weapon:** A weapon that fires ‘bullets’ from afar. Allows the player to feel safe from a distance, but is often balanced against a limited supply of ammunition.
- **Pivot Point:** This is the location around which an object moves and rotates. When designing weapons, it’s important that this location is wherever the weapon will be held.
- **Locators:** Empty Game Objects that can be used to determine the position and rotation of equipped items.
- **Knockback:** A physical pushing of objects that take damage. The more an enemy is knocked back, the stronger a weapon impact *feels*.

- **Attack Animations:** Animation Clips that are specifically made for different types of weapons. These traditionally don't need to loop.
- **Trail Renderer:** A component that will leave a line of a given width behind a moving object. Very useful for showing the arc of a weapons attack.

CHAPTER 10

All About Audio

When it comes to making a great video game, one of the most underrated aspects is **Audio Design**. While the graphics may create exciting screenshots, and gameplay may raise our adrenalin, it's the Music and Sound Effects that drive the *emotional* impact of a game.

The power of Video Game Music spans from AAA titles to indie hits. Think about the award-winning indie game *Undertale*. Simple graphics, clever morality-based gameplay, but it's tracks like *Heartache*, *Hopes and Dreams*, and *Megalovania* that resonate so deeply with players. The song *Theme of Love* from Squaresoft's *Final Fantasy 4* is taught in the Japanese 6th Grade Curriculum. And orchestral versions of video game music fill halls with gamers, excited to sit and listen to the music that filled their childhood memories.

Let's learn how Unity's Audio System gives you the tools you need to make a memorable soundscape for your players.

Structure

In this chapter we will discuss the following topics:

- The Audio Components
- Importing Music and Sound FX
- Creating our Music System
- Basics of 3D Sounds
- 3D Sounds for Items
- 3D Sounds for Weapons
- Mastering Audio
- Platform Considerations

Objectives

In this chapter, we'll be performing a deep dive into Unity's Audio components. We'll implement a system for playing music and 2D sound effects. We'll also tackle 3D sounds for creating the proper atmosphere for your game.

The Audio Components

The Unity Audio System is very straightforward. There are Listener objects that will hear sounds, and Source objects that will emit sounds.

Audio Listener

The most basic sound-related component is the **Audio Listener**. It lacks any sort of parameters - simply placing it on a Game Object will designate that object as a 'Listener', as shown in [figure 10.1](#). When that object gets close to a source, the audio will grow louder.

Figure 10.1: The Main Camera object has an Audio Listener by default.

Unity also uses the relative location of the listener and source to balance channel output. For example, if a monster is growling to the right of the listener, the sound will primarily come through the right speakers.

Audio Source

While the listener object is a key part of the audio system, it's the **Audio Source** component that will be playing the sound and music files for your game. Any GameObject that has an audio source component added, as shown in [figure 10.2](#), will emit sound from that location.

Figure 10.2: The Audio Source component.

Whereas the Listener script had no parameters, you'll notice that the Source component has many, as listed in [table 10.1](#). These are important to master, since they will control the music and sound being played in your game.

AudioClip	The name of the sound file being played.
Output	The Mixer Group through which this clip is played. Mixer Groups can alter and correct audio with various dials and effects.
Mute	A toggle used to Mute the output of this Audio Source. Note that Muting does not affect playback speed (ie. audio being played, then Muted, will continue being played in the background).
Bypass Effects	Use to enable / disable all Mixer effects applied to this source.
Bypass Listener Effects	Use to enable / disable all Listener effects.
Bypass Reverb Effects	Use to enable / disable all Reverb effects.
Play on Awake	Have this enabled for any sounds that should play the moment an object awakes. If this is disabled, you'll need to hook up a script to start playing the audio.
Loop	Should this file start playing again once it reaches the end?
Priority	Importance of this sound, from 0 (Most Important) to 256 (Least Important). If there are too many sounds being played, Unity will start pruning them out, starting with the least important ones.
Volume	The loudness of this audio. For 3D sounds and music, this is the volume when the listener object is 1 Unit (1 Meter) away from the source.
Pitch	You can modulate the pitch of an Audio Clip with this parameter, making it either higher or lower with a simple slider.
Stereo Pan	Which speaker this audio is emitted from (2D only).
Spatial Blend	Whether this audio should be played in 2D space, 3D space, or somewhere in between. In general, UI audio and music will be 2D and all other audio will be 3D.
Reverb Zone Mix	How much this audio source is affected by a Reverb Zone.

Table 10.1: The Audio Source component has many parameters to play with when perfecting the sounds and music in your game.

Importing Music and SoundFX

There are several ways to get music and sounds for your game. They are as follows:

Outsourcing

If you want great, unique music made for your game, there's no better option than outsourcing. By hiring a musician to collaborate with, you may spend more money on the process, but you can be confident that the music in your game matches your vision and, most importantly, won't be heard elsewhere.

Royalty Free Libraries

There are many libraries with pre-made, looping tracks and sound effects available for licensing. Making use of these tracks will save you some money while keeping a high level of musical quality.

Creative Commons

If you're pinching pennies, you can find a large assortment of free-to-use music and sound effects on various game development websites. While the price is right, the downside is that any free high-quality music is probably being used in countless other projects.

The Unity Store

A great starting point for your game development needs is the Unity Asset store. Besides Music and SoundFX, this storefront allows creators to sell a wide variety of assets that you can use in your games - and even entire game templates! For our purposes, we're going to start here in our quest to find some music to hook-up.

Figure 10.3: The Unity Asset Store (assetstore.unity.com) is a place for creators to sell music, models, and gameplay systems to other creators.

Using the Unity Asset Store is a browser based tool that you can access from anywhere. The steps to find some music for your game are simple and straightforward...

1. Open a web browser and go to **assetstore.unity.com**. Keep Unity running in the background, so when we choose our assets, they will automatically download into our project.
2. In the Search for Assets bar at the top, type in 'Music and Sounds'. You'll then want to navigate the right panel and select the **Free Assets**

checkbox (though supporting creators by purchasing music is also encouraged).

Figure 10.4: Our list of free Music and SoundFX samples in the Asset Store.

3. In the Search for Assets bar at the top, type in ‘Music and Sounds’. You’ll then want to navigate the Filters Panel, on the right side of the site, and select the **Free Assets** checkbox (though supporting creators by purchasing music is also encouraged).
4. You should now have several pages of choices. While variety can be nice, it can also be overwhelming. Use the Ratings filter option to narrow possibilities to only the highest rated content as shown in [figure 10.5](#)

Figure 10.5: Trim your options by using the Ratings filter.

5. At this point, it’s up to you to pick the music that you would like to add to the project. Click through some assets, listen to some samples, and find something that you like.

Figure 10.6: Once you’ve found something you like, press the Add to My Assets button.

6. Pressing the **Add to My Assets** button ([figure 10.6](#)) will bring up a window asking you to open the package in Unity. Press **Open in Unity** to start the import process ([figure 10.7](#)).

Figure 10.7: Adding an asset to your project can be triggered directly from the browser.

7. Back in Unity, the Package Manager will open, showing you the asset package that you just selected. Press **Download** in the lower right corner to bring it into your project ([figure 10.8](#)).

Figure 10.8: The Package Manager

8. Once downloaded, select the Import option. This will bring up a popup that lets you specify the specific package assets you want to

bring into your project ([figure 10.9](#)). By default, all assets are selected.

Figure 10.9: By default, Unity will suggest downloading everything from a package. If you want, however, you can limit the import process to only the assets you want.

9. Press the Import button to bring all music tracks into your project.

Importing Audio Clips

While the Asset Store makes it easy to get music and sound effects into the game, we'll also want to cover the process of importing through other methods. The steps are as follows...

1. Our first step is to create a folder structure for all of our incoming audio files. Go to the **Projects** window and create a folder named "Audio". Within that Audio folder, create two sub-folders: Music and SoundFX ([figure 10.10](#)).

Figure 10.10: Organize your Music and SoundFX now to avoid headaches later.

2. To keep everything organized, let's move any music from the Asset Store into our new Audio > Music directory. Do this by selecting the files and dragging them into the new folder, as shown in [figure 10.11](#)

Figure 10.11: Make sure all your Audio files stay organized!

3. Let's say you have external files you want to import. That process is just as easy! Simply open your Assets > Music folder, right click, and select the **Import New Asset** option. This will bring up an explorer popup that lets you import an external music file into your project.
4. Once your music folder contains some Audio Clips, select one to view its **Import Settings** in the Inspector Window ([figure 10.12](#)).

Figure 10.12: The Import Settings for Audio Clips. Since Audio Files can be fairly large, it's important to master the art of compression.

It's easy to accidentally make your final build too large by ignoring these important **Import** parameters. Sound and music files are easy to compress, so don't forget to use the parameters listed in [table 10.2](#)

Force to Mono	If an audio file has multiple channels, this option will condense them into a single channel. This is an easy way to cut file size in half (if not more).
Normalize	Should this audio be normalized (have its volume adjusted to reasonable levels) when Forced to Mono.
Load in Background	Should this clip be loaded in a separate thread (keeps the main thread from stalling/pausing for large sound files).
Ambisonic	Is this a sound meant for 360 video or an XR application?
Load Type	How should Unity process this audio at runtime?
Preload Audio Data	Should this clip be loaded when the scene is loaded?
Compression Format	The format used when compressing this audio (PCM, MP3, etc).
Quality	How compressed should this audio be? The lower the quality, the smaller the file size.
Sample Rate Setting	The sample rate used in the conversion process.
Size Data	This panel will show you the effect these Import settings will have on this clips filesize.

Table 10.2: The import and compression options you have when managing your games music and sounds effects.

We'll be keeping all of these as defaults, but later in the chapter we'll be discussing considerations to make when importing and compressing files for various platforms.

[Creating our Sound Manager](#)

With our music downloaded, organized, and given the proper import settings, it's time to hook up a Manager to handle the playing of music.

1. Open **Sample Scene** - the scene with your test level - so we can see how music and sound effects change the feel of our demo game.
2. Once that scene is open, add an **Empty GameObject** and rename it **SoundManager**. Use the Inspector Window to add an **component to it.**

Figure 10.13: An empty game object with an AudioSource script: the humble beginnings of our Sound Manager.

3. The AudioSource component is where we'll be setting and playing the game music from. In the **AudioClip** parameter, press the round circle button on the right and select a track to play for this level ([figure 10.14](#)).

Figure 10.14: The AudioClip parameter dictates the sound, or music track, played by this source object.

The two additional parameters you will want to validate are **Play On Awake**, which should be **On** by default, and **Spatial Blend**, a slider that should be completely set to the **2D** option.

With these settings verified, it's time to press the **Play** button! As you move through your level, you should hear the track that you've selected, injecting the game demo with some thematic excitement.

While we have some music playing in the scene, let's use this opportunity to learn more about the other type of spatial audio: **3D Sound**.

The Basics of 3D Sound

One of the most important AudioSource parameters is the **Spatial Blend** slider. This value will determine if the given AudioClip will be heard in **2D** space (unaffected by player position) or in **3D** space (where the volume and position change in relation to the player).

The music is currently being played in 2D space. Let's bring up the AudioSource component and change that.

Figure 10.15: Use the Spatial Blend slider to adjust a sound to play in 2D space, 3D space, or somewhere in between.

The testing of music and sound is best done in Play mode, where you can move around and hear the affect of your changes. Follow these steps as we walk through the testing of these 3D sounds...

- With the slider fully set in the **3D Space** position, let's press **Play** and walk through our level. Notice that, at first, as you walk around, the volume of the music will change. The closer you are to the SoundManager object, the louder it will be. The further you are, the quieter it will become.
- You may also notice a distortion effect as you move around. This is due to the **Doppler Level** - one of the *many 3D Sound Settings* you can adjust in your Audio Source component parameters ([figure 10.16](#)).

Figure 10.16: The 3D Sound Settings for this Audio Source.

You need to tweak any aspect of your 3D sounds directly in the Audio Source parameters (under the **3D Sound Settings** header). Full descriptions of these options are listed in [table 10.3](#)

Doppler Level	The intensity of the ‘Doppler Effect’ (change in pitch based on speed) between this Audio Source and the Listener.
Spread	The spread angle of this audio when played in 3D space.
Volume Rolloff	The mathematical curve of how the volume changes as the listener gets closer, or further, from the audio source.
Min Distance	The minimum distance for this sound. If set to anything greater than 0, the audio will always be heard, no matter how far the source is from the listener.
Max Distance	The maximum distance the sound can be heard from.
Falloff Curve	A visual representation of the change in volume over distance.

Table 10.3: The 3D sound settings for an Audio Source.

- Let's change the **Doppler Level** to **0**. On cars and objects that move quickly, it's a very convincing effect. Hearing the Doppler Effect on our level music, however, it sounds a bit *odd*.
- Next, let's tweak the Max Distance: by lowering that parameter to 30, the **Audio Rolloff** (change in volume based on distance) will be more noticeable.
- And speaking of rolloff, let's change the **Volume Rolloff** curve from Logarithmic Rolloff to **Custom Rolloff**. By default, the Custom Rolloff will keep the audio louder for longer, then have a nice smooth falloff as you get further away.

Figure 10.17: A nice, smooth rolloff curve for our music. Also, note the Listener text at the top. When the game is being played, this will move to show you the distance between this Audio Source and the scene's Listener Object.

3D Sound on Items

One of the most iconic sounds in the video game world is the ‘Pickup Coin’ sound from *Super Mario Brothers*. It’s simple, clean, and instantly triggers the satisfaction of gathering that spinning, digital gold.

Let’s make sure our coins give that same level of satisfaction with some sounds of our own!

First step is to find some ‘coin collection’ SoundFX that you like. They can be found on the Unity Asset Store, but there are also useful tools online for creating retro sounds for your games, as shown in [figure 10.18](#)

Figure 10.18: Sites such as www.sfxr.me let you generate retro sound effects and save them as wav files. While these may not be the SFX you’ll ship with, they’re perfect for white-boxing.

Once you have a ‘pickup coin’ sound exported, go to your Downloads folder and drag the sound file into the project’s **Assets > Audio > SoundFX** folder ([figure 10.19](#)).

Figure 10.19: The Coin Sound file, after being placed in our projects ‘SoundFX’ folder.

With a sound ready to hook up, it’s time to think through the implementation steps. Since Unity is a component based engine, and we now have a firm grasp on the Audio Source component, It would only make sense to add an Audio Source to a coin, then play a coin sound when that object is picked up.

Unfortunately, that option would actually result in NO sound being played. Why? It’s because the Pickup Item is *destroyed the moment it’s grabbed*. And if we destroy the game object tied to the Audio Source, then the Audio Source would also be destroyed. Meaning no soundFX.

The solution we’ll be using is to implement a new **SpawnedSoundFX Prefab** that can be created as needed. This object will be spawned, play its

sound, then self-destruct.

Create the SpawnedSoundFX object by following these steps...

1. Create an empty Game Object in the Hierarchy Window.
2. Rename this object **SpawnedSoundFX**.
3. Add an **Audio Source** component. Set the **AudioClip** to ‘PickupCoin’.
4. Add a **Self Destruct Timer** script (similar to what we did with the Bullet prefabs).
5. Add a new script called **SpawnedSoundFX**.

Once all these components have been made, open the SpawnedSoundFX script and fill it out with this block of code...

```
public class SpawnedSoundFX : MonoBehaviour
{
    public AudioSource _audioSource;
    static string _prefabPath = "Prefabs/SpawnedSoundFX";
    public static void Spawn( Vector3 pos, AudioClip clip = null
    )
    {
        // spawn soundFX object
        GameObject prefab = Resources.Load<GameObject>(_prefabPath);
        GameObject newObj = Instantiate( prefab, pos,
        Quaternion.identity);
        // todo: add randomness
        // todo: swap audio clip
    }
}
```

Right now, the only thing this class contains is a static **Spawn()** function that can be easily called from anywhere in the code. Let’s make use of that function in the **Pickup Item** script. Find the function **void onPickedUp()** and add these two lines of code to the top of the function...

```
// play soundFX
SpawnedSoundFX.Spawn( transform.position );
```

Back in the Unity editor, select the SpawnedSoundFX object. In the Inspector window, drag the AudioSource component into the corresponding

Audio Source parameter of our Spawned Sound FX Script ([figure 10.20](#)).

The last step is to drag the new object from the Hierarchy and into the **Assets > Resources > Prefabs** directory.

Figure 10.20: The components of the new SpawnedSoundFX Prefab.

Delete the original SpawnedSoundFX Prefab and press the **Play** button. You should now hear a ‘Coin Pickup’ sound every time you grab a spinning gold coin.

Feels great, doesn’t it?

The only complaint is that, like most repetitive sounds, it could use some modulation to break up the repetition.

Back in the SpawnedSoundFX script, go down to the “todo: add randomness” comment and replace it with the following code...

```
// add randomness to pitch
float rand = Random.Range(0.95f, 1.05f);
SpawnedSoundFX soundScript =
newObj.GetComponent<SpawnedSoundFX>();
soundScript._audioSource.pitch = rand;
```

By adding just a small bit of randomization to the pitch of the Audio Source, we can ensure that our coin collection sound never gets too repetitive.

3D Sounds for Weapons

Another part of our game in desperate need of audio attention is in our Weapon system. Sword swings and blaster fire are perfect spots to add some additional soundFX.

Figure 10.21: Find or create some sound effects for our two weapons attacks. Name them FireLaser and SwingSword.

Open the Weapon script and add this member to the top of the class...

```
[SerializeField, Tooltip("Audio to play when weapon is
used.")]  
public AudioClip _attackSoundFX = null;
```

At the top of the **onAttack()** function add..

```
// play 'attack' soundFX  
SpawnedSoundFX.Spawn(transform.position, _attackSoundFX);
```

Add this block of code to the **Spawn()** function in SpawnerSoundFX..

```
// swap audio clip  
if (clip)  
{  
    soundScript._audioSource.clip = clip;  
    soundScript._audioSource.Play();  
}
```

Now, back in Unity, bring up the weapon prefabs and assign the new attack sounds to the Weapon scripts ([figure 10.22](#))

Figure 10.22: Assign an Audio Clip to our Weapon script.

With the sword and blaster Attack Sound FX assigned, it's time to test. Press **Play** in your Weapon Sandbox scene, and every time you attack, you will hear the proper sound played in 3D space.

Mastering with Audio Mixers

In the field of sound design, there's a concept called **Audio Mastering**. This is the process of taking recorded audio and tweaking the output levels until it sounds *perfect*. Many tracks will come pre-mastered, but Unity gives you the ability to master audio directly in the tools with **Audio Mixers**.

Figure 10.23: Audio Mixers let you add effects and tweak output directly in Unity.

Let's make use of Unity's Audio Mixers by following these steps...

1. Add a new mixer by opening your Project window, and in the Assets > Audio folder, add a new Audio Mixer. Right click in the folder, and select **Create > Audio Mixer**. Rename the new mixer to **WeaponMixer** and double-click it to start editing.
2. While the Audio Mixer window is fairly extensible, there's a specific button we want to make use of here. Press **Add...** at the bottom of the

mixer to bring up a list of Effects you can assign. Select **Echo**, and use the Inspector Window to set the parameters shown in [table 10.4](#)

Delay	150ms
Decay	70%
Max Channels	1 Channel

Table 10.4: Tweaks to the effect parameters. Weapons and items sound will now have a nice Echo when they're played.

3. Mastery of Audio Mastering: Every effect listed comes with several parameters that can be used to tweak the output of an Audio Source. Between adding effects, routing output, and grouping mixers, Unity gives you a deep suite of tools to master the art of Audio Mastering. And while it's a deep topic that we can only briefly touch on, it's certainly a skill worth adding to your game design toolbox.
4. Before we'll actually hear this change, we need to assign this mixer to the Audio Source component. Open the **SpawnedSoundFX** prefab, bring up the Audio Source component, and find the **Output** parameter. Click the small circle button on the right ([figure 10.24](#)) and select the **WeaponMixer** option.

Figure 10.24: Assign your Mixer to an Audio Source using the Output parameter.

5. Now play the game. By using the **Audio Mixers** and effects, the weapon sounds should have a nice, subtle Echo - exactly as we intended.

Platform Considerations

One of Unity's 'super powers' is the ease of which you can take one project and use it to build for multiple platforms. Now only does Unity handle the tricky, under-the-covers aspects of porting, the editor gives you the option to tweak how assets are handled on a per-platform basis.

This is especially useful for audio, where a music track can get surprisingly large without proper compression. You don't want mobile gamers downloading a gig of data, where 70% of that is poorly compressed audio.

Figure 10.25: When importing Audio, use the tabs in Import Settings to override compression settings for the various platforms you're targeting.

Conclusion

Music and sound effects have the subtle power to interject personality into any game. Unity's Audio systems provide all the tools and components you need to make this happen. Using Audio Listeners and Audio Sources to create a world of 3D sound, to using Mixers and effects to tweak audio on the fly - Unity gives you everything needed to create an incredible soundscape for players to enjoy.

With a better understanding of the tools you have to great create audio, it's time to turn our attention to a visuals of a game. Let's learn the ways Unity empowers us to make some amazing looking *graphics*!

Questions

1. What are the different ways to get audio and sound effects for your game? What are the benefits of outsourcing music vs. pre-made libraries?
2. What is the difference between 3D and 2D sounds? When would you use one over the other?
3. How many audio listeners can be active at a time in a scene?
4. Why do sounds related to the destruction of an object need to be spawned using a separate prefab?
5. Knowing what you know about the animation and audio systems, how would you go about implementing脚步声 (footstep sfx)?

Key Terms

- **Sound Design:** An area of game development that combines the skills of a musician with the technical expertise of a developer. Sound Designers make sure the music, sounds effects, and audible part of a game experience is perfect.
- **Audio Listener:** A component that determines the listener in a scene. This location is used to adjust the volume and channel breakdown of

audio as the player would hear it.

- **Audio Source:** A component that plays audio for the listener object to hear.
- **2D Sounds:** Sounds that play in 2D space are not affected by the position of the listener. They will play at a given volume at all times. Useful for UI sounds and musical themes.
- **3D Sounds:** Sounds that exist within the 3D scene. As the player moves around, they'll hear sounds that are closer, while far-away sounds will be quiet.
- **Spatial Blend:** The balance between 2D and 3D for a given Audio Source.
- **Rolloff Curve:** How the volume changes as a sound gets further from the listener.
- **Doppler Effect:** A change in pitch heard as a fast object moves past a listener (example: when a cop car speeds past you with their siren on).
- **Mastering Audio:** The process of tweaking the output of an audio file (volume, channel balance, etc) until it sounds perfect.

CHAPTER 11

A Graphical Upgrade

Starting the game development process with simple, White-Boxed assets is a great first step. At some point, however, you'll need to think about your art style.

“A picture is worth a thousand words.”

Video games are a visual medium. Sure, we needed to learn about Input and Gameplay and Audio Design, but Screenshots, Trailers, and Dev Streams are just as important. If your game doesn't *look* compelling, players are going to pass on it for something that *does*.

It's time to leave our basic visuals behind and start talking about all the ways Unity lets you create something that *looks amazing*.

Structure

The concepts we'll cover in [Chapter 11](#) are...

- Unity's Art Tools
- ProBuilder
- Modeling with ProBuilder
- Placing Assets on a Grid
- Adjusting Level Assets
- Kitbashing
- Skyboxes
- Distance Fog
- Post-Process Effects

Objectives

Unity has several tools and systems designed to make it easy for you to create an impressive looking game. By the end of this chapter, you will have a solid understanding of creating assets within Unity using the powerful ProBuilder toolset.

You'll also learn about several ways you can polish the look of your game, by either by mastering the included Post-Processing systems, or by utilizing amazing assets directly from the Asset Store.

Unity's Art Tools

Traditionally, an asset pipeline would have artists creating textures, models, and other game assets in a tool separate from the game engine. They would then import the asset, then a designer would hook them up in-game. This multi-step process was prone to error and often had assets going back to the artists multiple times for cleanup and revisions.

In their push to make the editor a one-stop solution for game creation, Unity has included several tools and systems that allow you to create and hook up assets directly within the editor. No more juggling of multiple software packages. You can make all your graphical upgrades right in Unity.

You can set up the tools by following the easy steps listed below...

1. Let's start by opening the Unity Editor, and from the top menu bar, select **Windows > Package Manager**. The Package Manager is a central access popup from which you can enable and disable various systems that Unity provides.
2. Within the packages in the Unity Registry, you will find a feature set called **3D World Building**. Select that package, and press the **Install** button in the lower right corner ([figure 11.1](#)).

Figure 11.1: Install the 3D World Building feature set, which includes several powerful tools for creating assets directly within Unity.

3. Once these packages are downloaded and installed into your project, you'll want to also make a clean **ArtSandbox** scene to use when learning these new tools. Open the Scenes folder in your project window, add this new scene ([figure 11.2](#)), then open it.

Figure 11.2: An empty Art Sandbox scene will help us stay organized while learning Unity's art tools.

Grabbing some Materials

When talking about the visuals in any modern game engine, there are two key inputs that define what the visual output -

- **Meshes**
- **Materials.**

The mesh is a 3D shape, created from triangles, that gets calculated, rendered in 3D space, and displayed to the player. For example, all the Cubes, Spheres, and Cylinders we've been using have been **Meshes**.

The **Material** defines what colors, images, and various reflective effects get displayed on a given mesh. So far, all our materials are just color variations, but they could also be assigned a more interesting Textures for various purposes (normal, occlusion, roughness, etc).

We'll get into the specifics of Materials later, but for now there's one fact we need to acknowledge: without great looking materials, it's difficult to make a great looking game.

Let's fix this by heading into the Unity Asset Store to find some materials to make use of. Much like the music samples we found in [Chapter 10](#) "All About Audio", materials from the Asset Store will help us learn Unity's art creation pipeline without needing to be a master Artist. Search for "Free Materials", and once you've added the assets to your account, return to Unity and install them with the Package Manager, as shown in [figure 11.3](#)

Figure 11.3: Go to assetstore.unity.com and search for 'free materials'. I suggest grabbing a package with several ground materials to use in our demo.

After importing the assets, you should now have a new folder with several pre-made materials listed, as shown in [figure 11.4](#)

Figure 11.4: The folder with the Asset Store materials. These will be very useful as we create new assets for our demo game

ProBuilder

When creating assets for your game, the most versatile tool you have is **ProBuilder**. Letting you build meshes, apply materials, adjust UV maps, and set smoothing groups, ProBuilder puts the power of asset creation directly in the editor.

Open the ProBuilder tools by following these steps...

1. Go to the menu bar at the top of the Unity editor and select **Tools > ProBuilder > ProBuilder Window**.

Figure 11.5: Unhide the ProBuilder tools from the Unity Menu Bar.

2. You'll be using this popup window to access the available ProBuilder actions. Advanced users may want to make use of the 'Icon' mode, but users new to the toolset will feel more comfortable in 'Text' mode.

Figure 11.6: Use the 'three dots' on the top to toggle between Text and Icon mode.

The Probuilder Window gives you access to countless asset creation options. Let's cover the most important options now.

New Shape

There are 12 shape presets included with the ProBuilder tool. Select **New Shape** to bring up a submenu (figure11.7) that lets you quickly place an **Arch, Cone, Cube, Cylinder, Door, Pipe, Plane, Prism, Sphere, Sprite, Stairs, and Torus**.

Figure 11.7: Create shapes using the ProBuilder presets.

Creating a new shape simply requires you to select the shape present, then drag the mouse in the scene to create the shape. You can then use the toolbar at the top to edit **Vertices, Edges, Faces**, or the entire **Object**.

Figure 11.8: Use this toolbar to change what part of the object you want to select and edit: Object, Vertex, Edge, or Face.

New Poly Shape

The **Poly Shape** tool is similar to the New Shape tool, only here you get to define the shape being created. Click to add vertices, and once you close the shape, you can extrude it to the desired height ([figure 11.9](#))

Figure 11.9: New Poly Shape lets you define a uniquely shaped mesh. Here we've made a Poly Shape that looks like the Unity logo.

Smoothing

When defining a rounded mesh, it's important to understand how **Smoothing Groups** work. Smoothing Groups allow you to assign several faces to a 'group' so the edges between them get smoothed out ([figure 11.10](#)). This is ideal for spheres, cylinders, and anything where have many faces and polygons that need to look like a single, unified element.

Figure 11.10: The cylinder on the top-left has the faces on the side assigned to the same smoothing group. Remove the smoothing groups, and you see the lines between faces, as shown by the cylinder in the top right.

Material Editor

ProBuilder has a Material Editor tool that allows the quick assignment of different materials on the different faces of a model ([figure 11.11](#)).

Figure 11.11: Use the Material Editor to assign different materials to assignment hot-keys.

UV Editor

When modeling a 3D object, there is a set of data called the UV coordinates that tell the rendering system how to display a material over an object. Think of it like wrapping a gift - you have a flat set of material textures that get wrapped around the faces of an object.

The ProBuilder UV Editor ([figure 11.12](#)) allows you to manipulate this texture mapping data to ensure the object you're creating looks exactly how you envisioned it.

Figure 11.12: Use the UV Editor to make adjustments to how a material is wrapped over the model you're creating

Center Pivot

When shapes are initially created by Pro Builder, their default **Pivot Point** - the point at which the object is moved, rotated, and scaled - will be placed in the lower corner of the object. In most cases, however, it's more useful to have a pivot point directly in the center of your mesh ([figure 11.13](#)). To do this, simply select a placed object and click the **Center Pivot** option. You can also change the pivot of new objects by changing the Pivot option in the Create Shape panel.

Figure 11.13: The pivot point, before being centered (left) and after being centered (right).

Extrusion Hotkey: Shift

One last ProBuilder feature that you'll want to master is mesh **Extrusion**. When you select a mesh, you can then hold **Shift** and drag the selected faces outwards. This will **Extrude** that face - creating a new set of mesh data in the direction you're pulling the selection.

Like many art-related concepts, extrusion is better *shown* than *explained*, so let's build an **L** by extruding our way through these steps...

1. First, start by placing a ProBuilder Cube in the scene.
2. Second, make sure your selection mode is set to Faces. Select the top face, then extrude by dragging upwards while holding shift.
3. Lastly, select a side face and extrude outwards while holding shift.
4. You should now have an L mesh, as shown in [figure 11.14](#)

Figure 11.14: Using Extrusion to build the 'Letter L'.

The end result is a mesh that has new faces, edges and vertices to sculpt with. Extrusion is an invaluable skill to master when modeling 3D assets.

Building Blocks

With the ProBuilder basics covered, it's time to get started creating our first new asset, a staple of games for the past 30 years - the breakable *Brick Block*.

Let's go through the process, step-by-step, to cover how these creation tools can be used...

- a. Start by placing an empty object called **BreakableBrickBlock**. This will be our parent object for all the mesh objects we'll be creating.
- b. As a child of the BreakableBrickBlock, place a new **ProBuilder Cube** object. Use the **New Shape** tool, place a Cube, then use the Create Shape panel to set the size to 1, 1, 1.

*ProBuilder Note: You can also add shapes using the right-click menu in the Hierarchy window with the **Probuilder** > **Cube** option. A word of warning: this cube will not have the “ProBuilder Shape” component, so some editing features we use below will not be available.*

- c. Once this cube has been placed, name it **MainBlock**, and use the Hierarchy window to make sure it's set as the child of our BreakableBrick parent. Double click on its Hierarchy entry to center the camera on it.
- d. Next, using the **New Shape** tool, add another ProBuilder cube. Name it **Brick**, and set it as a child of the MainBlock object. In the Inspector window, find the **ProBuilder Shape** component and set the brick's size to **0.5, 0.3, 0.3**, as shown in [figure 11.15](#)

Figure 11.15: Using the ProBuilder Shape component, you can directly tweak the size and shape of a placed ProBuilder object.

- e. You'll also want to take this moment to remove the Mesh Collider component from the brick object. We'll want this block to be comprised of only one collider. One collider per brick would certainly be *overkill*.
- f. Now we'll do some extrusion to give the brick some more definition. With the ‘Select Faces’ mode set, click on the long side of the brick,

hold **Shift**, and pull outwards slightly. Then switch to the Scale tool, and scale the face to be a bit smaller. The end result should look like [Figure 11.16](#).

***Figure 11.16:** Extrude then resize to give the brick a edge. This will catch the light and give the model more definition.*

Since we'll be duplicating this brick several times, let's take a moment to re-color our objects for quick identification.

- g. Find **matBlack** and drag it onto the MainBlock object. This will make our large block nice and dark, allowing the brick objects to be easily visible against it.
- h. As for the bricks, create a new material called **matRedBrick**. Set **Albedo** to an reddish-orange color, and for the Normal Map, select a texture with a rocky feel to it. We'll use this to give our brick some subtle noise. With the texture set, you should also change the Texture Map value from 1 to 0.1. This will ensure the effect of the texture isn't overpowering.
- i. Assign our new material to the brick object, then use **Ctrl+D** to duplicate several copies. Place the bricks so they cover one side of the block ([Figure 11.17](#)).

***Figure 11.17:** Use the bricks to make one side of our object. We can duplicate this a few times to speed up the modeling process.*

- j. Select all the bricks we've placed, press **Ctrl+D** to duplicate the entire side, and rotate them to the backside of the MainBlock. Keep duplicating bricks until the MainBlock is entirely covered ([figure 11.18](#))

Concerning Precision: You'll notice that we're no longer laying out the specific location for these Brick objects. Since we're over halfway through the book, I feel confident in your ability to line up and place objects without too many specifics. Another thing to note about precision in 3D art is that, in general, modeling imperfections a necessary step in giving objects personality. The nature of computer graphics means it's easy to create a model

that's too perfect, resulting in assets that may feel mechanical or lifeless. It's unique imperfections that will make your 3D assets memorable!!

Figure 11.18: Our Breakable Brick!

- k. Our last step is to add a BoxCollider component to the base object. Select the **BreakableBrickBlock** parent and add a **BoxCollider** component. It should come in at a size of 1 meter cubed, so now it's just a matter of tweaking the child object to fit nicely within the bounds of the new collider ([figure 11.19](#))

Figure 11.19: Use the Scale tool to get the mesh fitting nicely within the bounds of its BoxCollider.

Grid Snapping and Asset Placement

With our Breakable Brick ready, it's a good time to learn about **Grid Snapping** in Unity. It's a great tool when placing objects that need to be aligned perfectly.

Use the following steps to set up your scene so incoming objects snap to a grid...

1. Start by toggling the **Grid Visibility** option in the Scene toolbar. If the icon is Blue, then you know the grid is Enabled. Press the arrow button to show the available options, which are listed in [table 11.1](#)

Grid Plane	Which axis is the grid going to be displayed on?
Opacity	Change the transparency of the grid based on how subtle you want it to be.
Move To	On the selected axis, should the grid be placed at the Origin (position zero) or at the Handle position of the selected object.

Table 11.1: The display and movement options for grid snapping.

Figure 11.20: Enable the Grid Visual options to show a grid on the axis of your choice. We'll be using the Y-Axis, which will show a grid on the floor plane.

- Once the grid is visible, we'll next tap the 'Magnet' icon to the right. Select the arrow to bring up our snapping options.

Grid Size	Our 'snapping grid' size. Note that it doesn't have to be the same granularity of our visible grid.
Align Selected	If you have multiple objects selected, this will let you align them on a given axis.

Table 11.2: Grid snapping options.

Figure 11.21: Since most of our assets will be 1 meter in size, it's a good idea to set our snapping grid to be 0.5 or 0.25 units, which lets us snap to the half or quarter meter.

- The last set of options is for **Increment Snapping**. These are the changes in value when you drag an object with the CTRL/Command button held.

Move	Change in incremented value when using the Movement gizmo.
Rotate	Change in value when incrementing with the Rotation gizmo.
Scale	Change in value when using the Scaling gizmo.

Table 11.3: Incremental movement snapping options.

Figure 11.22: The change in value to apply when holding CTRL/Command while using the Movement, Rotation, or Scale gizmos.

- At this point, it's a good idea to make a small test scene with our bricks. The intention is that our small hero should easily fit through, and navigate around, brick-sized gaps in the scene. Create a layout that looks like [Figure 11.23](#), then press play to make sure movement around the bricks feels good.

Figure 11.23: This is a perfect test scene. With collisions properly set, our small hero should easily fit through a single-block sized openings.

Building Level Prefabs

Building Blocks are great for smaller, more intricate forms of level design, but when creating an expansive environment to explore, we're going to

need some larger level assets to make use of.

1. Once again, let's start with a ProBuilder Cube.
2. Place it in the middle of the scene and rename it **LevelObj_Ground01**. Let's set its size to 3 meters cubed and assign an appealing rocky material to it ([figure 11.24](#)).

Figure 11.24: The start of our new 3x3x3 ‘ground’ level object.

3. Change the selection mode to **Face**, and select the topmost face of the cube. While holding shift, Pull the face up to perform a thin extrusion, then drag a thick extrusion, then a final thin extrusion. Once all three extrusions have been performed, your cube should now resemble [Figure 11.25](#).

Figure 11.25: The extruded top of our cube. There should now be a thick extrusion in between two thin extrusions.

4. With the top three extrusions made, let's select the middle ‘thick’ extrusion. While holding CTRL / Command, select the thick faces that we extruded. To get all the sides, you'll need to keep rotating the camera around the object using ALT.

Camera Control: Moving the camera while selecting faces, edges, or vertices is something you'll be doing a lot of when modeling. Take the time to get comfortably performing a multi-select on all sides of an object. Hold ALT to rotate the camera. Double-click objects in the hierarchy to center them in the viewport. Use the Camera Gizmo to change the camera angle to get exactly the view you need. The faster you can navigate 3D space, the more comfortable you'll be creating assets for your game.

5. Once you have the 4 faces selected, switch to the **Scale** tool and increase the scale of these quad just a bit. The end result will be a cube with a slightly rounded top ([Figure 11.26](#))

Figure 11.26: The top of our cube should now have slightly rounded sides.

6. We're now going to change the top of this ground object to use a material other than the existing rocky texture. Continue to hold CTRL / Command, and select all the other faces that make up the top section (like in [Figure 11.27](#)). Once they're all selected, drag a new material onto them. The selected faces will change to make use of your preferred material, giving the ground object an identifiable 'top'.

Figure 11.27: Drag materials to assign them to selected faces on the top of this ground object.

Anytime you're making a level asset, it's important to have the player object available as a point of reference. Place the Sphere Hero, and perhaps a Breakable Block, on the top of the ground object to help judge scale ([figure 11.28](#)).

Figure 11.28: Placing the Hero next to new level assets helps us determine if our objects have the proper scale.

With the hero in place, you can see how the rock material - which once felt large and substantial - now feels more like a wall of pebbles. Our scale is off, but luckily there's a few ways to fix that!

Let's first resize the ground to give the player more space to run around.

7. Set your selection mode to 'Vertices' and drag a selection box to select all the vertices on one side of the ground object.
8. Once all the vertices you want to select are tinted Yellow, use the Move tool to drag the selection outwards, making the Ground Object wider ([figure 11.29](#))

Figure 11.29: Resize these level assets by selecting and moving Vertices, Edges, or Faces. If you use the Scale tool, you'll squish the material's UV map.

9. Keep selecting vertices this way until your small ground object becomes a large ground object

Unfortunately, the larger size has amplified some of our visual problems. The rock and ground materials are now feeling much too

small. We need to adjust the tiling values of these to increase their texture sizes and make our ground object more appealing.

10. Go to the project, find the two materials that have been applied to the object, and under Tiling, set X and Y to each be 0.2. This will decrease the number of times this material tiles per-unit, which has the effect of making the rocks and grass look larger ([figure 11.30](#))

Figure 11.30: If you want your materials to appear larger, you'll want to make their tiling values smaller. Smaller values = less tiling = textures appear larger.

You now have all the tools needed to create additional level prefabs. Drag this object into your prefabs folder, then use ProBuilder to create more shapes: different sized rectangles, ramps, cylinders. Create around 5 objects that you can build a level from, then use those to make a fun little test area for your hero, similar to the one shown in [figure 11.31](#)

Figure 11.31: Use this opportunity to make even more level prefabs! Fences and additional Ground Shapes will be very useful once it's time to officially make some levels.

With a handful of new level objects made and pieced together, we now have the start of an interesting looking environment.

The next step is to add more visual variety, but creating additional trees, bushes, visual effects, and background art would simply take too much time. How do we polish our scene without hand-crafting the additional assets ourselves? There simply must be a way to leverage your skills as a designer with the artistic skills of the community...

KitBashing

Unity has some amazing tools for creating assets directly in the editor, but these internal techniques will never match the power of industry-leading software. **3D Studio Max**, **Maya**, **Blender**, **Z-Brush**, **Mudbox**, and **Substance Painter** are tools designed with one purpose - the creation of amazing 3D assets. And in the hands of skilled artists, the models produced are *awe-inspiring*.

The drawback of these tools, however, is a steep learning curve. As a game developer, you already have a *mountain* of tools and systems to juggle in the game creation process. If only there was a way to make use of high-quality assets without needing to master these additional tools.

This is where the idea of **Kitbashing** comes into play. A term taken from the days of physical modeling (ie. miniature cities and model trains) Kitbashing has you taking various unrelated models (kits) and piecing them together (bashing) to make something new and unique.

The Unity Asset Store makes Kitbashing even easier. Skilled artists in the unity deaf community can create an upload asset packages that everyone can use. Download Forests, Deserts, Sci-Fi Hallways, Ancient Castles, or Futuristic Cityscapes ([figure 11.32](#))

Figure 11.32: The Unity Asset Store has everything you need to start the Kitbashing process.

By downloading and utilizing professional models from the asset store, you can focus on what you do best - making a great game - while the artists in the community can focus on what they do best - creating awesome art.

1. For our demo game, we are going to use the asset Store to download some trees and ground foliage to decorate our scene with ([figure 11.33](#))

Figure 11.33: A great asset for this task is called ‘The Illustrated Nature - Sample’, but use any package that catches your eye.

2. Once the package has been downloaded, and the assets imported to your project, it’s time to get *creative*. Locate the prefabs from your package and simply drag them into the scene where you would like them.

The example package “The Illustrated Nature” came with two trees, several ‘grassy tufts’, and a beautiful ‘God-Rays’ sun particle effect. In less than 2 minutes takes us from an empty, barren environment to an atmospheric mini-forest, ready for exploration ([Figure 11.34](#))!

Figure 11.34: Kitbashing - the process of bringing in external models and combining them into something new - lets you implement professional-grade visuals without needing to learn professional-grade tools.

Skybox

Unity's rendering system uses a **PBR** (Physically Based Rendering) graphics system, where materials in the scene are lit in such a way that its closely approximates the way you'd see something in the real world. When it comes to the overall look of your assets, it's important to understand the way objects interact with the scene-wide texture known as the **Skybox**.

The Skybox is an image that surrounds the entire scene that gets used for the ambient lighting of objects. Up until this point, we've been using Unity's default skybox - a generate 'horizon' image that uses the direction of the main light to place the sun and color the sky and ground accordingly ([figure 11.35](#))

Figure 11.35: Unity's Default Skybox changes as you rotate the main Directional Light object.

1. For our demo, we'll want a sky with a specific feel to it. A blue sky with large, fluffy clouds would be perfect. Let's go to the Unity Asset Store, find a package we like, and import it into our project ([figure 11.36](#))

Figure 11.36: For my skybox, I downloaded the 'Fantasy Skybox Free' asset package, but you're free to grab any asset that you'd like to use.

2. Now that we have some options, changing our current Skybox only take three clicks. First, select **Window > Rendering > Lighting** to bring up the **Lighting Window**.
3. Next, select the **Environment** Tab on the top of the popup ([figure 11.37](#))

Figure 11.37: The Environment Tab in the Lighting Popup has several universal lighting settings for this scene. For now, we'll be focusing on the Skybox Material parameter.

- Locate the **Skybox Material** parameter, click on the small circle button to the right, then select the skybox you'd like to use. You'll see the change take place instantly, both in the Scene and Game views ([figure 11.38](#))

Figure 11.38: Our scene with a new cloud-filled skybox.

You'll notice a few things instantly, with the biggest alteration being the color change across all game objects. This is the PBR system using the Skybox to determine the ambient light of the scene. A 'sunset' skybox would give everything a red/orange tint. A space background of purple nebula would have an effect of tinting everything purple.

While this may feel jarring at first, this is an accurate representation of how objects in an environment are lit, and will reflect the light of the surrounding environment.

If you want to tone down this intensity of the skybox, however, simply adjust the **Intensity Multiplier** value under Environment Lighting.

Distance Fog

Another useful Environmental parameter we can play with is the Fog setting. Scroll down in the Lighting window until you see **Other Settings**. Make sure that's expanded, then find the **Fog** options.

Fog	Toggle the Fog effect on / off.
Color	The color of the fog effect.
Mode	How quickly the fog transitions from <i>No Fog</i> to <i>Full Fog</i> .
Density	The distance from the camera that the fog starts to take effect. The larger the number, the fogger the scene will look.

Table 11.4: Distance fog options.

You can play with the fog settings and see the change immediately in the game window. Set the values to match [figure 11.39](#) to push some of the objects into background and help your foreground elements (hero, level objects, etc) to stand out.

Figure 11.39: Distance fog adds personality to an environment while keeping the players attention on the foreground elements.

Keeping your Workspace Clean

One side effect of adding all these visual improvements is that your Scene View - the area where you do your design work - may start to feel *cluttered*. Distance Fog, fancy Skyboxes, and Post Processing effects may create a better-looking game, but they also create a scene that's more difficult to edit.

To combat this, you can use the **Toggle Effects** dropdown menu at the top of the View window ([figure 11.40](#)). Use the arrow to bring up all the visual effects that can be toggled while editing.

Disabling these here will not disable them in the Game - it simply cleans up your workspace to make the design process easier.

Figure 11.40: Is your scene view looking cluttered? Disable some of the visual effects using the 'Toggle Effects' menu.

Post Processing

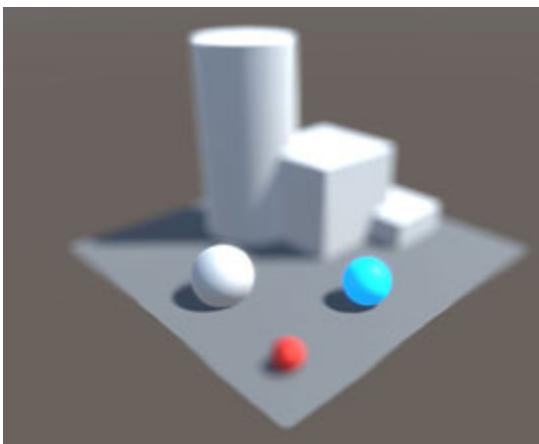
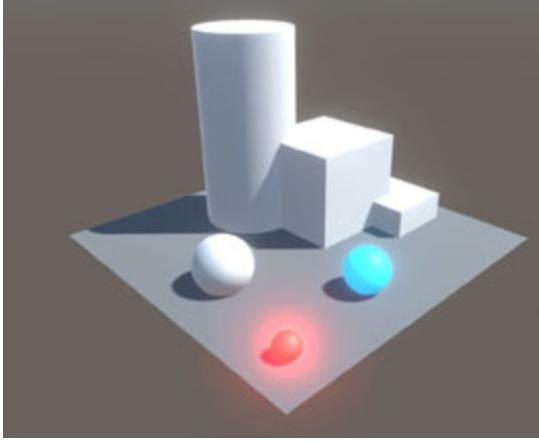
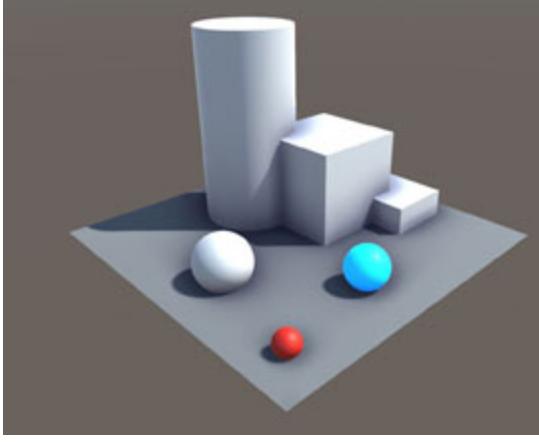
While all the steps up until this point have dealt with assets that get drawn in 3D Space, we're going to learn about the systems we can use AFTER the scene is rendered.

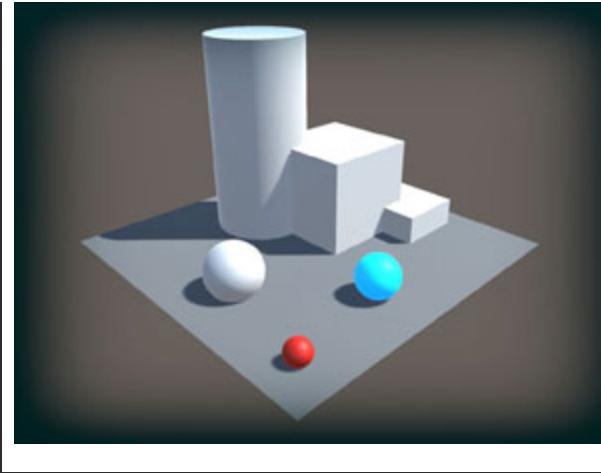
Specifically, we'll be learning about **Post Processing** - a pipeline of several effects that get applied once the 3D objects have all been processed. Post-Processing is a great way to add polish to your game's graphics without requiring new assets.

1. Before we get started, let's revisit the Package Manager and start importing the Post Processing package from the Unity Registry ([figure 11.41](#))

Figure 11.41: Post Processing is a core Unity system, but much like the other Art Tools, you'll need to install it through the Package Manager.

2. With that installed, you now have all the systems and effects you'll need to add Post Process effects to your camera. Let's take a moment to go over some of the more impactful Effects you can make use of...

	Depth of Field This Post-Process effect will blur foreground and background elements, ensuring the player is paying full attention to the most important objects in the scene. The effect is calculated based on the following parameters... <ul style="list-style-type: none">◦ Focus Distance◦ Aperture◦ Focal Length◦ Max Blur Size
	Bloom The Bloom effect will take the brightest parts of the rendered scene and give them a soft, ethereal glow to soften the scene. Additionally, it will put a substantial glow around any materials that have an Emission color or texture. Bloom is primarily calculated based on these key parameters... <ul style="list-style-type: none">◦ Intensity◦ Threshold◦ Diffusion◦ Color
	Ambient Occlusion This is an effect that simulates the subtle shadows that naturally occur at the meeting point of two surfaces (corners of a room tend to naturally exhibit this effect). The key parameters for AO are... <ul style="list-style-type: none">◦ Mode◦ Intensity◦ Color
	Vignette



A Vignette is a soft shadow that will border the rendered image. This is a way to gently draw the viewer's attention to the center of the screen. Because photos taken with older cameras exhibited this effect, it's also associated with a 'retro feel'. Parameters you can alter here include...

- Mode
- Color
- Center
- Intensity

Table 11.5: Several of the most impactful effects available through the Post Processing system.

3. There are several other effects you can add - Motion Blur, Grain, Color Grading, Lens Distortion, and Chromatic Aberration - but the ones listed above are by far the most useful, taking simple scenes and instantly making them look professional.
4. To add some Post Processing effects to our game, we'll need to make a Post Processing Profile. Create a new project folder called PostProcessing Data. Use the right click menu and select Create > Post-Processing Profile. Once that's made, go through and add the following effects.

Figure 11.42: Our Post-Processing Profile, with all the effects that we'll be hooking up.

With the effects added, you'll need to go through each and set up the proper values. Use [table 11.6](#) to fill out the Post Processing effects for our demo scene...

Ambient Occlusion	
Mode	Multi Scale Volumetric Obscurance
Intensity	1
Thickness Modifier	3
Vignette	
Mode	Classic
Intensity	0.4

Bloom	
Intensity	3
Depth of Field	
Focus Distance	8
Aperture	12
Focal Length	190
Max Blur Size	Large

Table 11.6: The parameter settings for our Post-Process Effects.

5. The next step is to add a **Post-Process Layer** to the **Main Camera**. This is a lightweight component that also allows you to enable Anti-Aliasing on the rendered scene. Once attached, the component should resemble [figure 11.43](#)

Figure 11.43: The Post-Processing Layer on our Main Camera.

6. The last step is to also add a **Post-Process Volume** to our Main Camera ([figure 11.44](#)). Once attached, make sure the Volume is set to **Is Global**, and set the **Profile** to be the one we created previously.

Figure 11.44: The Post-Processing Volume on our Main Camera.

7. If everything was done correctly, you'll now see all the effects being displayed in both your Scene and Game View windows, as shown in [figure 11.45](#)

Figure 11.45: Our demo game with some freshly polished Post Process effects!

Conclusion

Up until this chapter, we've spent all our energy on learning game systems and making prototype assets. This is as it should be at the start of a project - build your systems with the 'white-box' mentality so you can focus on creating fun, unique gameplay.

At some point, however, white-boxes get *really* boring, and it's time to give your game a Graphical Upgrade. The tools and concepts you've learned in this chapter do a great job getting you started on adding that visual polish every game needs.

And whether you're making your own assets, outsourcing new models, or kitbashing levels with assets from the Asset Store - feel confident that Unity gives you all the art tools needed to make your game stand out from the rest!

Questions

1. Looking back on the previous chapters, what temporary asset would you most enjoy recreating in ProBuilder?
2. If you have a game in mind, did you see any Asset Packs that matched the vision? Would you prefer to download an asset pack from the Unity Store, or create your own assets?
3. While it's tempting to learn art tools and manage Asset creation yourself, there is also a downside to juggling both the design and art tasks. What are the benefits of managing both? What are the drawbacks?
4. You now have all the tools to make a game in the visual style of your choice. What kind of art style do you most enjoy in your games? Cartoony? Realistic? Retro? If you made a game, would you want to make it in that style?

Key Terms

- **Meshes:** The 3D models of characters, enemies, items, and environments. These would traditionally be created using external modeling tools, but can now be created directly within the Unity Editor.
- **Materials:** The texture data for how the surfaces of a mesh are rendered. Materials control the shininess, bumpiness, color, and overall feel of rendered models.
- **ProBuilder:** Unity's primary tool for in-editor asset creation.

- **Vertex:** The points in 3D space that make up a mesh. Use ProBuilder to edit vertices for precise tweaks to an object's shape.
- **Edge:** The lines between the vertices that make up a 3D mesh. Use ProBuilder to change the position and rotation of edges that make up an object.
- **Face:** The side of a 3D mesh, made up of 3 or more Edges. Select and move these when making broad changes to the shape of an object.
- **Smoothing Groups:** A way to group faces together to hide the edges between them. Ideal for complex objects where multiple faces need to look like a single, unified element.
- **Pivot Point** - The point at which an object is moved, rotated, and scaled. In most cases, you'll want this in the center of a newly created mesh.
- **Grid Snapping:** Turn on Grid Snapping to make it easier to line up and place objects - especially blocks and cubes.
- **Texture Tiling:** How many times a texture repeats across an unit. The smaller the value, the larger a tile will look.
- **Kitbashing:** Building new creations from existing, premade assets. Use the Asset Store to download entire packages created for this purpose.
- **PBR:** Physically Based Rendering - a real time rendering model that attempts to create a realistic looking render of a 3D scene.
- **Skybox:** A texture that surrounds a scene. This is seen by the player in the background of an environment. It is also used by the rendering system to simulate reflected light in the scene.
- **Distance Fog:** When enabled, distant objects will transition to a flat color. Simulates a foggy environment, as well as the artist concept of 'Atmospheric Perspective'.
- **Post-Processing:** Effects that are applied to the rendered scene image.
- **Anti-Aliasing:** Smoothing of pixels, so you don't see jagged lines on the edges of 3D objects.
- **Ambient Occlusion:** A Post-Processing effect that simulates the subtle shadows you get in corners, where one surface meets another.

- **Distance Blur:** A Post-Processing effect where distant objects, as well as objects close to the camera, will be blurred. This allows the player's eyes to relax and focus entirely on the midground - where the game action tends to be occurring.
- **Bloom:** A subtle glowing effect on bright parts of a scene. Can also be used in conjunction with Emissive materials to make glowing effects (laser beams, fire, etc).
- **Toggle Effects:** A dropdown menu that lets you turn visual effects on/off as you edit your game.

CHAPTER 12

So Many Particles

There's no doubt that great video games balance the importance of gameplay and graphics in equal measure. Fun, engaging interaction coupled with clever, unique visuals will draw players in and keep them hooked.

Particle Effects are a perfect example of this fusion: using graphics and animation to show players the power of an attack, mystique of an environment, or simply draw the player's eye to something important.

There are a multitude of ways to make use of Particles in your game, and we're going to cover some of the highlights.

Structure

In this chapter we will discuss the following topics:

- About Particle Effects
- Particle Systems
- Environmental Particles
- Weapon Effects
- Explosive Effects
- Other Examples

Objectives

In this chapter, you'll learn all about Unity's Particle System - a way to create various effects for different situations. We'll start by creating some simple Environmental particles to learn the basics. We'll then move onto Weapon effects and Explosions as we get more comfortable with the more complex parameters available.

We'll end the chapter by downloading some additional samples and resources to learn from, giving us a well-rounded understanding of why, and how, particle effects can be utilized in game development.

About Particle Effects

Particle Effect systems use lots of small images or mesh objects, spawned with different sizes, shapes, speeds, and colors, to create a cohesive and dynamic visual effect. The fundamental skills you learned in the **Animation** chapter will be crucial here, since building VFX is an exercise using **Shape**, **Speed**, **Lifetime** and **Color** to build your intended effect.

Let's take a campfire effect, for instance. You'll need to have particles representing a flickering flame, the smoke plume, and some embers that occasionally rise up from the flame ([figure 12.1](#)).

Figure 12.1: Fires and explosions are a perfect use of the Particle Effect System.

Each of these layers has a different look and feel. The flame will consist of hot colors (whites, yellows, reds), mostly stay in place, and have a lifetime of less than a second. The smoke will use transparent gray and black colors to make a hazy plume that rises from the flame. Last, you will need embers: tiny, brightly flickering sparks that drift away in the direction of the smoke.

Any effect you want to make can be broken down like that, and Unity's Particle System gives you everything you need to make these effects a reality.

Filling the 3D Space

When creating an interesting, atmospheric 3D environment, particles can be used in various ways to add personality. Foggy clouds in a swamp, ash and embers around a volcano, or dust storms in a desert can all be created using particles.

For our scene, since it's a friendly, grassy environment, we're going to fill the area with **Dust Motes**: floating particles that represent the pollen and dust that fills the air, often without us even noticing it.

Create a new object, name it **ParticleFX_DustMotes**, and place it near the hero, as seen in [Figure 12.2](#)

Figure 12.2: Our New Particle Effect Starts with an Empty Object.

Once placed, you'll need to add a **Particle System** component to the object. After successfully placing the component, you should see several pink squares being emitted from the center of the object ([Figure 12.3](#)).

Figure 12.3: Adding the default Particle System component. The pink boxes give you a basic starting point for your effect.

You can see, from this list of parameters, that particle effects can be very complicated. To get the effect you want, you'll need to tweak values constantly. Helping to make this process easier is the **Particle Effect Panel**, a window in the corner of the Scene View that lets you control playback directly in the editor. This panel has several useful options, as listed in [table 12.1](#)

Pause / Restart / Stop	Buttons to control playback of the selected effect.
Playback Speed	The speed of the particle simulation. This is 1.0 by default, but can be tweaked in real time.
Playback Time	How long the particle has been playing for. This can also be tweaked, if you want to pause on a specific moment.
Particles	The number of particles currently being rendered.
Simulate Layers	Set this parameter to automatically play effects set to a given layer.
Resimulate	When selected, Unity will immediately apply changes from a Particle System to particles that have already been spawned.
Show Bounds	Display a yellow rectangle around the bounds of the Particle Effects currently being played.
Show Only Selected	Toggle to hide the unselected particles - useful for effects that get too complex.

Table 12.1: The Particle Effect Panel has all the options you need to play, pause, and debug the currently selected particle effect.

Let's start by getting the movement right on our Dust Motes. You'll notice that our particles are currently being emitted from a single point, then moving outwards. This is because the **Shape** of the emitter is set to **Cone**. With Environmental Effects, you normally want to fill an area with particles using a **Box** or **Sphere** shape. Lets select the Shape option and change the top parameter from Cone to Box ([Figure 12.4](#)).

Figure 12.4: The Shape parameters let you change how particles are spawned and what direction they're initially moving.

Environmental effects require a large shape to spawn within, so let's set **Scale** to 20, 20, 20. Let's also set **Randomize Direction** to 1, which tells the particle system to spawn each new particle in a completely unique direction.

This is also a good time to hide some of our other effects. Let's disable the Post-Processing effects, which will give us a clear view of how those Shape settings change the particle positioning and movement ([figure 12.5](#)).

Figure 12.5: With the Post-Processing effects disabled in the Scene View, we can clearly see how the particles are spawned within the Box shape.

Since Dust Motes tend to move in a leisurely, subtle manner, we'll need to tone down the movement of our particles to match those intentions. Go to the **Start Speed** parameter, and change the input to something more organic, more *random*. Click the small downward arrow, and change our mode of input from **Constant** to **Random Between Two Constants**. This allows us to enter two values that become the range that the **Speed** parameter will randomly pick from ([figure 12.6](#)).

Figure 12.6: Since we want the Start Speed to be a random value, click the dropdown arrow and select 'Random Between Two Constants'. Two entry fields will appear. Set the Min and Max values for your randomized range.

With the speed randomized, let's also set a random range for some other parameters: the **Start Lifetime** and the **Start Size**. Select the **Random Between Two Constants** option and set the Minimum and Maximum values to match those in [Figure 12.7](#).

Figure 12.7: By randomizing particles, you ensure the effect feels natural. Without randomization, effects tend to feel mechanical.

Now that we have movement we're happy with, let's change the sprite from the pink texture to a small circle - Unity's default particle texture.

Staying in the **Particle System** of the ParticleFX_DustMotes object, scroll down to the **Renderer** parameters. Expand this panel, and next to where it says Material: None, press the small circle button to bring up our material options, as in [figure 12.8](#), and select the Default circle material that comes included with Unity.

Figure 12.8: Changing the material of a particle will result in an instant change in the scene.

A Tip for Simplified Searching: Now that we've downloaded several packages into our project, you may be overwhelmed by options in the Select Material popup. Make your life easier here by utilizing the 'Search' bar. Type in 'particle' and the list will instantly filter out any materials without the word 'particle' in it. Be sure to make use of the search bar on any large project you're working on.

You will notice an orange 'Selection Outline' that surrounds the particles. If you find this distracting, you can toggle the visibility of these outlines using the Gizmo Visibility popup, which you can unhide with the button in the upper-right corner of the Scene View window ([Figure 12.9](#)).

Figure 12.9: Disable the Selection Outline option in the Gizmo Visibility popup. This will remove the orange wireframe around our Dust Mote particles.

Our final step is to set the **Color** of our Dust Mote particles. Right now, our tiny circles pop in and out of the scene in a distracting, unpolished manner.

We can fix this in the **Color Over Lifetime** option. Click the **Color** parameter, which opens the **Gradient Editor**, letting you define the alpha transparency values, and color values, that will be applied over the life of the particle.

Figure 12.10: For this effect, we simply want to have the color fade in, then fade out. Nothing too fancy.

Click towards the top of the gradient box to add some additional alpha values. We'll want four keyframes in total, with values set to match [table 12.2](#)

Location	0%	15%	85%	100%
Alpha	0	50	50	0

Table 12.2: The Alpha values for the four keyframes in our Color Over Lifetime gradient.

One great thing about particle editing is that you can see your changes in action as they're made. Always be watching your scene to make sure the changes you're making are producing the results as you expect.

Now press **Play** to see your particles in action!

Figure 12.11: Our first particle effect - subtle, environmental Dust Motes!

We've filled the 3D scene with our first environmental particle effects: small, subtle 'dust motes'. If you're feeling adventurous, you can use particles to add more animation and life to your scene (with fog, clouds, and falling leaves).

Weapon Particles

While environmental effects will help immerse a player in your world, particles can also be used in other areas of gameplay.

Combat, for instance, is a great place to make use of particle effects. Using effects of various colors, sizes, and intensities will help convey the strength of a given attack.

For our purposes, we're going to upgrade our bullet object to something more exciting. Let's start by returning to the **Weapon Sandbox** scene.

Figure 12.12: These bullets were fine as a prototype asset, but now that we understand particle effects, it's time to make them look awesome.

Because our bullet object is a prefab, adding an effect will be easy.

1. Open the **BulletObj_Basic** prefab for editing by finding it in the project folder and double-clicking on it.
2. Select the root object and add a particle system directly to it. You can do this in the Inspector Window, by selecting Add Component and adding a ‘Particle System’ ([Figure 12.13](#)).

Figure 12.13: Just like with our ‘Dust Motes’, successful adding a Particle System component will result in the spawning of many pink rectangles.

The new component will immediately start spawning particles, but you’ll notice that they’re fairly squished. This is because they’re getting the transform of the parent object.

3. Since you don’t want to be counter-scaling our effects, let’s unsquish them by resetting the parent’s scale to 1,1,1. Once reset, the bullet and particles should look similar to the screenshot in [Figure 12.14](#)

Figure 12.14: Our un-squished bullet. Always make sure your parent object is at a proper scale before authoring particle effects.

Focusing on the movement of the particles, there’s a subtle issue with the way these squares are being spawned. Because the Z-axis is ‘forward’, and because these particles are moving outwards into the Z direction, what we have is a bullet spawning particles that spray out *ahead* of the bullet.

4. Since we want our particles to act as a *tail*, let’s make sure our particles are moving in the *other* direction. We can accomplish this with a **Negative Speed**, as shown in [Figure 12.15](#)

Figure 12.15: With a negative speed, the particles will now create a trail behind the bullet.

We now have movement that we’re happy with, so lets switch away from the Pink Squares and over to the particle sphere we used for the Dust Motes.

5. Go to the Renderer and switch **Material** to Default-Particle. We’ll also change the **Render Alignment** settings while we’re here. This

changes how the particle is rendered in relation to the player's camera ([Figure 12.16](#)).

Figure 12.16: By default, billboards will face the camera. For our bullet trail, however, we want them to be aligned to their local transform. This will allow us to stretch them in interesting ways, while ensuring they're always visually aligned.

6. Let's tighten up the spread of these particles, which are going far too wide to feel like a proper *tail*. By tweaking the **Lifetime**, **Size**, and **Rotation** ([Figure 12.17](#)) we can get these particles to line up nicely.
7. Start with setting lifetime to 0.3. Because we are now rendering based on the local space, you'll notice the particles are difficult to see from the side. The changes we then make to **3D Start Size** and **3D Start Rotation** will resize and rotate the particles so they're properly visible from a side view.

Figure 12.17: Tweaking the Lifetime, Size, and Rotation values of our bullet trail.

8. Fixing the spread also requires a change to the emitter **Shape**. Open the Shape sub-panel and change the **Angle** and **Radius** to match the values in [figure 12.18](#)

Figure 12.18: By setting the Angle to 0, and the Radius to 0.02, we're forcing the emitter to spawn the trail in a more uniform direction.

9. Our particles are starting to be obscured by our bullet mesh, so let's turn that off by disabling the **Mesh Renderer** component.
10. Let's also set a **Size over Lifetime** curve, which will give us direct control over the size of our particles from the time they spawn to the time they're destroyed. Enable and open the Size over Lifetime sub-panel and click on the **Size** curve. This will open this curve in the lower right corner of the editor, in a window called **Particle System Curves**. Since we want the particle to get smaller over time, let's click the preset on the bottom right of the panel ([figure 12.19](#)).

Figure 12.19: Use the curve editor to have the size decrease from 1.0 to 0.0 over the lifetime of the particle.

11. Our tail is sizing nicely, but it feels a bit *sparse*. By spawning more particles, it will increase the density and make our tail feel more substantial. Select the **Emission** panel and raise the emissive **Rate over Time** to **50**, as shown in [figure 12.20](#)

Figure 12.20: Raise the Emission rate to 50 to thicken up the density of our effect.

12. It's now time to give the particles some interesting colors to interpolate between. Select and open the **Color over Lifetime** sub-panel and open the **Gradient Editor** ([figure 12.21](#)). Add some color values to blend between - preferably in the *blue* family of colors, to match the feel of our bullet material we've been using.

Figure 12.21: Color plays a big role in the appeal of visual effects.

13. Since effect creation is an exercise in iteration, now is probably a good point to test our updated bullet. Press **Play**, grab the blaster weapon, and fire off a few shots.

Because we're using the default circle texture, the trail is feeling a bit *soft*. Luckily, there are a handful of tweaks we can make to improve it.

14. First, let's re-add a bullet mesh, but this time it'll be a child object, allowing us to alter its scale without affecting the particles. Add a 3D Sphere as a child object and call it **BulletMesh**. Disable the Sphere Collider and assign it to use the matBullet material. Set its scale to be **0.1, 0.1, 0.4**, as shown in [figure 12.22](#)

Figure 12.22: A mesh object for the particle trail to emit from.

15. Second, let's deal with the *overall bulkiness* of the effect. By making it thinner, and giving it a more interesting tail, we can ensure the visual shape does a better job matching the bullet's collider. Head to the top of the **Particle System** component and tweak the **Start Lifetime** and **3D Start Size** parameters to get the trail looking thinner and a bit sparser on the end

Figure 12.23: By randomizing a wider gap in lifetime, we'll have a trail where some particles will linger (instead of all being destroyed at the same location).

16. We should also add some random rotation to our particles. This simply adds variance to the end of our tail, which will make it look a bit more interesting.

Figure 12.24: Randomize the rotation to give the end of the tail some appealing variance.

Our final bullet-related task is related to *light*. We have all these bright particles being emitted, but that brightness isn't currently conveyed on the surrounding surfaces.

17. We can fix that, however, with a **Point Light!** Point Lights will apply light to a given area based on a radius, intensity, and color. Use the right-click menu to add a **Light > Point Light** child object to our bullet.
18. Once added, bring up the Inspector Window and set the Point Light parameters to match what's listed in [figure 12.25](#)

Figure 12.25: A teal Point Light will make our bright particle effects look more convincing.

With our particle tweaks made, it's time to press Play and test our updated bullets. They should now cast a light on the surrounding objects and emit a nice, thin trail of particles behind it ([figure 12.16](#)).

Figure 12.26: Taking out enemies with our updated bullet effect!

With our bullets looking considerably better, it's time to think about other ways we can utilize particle effects.

Explosive Effects

Another Gameplay system that can make use of particles is the area of **Death Effects**. These are the animations that let the player know they've successfully vanquished the foe. Especially after a boss battle, having a cool

‘fade to dust’ or ‘massive explosion’ to reward the player is a great use of visual effects.

Let’s take what we’ve learned and go with that second example: a big, impressive explosion.

1. Stay in the **Weapon Sandbox**, make an empty game object called **ParticleFX_Explosion**.
2. Add a Particle Effect component and set its **Shape** to be **Sphere** ([figure 12.27](#)).

Figure 12.27: The humble beginnings of our epic explosion.

3. Explosions are all about color and timing, so let’s set the Renderer to use the Default-Particle material, and set the **Color Over Lifetime** to fade from ‘Fire’ colors into ‘Smoke’ colors.

Figure 12.28: Our explosive core needs a fire-to-smoke color gradient.

4. When making fire-related particles, you’ll often make use of this ‘fire-to-smoke’ color gradient. Use the values in [table 12.3](#) to produce a nice hot flash, followed by reds, then blending into smoke. You can also store this for later use by clicking the NEW button under presets (as seen in [Figure 12.28](#)).

Location	0%	7%	15%	30%	60%	100%
Color	White	White	Yellow	Red	Gray	Black
Alpha	255	-	-	-	255	0

Table 12.3: The basic steps of our color gradient. Tweak the colors until you’re happy with them.

5. With the colors set, it’s time to fix the spawning and movement of these particles. At the top of the Particle System component, adjust the parameters listed in [table 12.4](#)

Start Lifetime	0.5	1.5
Start Speed	1	
Start Size	4	

Table 12.4: The Lifetime, Speed and Size of our explosive core.

6. We're getting closer. The animation and size looks better, but it just keeps looping ([Figure 12.29](#)).

Figure 12.29: Our explosion effect in its current ever-looping form.

7. By default, particles are set up to keep spawning, on loop, forever. The number of particles set in the **Emission** parameters is the number spawned per-frame. There are times you want to spawn a given number of particles and be done. Since we want a burst of particles when the explosion happens, but no more than that, we'll need to make use of **Bursts**.
8. Open the Emission sub-options, reduce the **Rate over Time** to **0**. Add a new **Burst** entry that emits **Count: 30** of this particle at **Time: 0.01**, as shown in [Figure 12.30](#)

Figure 12.30: We want our explosion to spawn a bunch of particles at once. To do that, we'll be using emission Bursts.

You should now see a burst of explosive color that fades to smoke. If you sit long enough, you'll eventually see this effect happen again. That is because the entire effect is set to **Loop** with a 5 second **Duration**. This is OK, since it helps speed up the iteration process.

We're done with the core, but our explosion is far from finished. The next part of the effect are generic, fiery embers that fly in all directions.

9. Create an empty child object called **BouncingEmbers** and add a Particle System component to it. Use [Table 12.5](#) to fill some of the basic parameters.

Table 12.5: This is the first time we've used a Gravity Modifier. Positive values will cause the particles to drop to the floor at various speeds.

10. Once the base parameters have been set, you should have a shower of tiny pink lines spraying from our explosive center ([Figure 12.31](#)).

Figure 12.31: The start of our explosive Bouncing Embers.

The animation we're looking for is the following...

- Emit at the moment the explosion occurs
- Spray upwards
- Apply rotation over time
- Collide / bounce off objects in the world

All of these are doable - we just need to set the proper values. Staying in the Particle System component, adjust the parameters listed in [table 12.6](#)

Table 12.6: The main parameters for our Bouncing Ember particles.

It's time to get rid of the 'pink rectangles' and set the color to something more 'ember-like'.

1. Set the Renderer to use the default particle material (the default circle that we keep using) and set up the **Color over Lifetime** to match the Alpha and Color keys listed in [table 12.7](#)

Location	0%	20%	40%	60%	80%	100%
Color	Yellow	-	-	-	-	Red
Alpha	255	30	220	20	180	0

Table 12.7: Our embers should use these color gradient keys to have a nice flickering glow.

2. Whenever you need a particle that pulses, flickers, or glows, simply add **Alpha** keys that alternate between opaque and transparent ([figure 12.32](#)). By fading in and out, you can use particles to simulate lights, embers, fireflies, and countless other effects.

Figure 12.32: Use Alpha keys (the markers on the top of the gradient) that alternate between opaque and transparent to simulate a flickering effect for the embers.

3. A final bit of polish for these embers - **Trails**. Go to the Trails panel, activate and expand it, then enter the parameters listed in [Table 12.8](#)

Table 12.8: These trails will inherit the ember's color, as set in the Color over Lifetime settings.

4. The last layer of our explosive effect is a Center Fireball - something to represent the flames that tend to accompany any self-respecting explosion. Make an empty child object under the object ParticleFX_Explosion. Name it **CenterFireball**, and add a Particle System component to it.
5. Press **Restart** on the **Particle Effect Panel** to make sure you see some Purple Particles. Once verified, it's time to play with the particle parameters ([Table 12.9](#)).

Table 12.9: A burst of fire in the center of our explosion.

6. Like with the previous layers, it's necessary to set a red-hot color gradient for our fire particles. Bring up the Color over Lifetime panel and fill the keyframes, as listed in [table 12.10](#)

Location	0%	10%	100%
Color	White	Orange	Red
Alpha	255	-	0

Table 12.10: The colors of our center fireball.

This color gradient should result in a core to our explosion that look hot, colorful, and only lasts momentarily ([figure 12.33](#)).

Figure 12.33: Once set, this color gradient will ensure our explosion has a hot, fiery core that slowly fades out.

One useful set of parameters is the **Lights** panel. It can be used to spawn extra lights in the scene to really sell the brightness of our explosion.

1. Simply add a Point Light object as a child object and name it **Explosion Point Light**. Set its Range to 50, Intensity to 2, then

DISABLE the object ([Figure 12.34](#)). We'll be enabling it through our particle effect parameters.

Figure 12.34: Make a point light to spawn, then disable it. Our particle system will be in charge of spawning it into the scene.

2. Once the point light has been made, and disabled, you can connect it to the CenterFireball Particle System. Select CenterFireball and scroll down to the **Lights** panel. Select it, and set the **Explosion Point Light** object to the main **Light** parameter ([Figure 12.35](#)).
3. Since Point Lights can be fairly expensive, we'll also want to lower the **Maximum Lights** parameter to **3**, which will limit the number of spawned lights ([figure 12.35](#)).

Figure 12.35: Our explosion will now properly light up the objects around it.

We now have an epic explosion, complete with a fireball core, a spray of embers, lingering smoke, and some spawned point lights ([figure 12.36](#)).

Figure 12.36: Ending our hand-crafted examples with a literal BANG!

We were able to do a lot with the default particle texture. The soft circle was a great way to learn the fundamentals of the particle system.

However, if we're going to make particle effects that rival other games, we need to create - or at least *download* - additional textures and materials. Without variety in shape, you'll struggle to make particles that feel substantially polished.

Figure 12.37: The magic of materials: both of these fireballs use the same emitter data, but the left effect uses the default circle, while the right effect utilizes custom materials and textures.

Luckily, Unity has provided a great resource, not just for additional assets, but to also explore everything possible with the Particle System. It's time to download the **Unity Particle Pack**.

The Unity Particle Pack

Like most tools provided by Unity, the Particle System is both accessible, yet deep - full of potential that can only be unlocked through education and practice. One of the best ways tap into that potential is by dissecting example scenes and effects, and Unity has provided both in the **Unity Particle Pack**, which is available for download in the Asset Store.

1. Unlike other asset packs, we'll be loading this one directly into a brand new project. Save and close your Unity project, then use a browser to visit assetstore.unity.com. In the top search bar, look for 'Unity Particle Pack', which should come up as a free asset, as seen in [figure 12.38](#)

Figure 12.38: Add the Unity Particle Pack to your assets.

2. Next, open the **Unity Hub** and start a new unity project called **ParticleSamples**. Once created, go to package manager ([figure 12.39](#)) and import the Unity Particle Pack.

Figure 12.39: Import the particle pack as a 'complete project'. This will overwrite existing assets, which is why we used a completely empty project.

3. Once imported, open the **Main** scene and press the **Play** button. You can now walk around to view a large assortment of sample particles, ranging from fireballs to dust storms, magical fireflies to teleportation effects.
4. Use the WASD keys to move around, and if you want more information about an effect, simply press the Spacebar. An effect breaking down will come up, giving you insight into the creation process ([Figure 12.40](#)).

Figure 12.40: Pressing Spacebar will bring up a paragraph talking about the creation of the effect you're currently looking at.

5. If you want a deeper dive into these effects, simply jump over to Scene View mode, select an effect, and pull up the inspector window

([Figure 12.41](#)). You can dig through the Particle System settings used, play with the values, and learn the specific components and parameters used to create these awesome effects!

Figure 12.41: Use the Inspector Window to dissect all the amazing visual effects in Unity's Particle Pack!

With the knowledge you gained in the first part of this chapter, you'll have a firm understanding of how, and why, these effects were made. Smoke plumes, embers, dust motes - use this scene to expand your knowledge, and feel free to utilize the textures, materials, and effects in any of your own projects!

Figure 12.42: Unity's Particle Playground gives you a wide assortment of effects to enjoy, inspect, and learn from.

Conclusion

Particle effects are the swiss-army-knife of game development. They can be used in countless different ways, from large bombastic explosions, to flickering sparkles that draw the players to something interactive. Particles can be used to set the mood of an environment: choking the player with thick fog or filling a magical forest with fireflies. And particle effects are a great way to show the power of the weapons being used by the player (and their enemies).

Unity's particle system is also lightweight, letting you fill your scene with cool effects at almost no cost to your rendering speed.

Particles are a powerful, fun, and easy way to add personality, feedback, and polish into any game.

And while Unity makes it fun and easy to focus on making a great looking game, you should always be thinking about the core gameplay that the player is engaging with. Lets turn our attention away from graphics and back to gameplay, where we'll be learning about all the ways – outside of awesome graphics – that you can keep a player engaged and entertained. It's time to learn about the design concept of *Player Progression*!

Questions

1. The most obvious use of Particles is for weapon effects and big, impressive explosions, but there are plenty of other uses. What are some non-combat situations where particles can be used?
2. Particle emitters can spawn mesh objects as well as particle quads. What cases would be useful to spawn mesh-based particles (opposed to flat quads)?
3. Why is it good practice to randomize lifetime and speed values?
4. You can control the color of your particle by the colors present directly in the image texture used. What is another way to adjust the color of your particles over the lifetime of the effect?
5. True or False: By default, particles will collide with physics objects in the scene.
6. Why is a particle texture so important to the look of your effect?

Key Terms

- **Particle:** The objects that get spawned by a particle system. Can be a billboard or mesh.
- **Billboard:** A flat plane with a texture on it. By default, billboards will always face the camera position.
- **Particle System:** The main component used with creating particle effects. Placing a ‘Particle System’ script on any game object will let you emit particles from that object.
- **Particle Effect Panel:** A window that lets you control the playback of the selected particle effect. This panel allows you to easily create particles directly in the editor..
- **Environmental Effects:** Particle effects that set the mood of an area. Fog, dust motes, and other particles fill and give personality to empty 3D space.
- **Weapon Effects:** Effects that play when a weapon is used. Variance in effects can show differences in strength and attack type.

- **Lifetime:** How many seconds a particle will stick around until it's destroyed.
- **Speed:** The speed at which the particle will move in its set direction.
- **Negative Speed:** There are times when you need particles to move in the opposite direction. You can do this a handful of ways (shape rotation, etc), but the easiest is to simply set speed to be negative.
- **Scale:** The size of the particle when spawned. Can be set in 2D space with a single value, or in 3D space along all three axes.
- **Rotation:** The particle's rotation, either in 2D or 3D space.
- **Color:** The color of the particle, set when emitted with the 'Start Color' parameter, or to modify the color over the lifetime of the particle using the 'Color over Lifetime' options.
- **Emission:** The parameters that determine how particles are spawned by a given system. Can be spawned every frame, as well as in bursts at a specified time.
- **Shape:** The shape that particles are emitted with. There are 3D options (Cone, Sphere, Hemisphere, Box, Mesh) or 2D options (Circle, Edge, Rectangle). Each shape also has its own unique set of parameters to play with.
- **Renderer:** Parameters specifying what particles are spawned, including materials used and what type of object is spawned (Billboard, Mesh, etc).
- **Randomization Options:** Almost any parameter in the Particle System can be set to a flat value, or set to randomly pick a value from a range of values. Click the small arrow to the right of any parameter to pick a randomization option.
- **Texture Sheet Animation:** Particles can be made using frames of animation from a texture sheet (a grid of animated images). Use these options to specify a sheet and how you want the frames to animate.
- **Collision:** A set of parameters that specify physics interactions. Use the parameters for particles that should interact with, and bounce-off, colliders in the scene.

CHAPTER 13

Mastering Player Progression

Video games are interactive experiences that lead the player on a journey from the *Unknown*, to the *Known*, and eventually to *Mastery*. Players start a game with limited knowledge of the systems, rules, and world, but by the end of the game, they will have mastered these and achieved victory.

This growth, from *pupil* to *pro*, is known as **Player Progression**. As the designer of your game world, it's up to you to teach the player everything they need to achieve victory. How to use navigate the UI. How to interact with the world. How to talk with friends, and how to fight off foes. You'll need to help the player grow in knowledge and skill, giving them the tools needed overcome the challenges presented.

Every game will manage this progression in its own unique way, but there are several fundamental design concepts we can apply across any genre. Whether you're making a side scroller, a strategy game, an RPG, or a puzzle game, it's vital that you understand and master this responsibility.

As much as game design is about making something *fun* and *exciting*, it's also about *teaching* and *testing*: crafting a compelling experience, where the player can confidently progress from student to master.

Structure

- The Difficulty Curve
- Onboarding
- Tutorials
- Teaching through Level Design
- A Simple Test
- Mastery of Knowledge
- Progression Trees
- Achievements
- New Game +

- World Map

Objectives

In this chapter, we'll be learning about the various ways to guide the player as they progress through your game. We'll cover the concept of the Difficulty Curve, and the various ways you can use it to craft an experience that alternates between teaching the player, and testing the player.

We'll cover some level design specifics, introducing a new gameplay mechanic to our player, then ramping up the difficulty.

We'll also get into other theories and applicable systems (Pacing, Onboarding, World Maps) that will ensure you have all the tools you need to design a great game with strong player progression.

The Difficulty Curve

When considering **Player Progression** - and the subsequent balance between teaching and testing - one of the most important concepts to consider is the **Difficulty Curve**. This is a mental model where we map the **time** spent playing against the **difficulty** of a given level, area, or challenge. Early trials should be considerably less difficult than your ‘endgame’ challenges, but the player should always have an engaging obstacle to overcome. It’s a tricky, yet vital, design skill to master. Let’s start with the most basic of Difficulty Curves - **Linear** progression. This would have the player dealing with a steady set of challenges that gradually get harder as they play, as shown by the difficulty curve in [figure 13.1](#).

Figure 13.1: A ‘Linear’ Difficulty Curve, where each dot is a ‘Level’. Note that the earlier levels are easier than the later levels.

This gradual, steady progression from Easy to Hard is a great starting point, but this ceaseless climb can also feel *grueling*. We can fix this by incorporating slight upticks in difficulty, followed by restful periods. These are called **Peaks** and **Valleys**, and as shown in [figure 13.2](#), they’re a great way to turn a slow grind into something rhythmic and interesting.

Figure 13.2: A Difficulty Curve with Peaks and Valleys, which cycle between moments of intensity and moments of rest.

The important thing to remember is that, while the above chart gives a nice, generalized template to follow, each game should be different. You may want a huge difficulty spike somewhere in your game, or perhaps a planned, exaggerated lull before an intense area. If your game is story driven, you'll often want difficulty peaks punctuating a chapter in your story. An adventure following a 3-act structure, for instance, may have a curve that looks closer to [Figure 13.3](#)

Figure 13.3: You can use difficulty spikes to punctuate the end of a chapter or act.

The Difficulty Curve is a great tool for visualizing the intended Player Progression for your game, but - unfortunately - it's a concept that doesn't easily translate into *hard data*. There's no Unity plugin to take a difficulty curve and turn it into a game that gets hard at peaks and easy at valleys.

There are, however, game design rules that directly map to the concept of Peaks and Valleys, which we'll get to shortly.

A Note on Retaining Your User Base: Player Retention - the ability for your game to keep players interested - directly correlates with the balance struck by your Difficulty Curve. If the player is dropped into a game with an insanely high difficulty at the start, players will quit early in frustration. The flip side is also true. If a game is too easy, and the player doesn't feel the satisfaction of overcoming interesting challenges, they'll leave to find a game that's more exciting. A well-balanced Difficulty Curve will result in a strong rate of Player Retention.

Managing Difficulty

Managing your **Difficulty Curve** is a concept that makes sense on paper, but can be easily mishandled in practice, especially if you're working solo. Over time, replaying the same starting levels is going to get both easy *and* monotonous. Your inclination is going to raise the difficulty of these starting levels to something more exciting - we can't have our first 10 minutes be boring, so the levels need to be updated, right?

Wrong! Changing these levels would be the incorrect choice. You should never balance your games' difficulty against the skills of the *person making the game*.

Always make changes only after several **Playtesting** sessions with outside testers. Most players will have no experience with your game during these early stages, so only tweak difficulty based on the feedback of that user demographic.

In [Chapter 18](#) “From Concept to Completion”, we’ll dive further into Playtesting, as well as another useful tool in gauging Player Progression: **Analytics**.

Onboarding and Tutorials

As a game designer, you have a responsibility to make sure your systems, mechanics and UI make sense to the player. When you’re close to a project, it’s easy to take certain things for granted: what buttons to press, what direction to go, how to perform a complicated action.

To incoming players, however, these concepts may prove difficult and frustrating to the player. And if there’s one sure way to see players abandon your game, it’s frustration.

There are two straightforward ways you can tackle the problem of ‘Teaching’ the player: **Onboarding** and **Tutorials**.

Onboarding will make use of in game prompts and text to explain how to interact with your game world. They will spell out everything plainly and will often be tied to an ‘Advisor’ character that’s walking you through the early steps.

And while these prompts are useful, they can quickly get annoying. Make sure Onboarding help is something the player can disable, and in general they should only appear in the first sections of your game. Once you’ve explained the basics, leave some mystery for the player to discover on their own, and use Playtesting to determine when Onboarding should stop.

Where Onboarding takes place within the main flow of your game, **Tutorials** are a way to create a safe area - away from the dangers of the main game - for players to learn the specifics of how your game is played. Typically, an option directly on the Title Screen, a Tutorial will present the player with a simplified version of the full game where all possible actions are explained. The player will have to follow a set of prompts, like “Pick Up the Key with (A)”, “Walk to the Door using the Left Joystick”, and “Press Start to open your Inventory”. The Tutorial will have several steps, where failure at one step will keep you from progressing.

While this can be viewed as a heavy-handed approach, including a Tutorial ensures that the player knows all the nuances of your game before jumping into the main adventure.

Onboarding and Tutorials are very UI heavy options that do a great job teaching, but what if you want a more *subtle* approach?

Teaching through Level Design

While communicating through prompts and tutorials to teach the player, you can also use several design tricks to communicate new knowledge without a giant block of text.

The key idea to this design practice is **Teaching through Level Design**: creating an environment where the player will naturally absorb and learn a concept without needing to be told. This area is typically free from major dangers, giving the player many attempts to learn the skill, tool, or idea or being taught. Once understood, the level is designed to let them progress with this new knowledge, which can be used to overcome the dangers ahead.

Take a simple concept of the **Bottomless Pit** in a platformer ([Figure 13.4](#)). Modern gamers will see a pit and instinctively understand that it needs to be jumped, and a misjump will end in a lost life.

Figure 13.4: Without Onboarding or Tutorials, early game designers had to be clever about teaching new concepts to the player.

There was a time, however, when players had to *learn* that mechanic. *Falling down a pit means you're not coming back*. Without onboarding prompts or a tutorial, this is something game designers needed to *show* the player.

In the first Super Mario Brothers, the game design team needed to be resourceful. They placed two enemies on the right side of the pit, with the player coming up to the left side. An unknowledgeable player may encounter this *giant hole in the ground*, get nervous, and stop for a few seconds. In those seconds, the two enemies would walk toward the pit, fall in, and never return. This was a clear sign to the player that *jumping the pit* was the preferable option.

Without a prompt, sign, warning or tutorial, the level designers were able to communicate through the *level itself* - teaching new players by utilizing the environment and enemies in clever ways. Again, while Onboarding and

Tutorials will teach, this *invisible* method of learning through level design is almost always preferable. **Show, don't Tell.**

Lighting the Way

Let's use what we've learned and teach the player a new concept in our game demo: **Lit Environments**. The player will have to use a new 'Torch' pickup item to navigate a level obscured by darkness. As the level progresses, new ideas and challenges will be presented to the player, teaching them how to use the torch through our level design.

Start in the WeaponSandbox scene, and let's use ProBuilder, Particles, and Point Lights to build our Torch. Create an empty GameObject, call it **WeaponObj_Torch**, then use everything you know to create a torch that our player can use, similar to the one shown in [figure 13.5](#).

Figure 13.5: Use everything we've learned in the previous chapters to model our new Torch!

A Note on Lighting Your Own Way: This torch object is a great opportunity to stretch your creative muscles again and build a Torch that is uniquely you. I'll be laying out the necessary components we'll need to add, but in terms of modeling a Torch object, it's up to you! It can be as simple as an upside-down cone, or as complex as a wrought-iron lantern. Get creative, get crafty, and have fun!

Because the torch is being used to light the environment, it needs to have a Point Light child object. In the hierarchy window, place a new Point Light under the WeaponObj_Torch, then set the parameters to match those in [Figure 13.6](#)

Figure 13.6: The parameters of the Torch's Point Light.

Since the Torch will function similarly to a weapon (where it will be picked up and held by the player), the other required components are the *Weapon*, *PickupItem*, *Rigid Body*, and *Collider*. Make sure the Collider has **Its Trigger** enabled, and the Rigid Body has **Use Gravity** disabled, as shown in [Figure 13.7](#)

Figure 13.7: Our Torch object with all the necessary components.

One last step for the Torch - we're going to add a new, empty script called **Fire Object**. Use the Add Component button to make and add this script, which will be used later to determine if a weapon is 'Fire' or not.

The second new object we'll be making is a Torchfire Block. This object, when 'lit' by the torch, will allow the entire scene to be visible for a few seconds, giving us a fun new system for the player to learn in this level.

Just like with the Torch, you have full creative control over the modeling here. It's suggested that you use a ProBuilder cube to make a block, similar in size to the Red Brick object we already made. Stack a Pyre of logs on top to visually indicate this is a flammable object, and add a particle effect object for a flame, similar to the Pyre made in [figure 13.8](#)

Figure 13.8: An example of a Torchfire Block. Use ProBuilder and the Particle System to make something similar, or completely unique - it's up to you!

The most important part of this object is a new script we need to create. Use Add Component to attach a **TorchfireBlock** script that will handle the collision between this and the torch, manage the toggling of the particle effect, and enable/disable the scene lighting as blocks are lit.

Once created, open the new script in Visual Studio, and add the following code...

```
public class TorchfireBlock : MonoBehaviour
{
    [SerializeField, Tooltip("Seconds until fire goes out.")]
    float _fireTimer = 0;

    [SerializeField, Tooltip("Seconds applied when re-lit.")]
    float _maxTimer = 15;

    [SerializeField, Tooltip("The fire particle effect.")]
    ParticleSystem _fireParticle;

    // number of blocks in the level that are currently lit
    static int s_numLitBlocks = 0;
    void Start()
    {
        EnableFireEffects(false);
        EnableSceneLights(false);
```

```

}

void Update()
{
    if (_fireTimer > 0f)
    {
        _fireTimer -= Time.deltaTime;
        if (_fireTimer < 0)
        {
            // fire has gone out
            AdjustFireBlockCount(-1);
            EnableFireEffects(false);
        }
    }
}

void EnableFireEffects(bool status)
{
    // grab the particle effect emitter
    // and enable (or disable) it
    var emit = _fireParticle.emission;
    emit.enabled = status;
}

void OnTriggerEnter(Collider collider)
{
    // did we collide with a flammable object?
    if (collider.gameObject.GetComponent<FireObject>())
    {
        // re-light the pyre
        if (_fireTimer <= 0)
        {
            AdjustFireBlockCount(1);
            EnableFireEffects(true);
        }
        _fireTimer = _maxTimer;
    }
}

static public void AdjustFireBlockCount(int change)
{
    // update number of 'lit' torchfire
    // blocks in the scene
}

```

```

s_numLitBlocks += change;
// enable / disable the lights based on
// the number of lit blocks
bool lightStatus = true;
if (s_numLitBlocks <= 0)
    lightStatus = false;
EnableSceneLights(lightStatus);
}
static public void EnableSceneLights(bool status)
{
    Light[] lights = FindObjectsOfType<Light>();
    foreach (Light l in lights)
    {
        if (l.type == LightType.Directional)
            l.enabled = status;
    }
}
}

```

To ensure the block works as intended, make sure it has the proper components ([Figure 13.9](#)): Rigidbody, Box Collider (for physical collisions), and a Sphere Collider (which triggers collision with the torch).

Figure 13.9: Our Torchfire Block, assembled with the necessary Components.

In the Rigidbody, use the Constraint parameters to lock both the Position and Rotation of the object.

Press the **Play** button to test it. If it's working correctly, drag the **TorchfireBlock** object into the Prefabs folder to use later. It's now time to build our level!

Teaching the Player

When first placed in an unknown situation, with a gameplay feature that needs to be learned, we want to teach the player about the feature through level design. The player will learn best when that learning is free from danger and distraction, so we need to clear the area of any secondary dangers. This ensures that the only struggle is with the specific system being taught.

Let's make a safe space for the player to learn about the Torch. Since we're teaching the player how to navigate dark levels by torchlight, the only challenge here should be advancing to the next area utilizing this new item.

1. Start by making a duplicate of the WeaponSandbox scene. For to **File > Save As**, then name the copy scene **DarkDungeonLevel**.
2. To get the 'darkness' effect we're looking for, we'll need to disable several lighting options in our new level. Open the **Rendering > Lighting > Environment** window ([Figure 13.11](#)), and find the Intensity Modifiers for **Environment Lighting**. Set it to **0**. Also set this parameter for **0** under **Environment Reflections**.

***Figure 13.10:** By turning off the 'Intensity Multiplier' of our skybox when lighting objects, we can give our level the extreme darkness required.*

3. To fully disable the skybox, we'll also need to tweak some parameters in our **Main Camera**. Select Main Camera, and let's set the **Clear Flags** to **Solid Color**, and set **Background** to pure black (0,0,0), as shown in [Figure 13.11](#)

***Figure 13.11:** Clearing the camera background to Solid Black will significantly raise the 'Darkness' Factor of our level.*

4. One last lighting adjustment will be to the main **Directional Light**. Select it, and tweak the Intensity (lower it to 0.7) and adjust the rotation to be almost straight down (85, 0, 0), as shown in [Figure 13.12](#)

***Figure 13.12:** Setting up the direction light for our 'Dark Dungeon' level.*

This will make the light look less harsh as it's toggled on and off by our Torchfire Blocks. Also, you can use this opportunity to disable the light and view the lighting results in the Game Window. If you did everything right, the only things visible should be the items with a Point Light attached to them (which will probably only be the Torch). Just make sure to re-enable the light so we can see the level we're designing.

It's time to create our 'Learning Area' for teaching the player about the Torch. You have some flexibility here, but the room should have the following features...

- The room should start with the player hidden in darkness, with the only visible object being the Torch.
- The room should be almost impossible to leave without the torch (I used a pit, with a simple jumping challenge to get out from).
- The room should be clear of any dangers. Put a box collider on the front-side of the area so players cannot die in a bottomless pit.

Figure 13.13: This is the ‘Learning Area’ I created - a low room that has to be climbed out of. In the editor, you can see the way out easily, but in the game, the only visible object will be the Torch that the player needs to grab.

As you can see in my example ([Figure 13.13](#)) this area can be as simple as a small room with a few blocks used to climb with. The challenge is not the platforming, but the fact that, without the Torch to light the way, the solution would be impossible to see, as you (mostly) see in [Figure 13.14](#)

Figure 13.14: Once in-game, the player will learn quickly that they’re going to need a Torch to light the way forward.

Once your scene is built, press the **Play** button. If your scene is similar to the example, your hero can run around the dark level, feeling a bit lost and claustrophobic. Only the Torch is fully visible, which naturally draws you to it. Once grabbed, it attaches to the player’s hand and becomes the key to finding a way out.

Moth to Flame: One of the most powerful Level Design concepts you can learn is using light to guide the player. In a dark situation, a player will instinctively move towards a light source. Use torches on the wall, or a distant ‘light at the end of the tunnel’ to guide a player with an invisible hand.

We’ve talked about this a few times, but it can’t be understated how important a safe area is at this point. Giving the player a micro-challenge that **ONLY** stresses the new mechanic will give them a stepping stone to greater, more exciting trials.

Figure 13.15: Keeping this starting area safe. Dying should be nearly impossible here.

Testing the Player

With the new concept learned, it's time to introduce a bit of danger to keep the player on their toes.

We'll also use this opportunity to include other ways for the player to use their new knowledge (and tools). For this secondary gameplay mechanic, we'll teach the player how to use Torchfire Blocks to illuminate an entire level for a short period of time.

Start by making a new area that would be difficult to see with only a Torch. This will probably need to be a platform positioned *within* jumping range, but *outside* the radius of the torch. A well placed enemy should also be used to increase the danger, as shown in [figure 13.16](#)

Figure 13.16: An example of the first test in our Dark Dungeon.

Again, you're encouraged to get crafty and design your own area, but take a moment to dissect the example above. It has a few ideas that you may want to use in your design, as listed below...

- The major challenge is that the path forward requires a jump that's almost invisible using just the Torch.
- Once the Torchfire Block is lit, the solution becomes visible, but there is also an Enemy and Bottomless Pit that first have to be overcome.
- The 1st block will extinguish before the player can reach the 2nd block. This teaches the player about the 'timer' associated with lighting the entire scene. They'll need to be careful - and quick - when navigating complex areas using these blocks

[Figure 13.17](#) shows this in practice. The path forward is, once again, obscured by the thick darkness. Only our new gameplay mechanic - the Torchfire Block - can light the way.

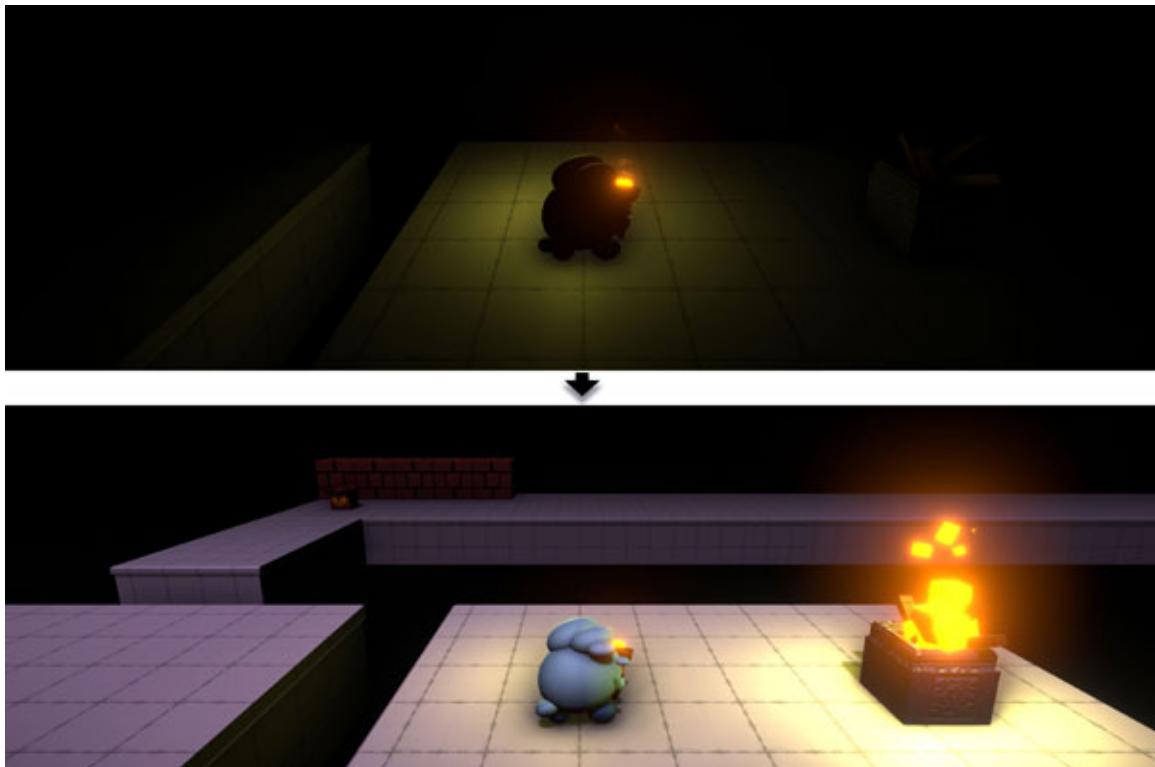


Figure 13.17: When the Torchfire Block is lit, the scene will be illuminated and the player will see a platform they would have missed in the dark.

Trials and Mastery

Now we will get to have some real fun. Our player understands the new mechanics, and has been tested a bit, so now it's time to throw some substantial challenges their way. In [Figure 13.18](#), we have a platform maze with several jumping puzzles, enemies, and Torchfire Blocks.

Figure 13.18: The player has to light their way forward using all the knowledge and skill they gained in the previous sections of this level.

You'll want to come up with several challenges like this to continue to challenge your player. Keep your ideal Difficulty Curve in mind as you build, creating escalating trials that continue to push the player forward.

Once you're happy with the white-boxed version of your level, be sure to apply materials and add particle effects to finalize your scene, as shown in [figure 13.19](#)

Figure 13.19: Our Dark Dungeon level, fully polished with ground textures and particle effects.

You now have a level that both *teaches* and *tests* the player - the two main tools of *Player Progression*. Designing levels this way ensures that every new area has a difficulty curve that is both engaging and challenging to the player.

And while this method of game design works for projects that require on interesting levels, there are plenty of titles that don't have specific areas to design. Sandbox games, titles that generate a random world to play in, require some other Progression tactics to engage, entice, and excite the player.

Pacing

Much like how the Difficulty Curve deals with the rhythm of challenges being presented, **Pacing** deals with the frequency of rewards to the player.

There are a wide range of rewards that can serve to build a satisfying gameplay Pace as stated below:

- Unlocking a new Gameplay Mechanic
- Learning of a new skill
- Gaining a new Item
- Encountering a new area
- Uncovering a new part of the story
- Meeting a new Character
- Accomplishing a Major Goal
- Defeating a Key Adversary

These rewards can either be the gaining/encountering of something new, or the payoff to a long-running quest. Either way, ensuring the player is always being presented with an interesting reward will keep them fully engaged. They will be excited, not just by the reward, but also at the idea of *future* rewards.

Strong Pacing will give rewards in a rhythmic way - think back to the Peaks and Valleys of the Difficulty Curve. The player should always feel like their gameplay accomplishments are being acknowledged. Rewards should be earned, and never feel cheap.

Weak Pacing is caused by too few rewards, or rewards that aren't exciting. New mechanics and areas are great, but a handful of gold or a slight upgrade to stats may feel *anticlimactic*. Spend time creating interesting rewards that keep the player excited for what's next.

Overwhelming Pacing is when you dump too much on the player at once. If rewards feel like a burden, you likely have a Pacing or Onboarding problem. Spread out the introduction of new mechanics to entice new players (opposed to scaring them off). Overwhelming players with cheap, uninteresting rewards is also a danger.

Progression Trees

When it comes to rewards, one of the best tools for exciting the player is a **Progression Tree**. This is a UI menu that maps out the various upgrades, skills, and unlockables that can be gained through continued play.

This is also a great place to balance pacing. Since most tree nodes have a point cost associated with them, you can tweak these costs to balance rewards and make sure the player isn't bored waiting for the next one (or overwhelmed from unlocking too much at once).

We'll be creating our own Progression Tree in the next chapter, which covers Advanced UI systems.

Achievements

One of the big additions to modern games is the concept of **Achievements**. These are awards handed out to players based on meeting specific criteria within a game. The major draw of Achievements, besides giving the player a secondary set of goals to accomplish, are their outward-facing nature. Achievements tend to be directly tied to a player's profile, with these awards being displayed for other gamers to see. They're a badge of honor to the most diehard players, so don't forget to include them.

New Game +

A great way to reward a victorious player is to include an extended adventure upon completing the game. With **New Game +**, players get to take their skills, knowledge, and love for the game and start off on a modified version of the adventure they just completed.

With relatively little work, you can create a 'Second Quest' that drastically raises the difficulty level. And since this is intended for the most hardcore players, you can really have fun with the challenges presented.

World Map

For anyone learning these Game Design concepts for the first time, the idea of juggling Difficulty, Retention, Pacing, Teaching, Testing, and Rewarding can be *fairly overwhelming*. It's one thing to make a small sample game. It's another to plan out a full adventure that players will find satisfying (and worth paying for).

Luckily, there's one gameplay system that can enforce an interesting Difficulty Curve, manage Pacing, provide Rewards, and even service the Story.

That system is the **World Map**.

A World Map is a scene that holds all the levels on it. The player can explore the map at their own pace, picking the levels they want to go to. Easier levels will be placed closer to the starting position, and harder levels will be placed further away. You can have levels that are inaccessible without completing earlier levels, or areas that aren't even on the map until a story event causes them to appear.

This system of progression has several positive effects, as listed below...

- You have tight control over the Difficulty Curve. The position of levels can directly correlate to the challenge provided, keeping the early levels easy and giving later levels a satisfying challenge.
- You can implement pre-requisite levels that serve as a **Skill Gate**. If players don't have full Mastery of a skill that is needed later, you can design a level with the singular purpose of teaching the player that skill. A skill gate level ensures the player always knows how to handle future challenges.
- A World Map allows you to fill your game with interesting world-building places and events. Cities, huts, friendly grotto's, and NPCs can perform story exposition in a way that feels natural and fun.

Let's take a moment to learn some of the tools provided by Unity for building our own World Map.

1. Create a new scene, name it World Map, and open it. Right click in the hierarchy Window and select **3D Object > Terrain**. This will place a basic **Terrain** object in the scene, as shown in [Figure 13.20](#)

Figure 13.20: Terrain starts as a large checkerboard plane, an empty canvas for us to sculpt.

2. When selected, you'll see that the Terrain object has a corresponding Terrain component. Under the title you'll see a row of 5 TerrainTool buttons.
3. Select the second tool: **Paint Terrain**. This is a multi-purpose tool that you'll use for various purposes. Select **Raise or Lower Terrain** from the drop down menu, which we'll use to sculpt our world.

Figure 13.21: You'll be using various brushes, with varying strengths and sizes, to raise and lower the Terrain geometry.

If you ever used a paint program before, you should feel right at home with the Terrain editing tools. by selecting a brush, and setting its strength and size, you can paint modifications to your world map's highlight.

4. Click anywhere to raise the height, or hold CTRL while clicking to lower the height. With these two interactions you can raise mountains, carve out valleys, create subtle Dunes, and everything in between.
5. Next, let's return to the drop down menu to select our next tool: **Paint Texture**, as seen in [Figure 13.22](#)

Figure 13.22: Use the Paint Texture tool to turn the giant checkerboard Terrain into something more natural looking.

6. The reason our world map shows up as a checkerboard pattern is because it has no specified Terrain Palette, which would list the textures that can be applied to this Terrain. Scroll down to the Layer Palette, and select the Add Layer button ([Figure 13.23](#)).

Figure 13.23: We need to add some Textures to our Layer Palette before we can start painting our World Map.

7. Depending on the Asset Packages you've added to the project, you should have some Terrain Layers to add. If you don't, however, you may need to make some from other textures in your project (or download some from the Asset Store). Press the Create button to use textures in your project to make a new **Terrain Layer** ([Figure 13.24](#)).

Figure 13.24: You can use any texture to make a Terrain Layer for your World Map.

- Once your Terrain Palette has a few textures to use, you will paint these textures just like raising/lowering land. Pick a brush, tweak strength and size, and the rest is up to you.

Figure 13.25: Painting terrain textures works just like raising/lowering land. Simply click on the layer you want to apply to the ground, then use a brush to paint it.

- The last of the tools we'll be learning is the **Paint Trees** tool. Select the third button in the Terrain component, the press the **Edit Trees** option. This will unhide a popup of the different tree prefabs assigned to the brush. Select prefabs you want to place, as seen in [Figure 13.26](#)

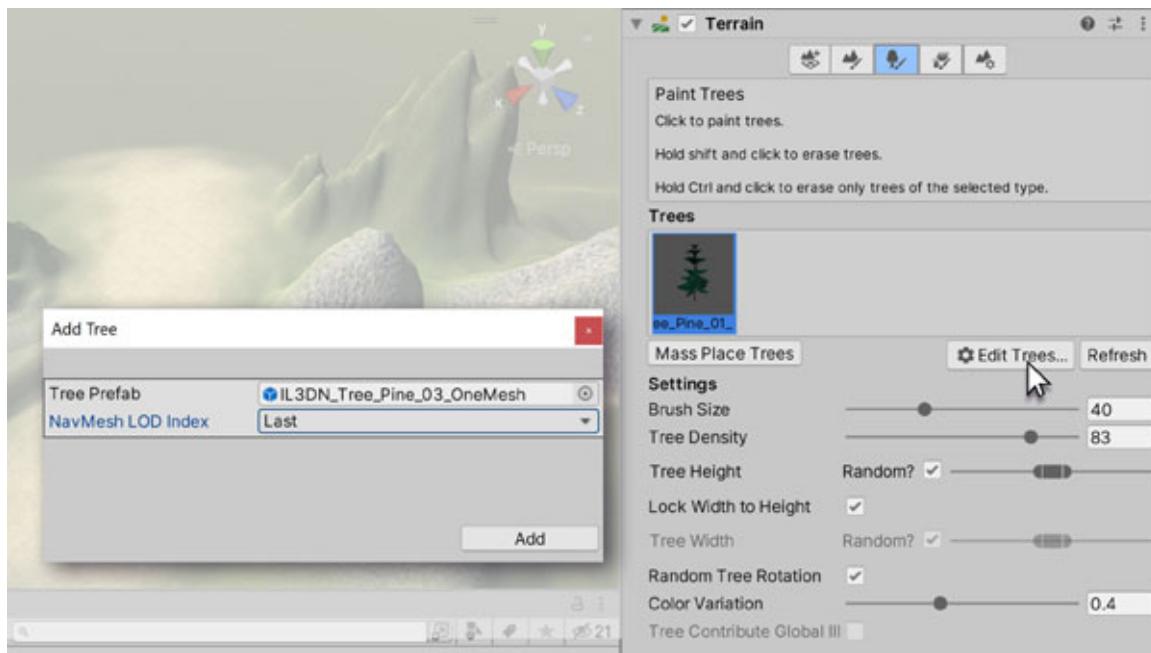


Figure 13.26: Painting Trees simply requires a few prefabs to be defined.

- Just like with the Raising/Lowering of terrain, and the painting of textures, placing trees uses a brush that will spawn trees in a given area. Simply click with the cursor, and the trees will be placed!



Figure 13.27: Our World Map, now covered in forests by using the Paint Trees tool.

11. With mountains, plains, and forests, you officially have the start of a World Map. By using ProBuilder, you can now make some Map Markers to place entrances to the various levels of your game. Throw on Post Processing and put your hero on the map ([Figure 13.28](#)), and you'll have a World Map that feels pretty great.



Figure 13.28: With your hero, some markers and post-processing effects, your World Map will feel like an environment worth exploring!

You now have all have the tools needed to get creative, sculpting your own map for players to explore. When it comes to player progression – and the management of difficulty and pacing – there are few mechanics as useful as a World Map.

Conclusion

Managing player progression is undoubtedly an advanced Game Development skill, But when designing a game to be played by the masses, it's also critical.

We live in a time when players are overwhelmed by game titles. Every new week there are hundreds of new titles filling the digital Video Game marketplace. If players try a game and aren't hooked in the first 10 minutes - either due to frustration or boredom - players are going to leave in search of another title to play.

It's your job to balance difficulty, manage pacing, and create interesting Rewards that keep the player invested, excited, and coming back for more. If you do everything right, the player won't be able to put down the controller.

In the next chapter, we'll be taking a step back from player progression to look at User Experience as a whole. By using Iconography, Typography, Advanced UI, and Level Design, you'll be able to ensure the first 10 Minutes of your game are amazing, and that the player is excited to see what comes next in your game. Let's learn the art of creating a great UX!

Questions

1. When talking about the Difficulty Curve, what do the terms ‘Peaks’ and ‘Valleys’ stand for?
2. Why is it important to have difficult portions of your game, followed by easier sections? How is this preferable to a difficulty curve that is always gradually getting harder and harder?
3. Why is it important to make difficulty changes based on player feedback, opposed to your personal playtesting sessions?
4. Onboarding and Tutorials are useful tools to teach the user how to play your game. How can you use the level design to teach gameplay mechanics?
5. When you first introduce a new mechanic, why is it important to keep this area as safe as possible?

6. Good pacing will give the player impressive rewards as they overcome challenges. What are the different types of rewards you can give a player? What are some of your favorite rewards from previous games you've played?
7. How does a World Map enforce both difficulty and pacing in a game?

Key Terms

- **Player Progression:** The growth of a player from novice to master. Game designers have to guide the player in a compelling way, balancing teaching and testing.
- **Difficulty Curve:** A way to chart the difficulty of a game against the time spent by the player. Earlier levels should be easier, later levels should be more challenging, and the difficulty should have a rhythmic rise and fall, with difficulty spikes followed by lulls in intensity.
- **Peaks and Valleys:** The rise and fall in difficulty across a Difficulty Curve. Peaks are points where the player is challenged considerably more than previously (boss battles, for instance). After an especially difficult section, they should encounter a difficulty valley - a period of rest as a reward.
- **Player Retention:** Your game's ability to hold the attention of the player. A solid balance between learning new skills and challenging the player will help to keep the player engaged and improve retention. Frustration with a steep difficulty curve, or boredom from a flat curve, will lower retention.
- **Onboarding:** Prompts that communicate with a player, especially early in a game, how the mechanics and UI work. Onboarding gives the player information and hints without being obnoxious.
- **Tutorials:** A separate, condensed slice of the main game, where the player walks step by step through the basics. A heavy handed way to teach the player, but fairly reliable.
- **Teaching through Level Design:** Using clever level design to show the player how new mechanics work (opposed to telling them through Onboarding or Tutorials). Make use of safe areas to focus the players attention completely on the new concept that they need to learn before moving on.

- **Testing the Player:** Once the player learns a new mechanic, it's important to test them. Whereas a safe area is preferable for learning, testing requires some actual danger to ensure the player has a strong understanding of a system, skill, or ability.
- **Pacing:** The frequency that a player is rewarded, either by new skills, new story elements, or any number of exciting events that keep them engaged.
- **Progression Tree:** A visual indicator of the rewards that will eventually be available for the player to unlock. These come in various forms (Skill Tree, Tech Tree, Upgrade Tree) but they're a great way to manage pacing and always have fresh mechanics for the player to enjoy.
- **World Map:** A sprawling map of the game world that the player can explore. This allows the designer to organize levels in such a way that it follows the peaks and valleys of a desired Difficulty Curve.

CHAPTER 14

UX

**“I’m so confused...”
“What do I do next?”**

These phrases of confusion and frustration are NOT what a user should be thinking when playing your game. At all times the player should know **What** to do, **Why** they’re doing it, and **How** it should be done.

A game is complex, and since you’re not personally available to explain these complexities directly to the player, you’ll use Unity (and its systems) to communicate for you. You’ll need to use sound, UI design, game flow, particle effects, animation, and every other system at your disposal to ensure the player understands the what’s, why’s, and how’s of your game.

This process of breaking down the complexities of your game, and presenting them to the player in an easy to digest (and hopefully subtle) way, is known as crafting the **User Experience**.

User Experience - or UX - was a concept we first brought up in [Chapter 4](#), where we learned about the User Interface. While UX and UI have some overlap, its goals are much deeper than simply *presenting information*. Creating a great UX will really force you to take a step back and view your game through the eyes of the user.

So let’s take those first steps towards creating a *phenomenal* User Experience.

Structure

In this chapter we will discuss the following topics:

- UX?
- Better UI for a Better UX
- Iconography
- Typography

- The First 10 Minutes
- World Space Prompts
- Options and Settings
- Perfecting our Camera
- A World of Levels
- A Polished Experience
- Mastering UX Design

Objectives

In this chapter, we're going to be learning about the importance of a well-crafted user experience. We'll cover several UI Design rules that will take complex systems and help break them into easy to understand elements.

We'll also cover some best practices concerning level design, and the importance of hooking the player in the first 10 minutes. Polishing and perfecting the flow of gameplay will keep the User Experience compelling, frustration-free, and - most importantly - memorable.

UX?

The phrase User Experience feels very broad - perhaps *too* broad. After all, everything the player sees, hears, reads, feels and does is part of their User Experience. In the world of interactive design, however, this phrase has a very specific meaning.

Put *simply*, building a great **User Experience** is the art of *simplifying a complex system*. And since Video Games are the product of MANY complex systems, creating a UX that guides, teaches, and explains - all in a subtle, streamlined way - is a very important concern for any Game Designer.

Note: A useful way consider the concept of UX is to step away from games and consider something more common – like a kitchen Stove. A stove has a UI: an instructional guide to install it, the knobs to light the burners, a handle to open the oven door, and a display for showing temperature and time. The UX, however, goes beyond displaying info and receiving input. How easy is it to remove the stove from the shipping container? How clear are the installation instructions? How heavy is the front door? How annoying is the timer alarm? Where UI deals with input and output, great

UX design has you identifying the sharp edges of any experience and smoothing them down.

Better UI for a Better UX

There are many ways UX and UI overlap, so much that they're terms that are often used interchangeably. But UI is simply one of the tools UX designers use to communicate complex mechanics to the player.

Let's break down some of the gameplay complexities the player will be managing...

- Making sure the player always knows where to go.
- Making sure the player knows how to do what they need to do next.
- Making sure only the *most important* information is displayed.
- Making sure there is always guidance in place for players who get stuck.
- Making sure the player doesn't have to 'click' too many times to navigate the UI.

To do all this, we'll be using colors, shapes, animation, sound, and vFX to communicate, all in a way that isn't obnoxious. Good UX will be subtle. Don't distract the player with UI when players just want to enjoy the world they're exploring.

Since one of the easiest mistakes to make in UX is presenting too much visual information, let's talk through some ways to fix this.

Iconography

One of the easiest ways to overwhelm a player is by building screens that have too much *text*. The inclination for a **Text Heavy** UI is understandable – words are the *clearest and most precise* way to communicate. However, it's not the *fastest*, and if there's one thing players need, it's *quick feedback*.

In our demo, we're using a text-only approach for displaying information to the player ([figure 14.1](#)), but this text heavy approach is not ideal.

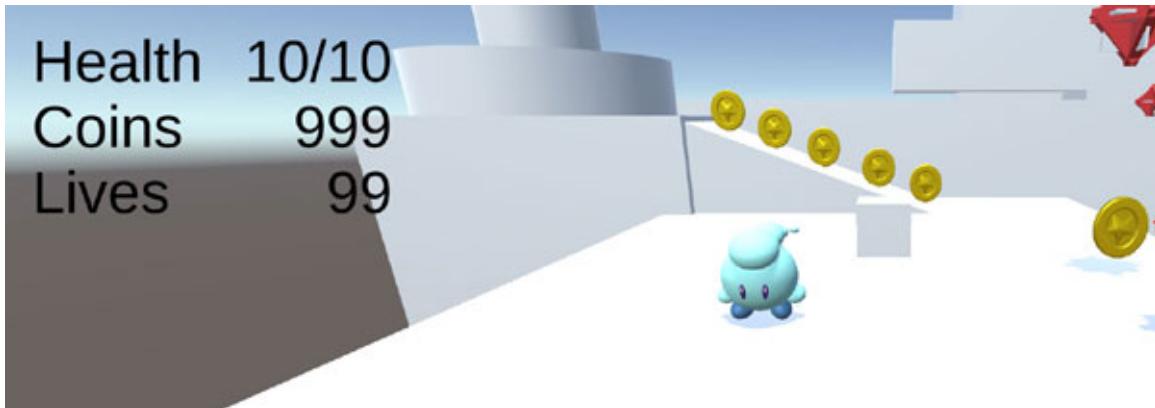


Figure 14.1: Our original HUD (Heads-Up Display). It's not necessarily overwhelming, but all that text certainly makes it look unfinished.

Since Video Games are a visual medium, the solution to ‘simplify complexity’ of a text-heavy interface is to simply replace that text with **Icons**.

Icons – and the art form of **Iconography** – take a complex idea, theme, or concept and distill it into a visual representation. In UI/UX terms, icons are small images that represent a concept from your game, like Health, Money, Equipment, or Resources.

Video games make heavy use of icons to convey information, and luckily for us, Unity gives us the tools needed to create these graphics directly from our game assets. Using Post Processing effects, Point Lights, and Unity’s high-quality Material system, you can make some professional looking art right in the Scene View window.

Follow the steps to make icons:

1. To make our icons, the first step is to create a new Scene.
2. Name it **IconSandbox** and hook up the main camera to use the Post Processor.
3. Next, line up all the assets you need to make Icons for. If you are making an Icon for a character, you can even use the opportunity to pose them in an interesting way.
4. Also, to make it easier to crop the icons later, disable the Scene Window’s Skybox option ([figure 14.2](#)). This will give us a nice, generic gray background, making the outline of the object easy to distinguish.

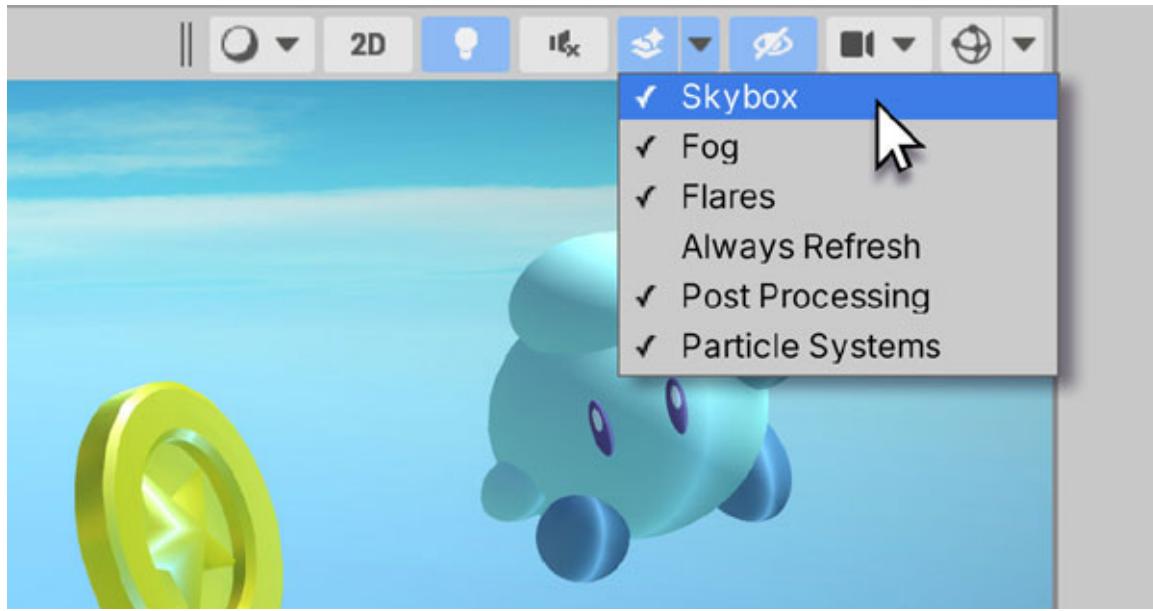


Figure 14.2: When using the Scene view to create icons, be sure to disable the Skybox. This will make it easier to cut out our posed hero (and other icons).

5. Another useful tool for converting assets into icons is the Point Light. Place one light next to each asset to create a **Rim Light** – a thin, bright edge on one side of the object ([figure 14.3](#)).

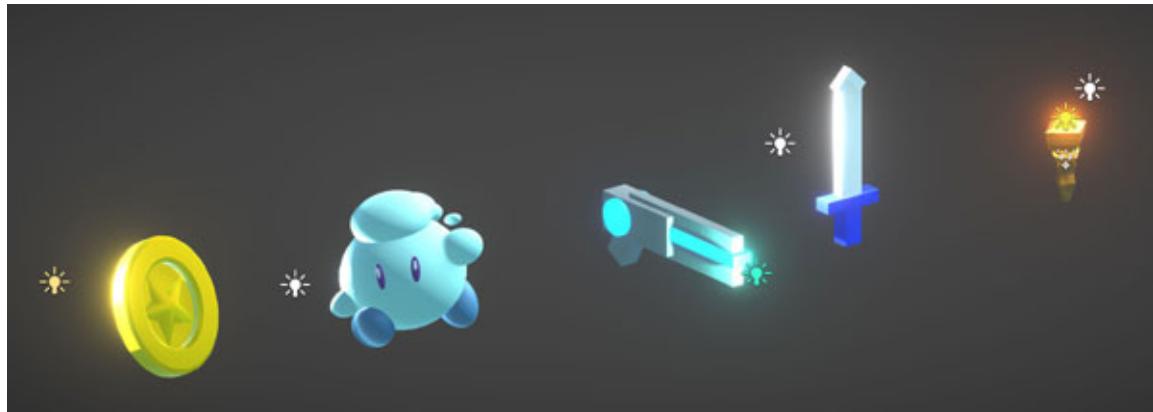


Figure 14.3: You can use Point Lights to give some extra visual polish to your icons.

6. Once you snap a high-resolution screenshot of the icons, you can simply use your Image Editing tool of choice to finish the job. Take each image, get rid of the background, and save as a transparent image (PNG, TGA, etc), as shown in [figure 14.4](#)

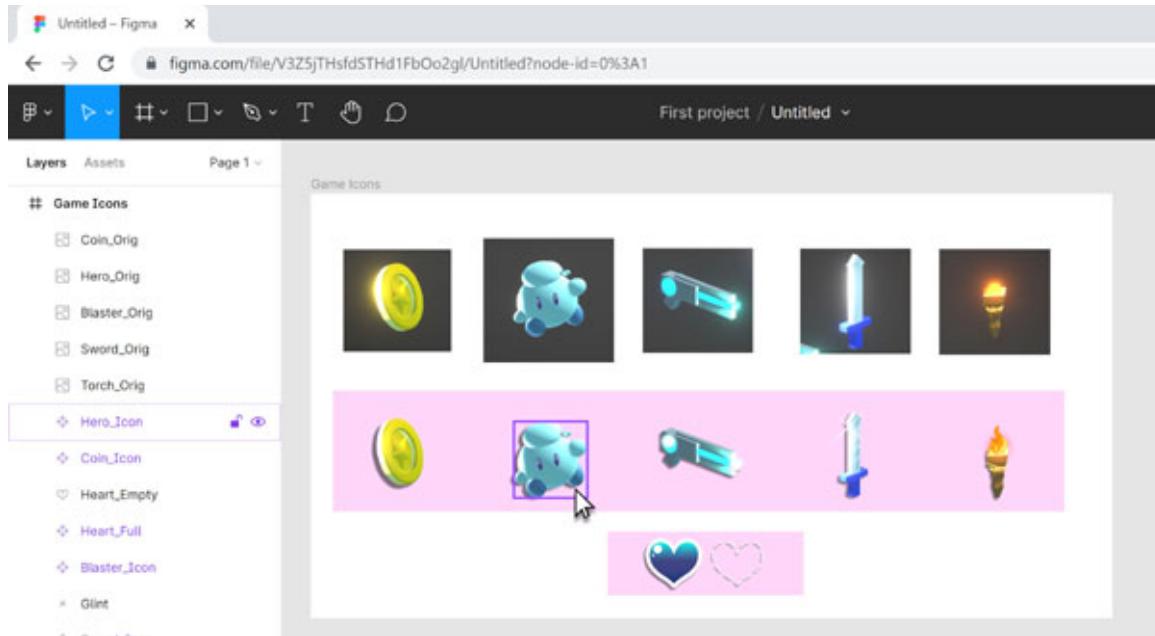


Figure 14.4: One industry leading Image Editing tool is the cloud-based Figma (www.Figma.com) but you can use any editing tool or software to crop the icon out of the background.

7. You'll need to keep these incoming images organized, so create a new folder called "Icons" (Assets > Resources > Icons) and place your new assets in there. Once added to the projects, they should look similar to those shown in [figure 14.5](#)

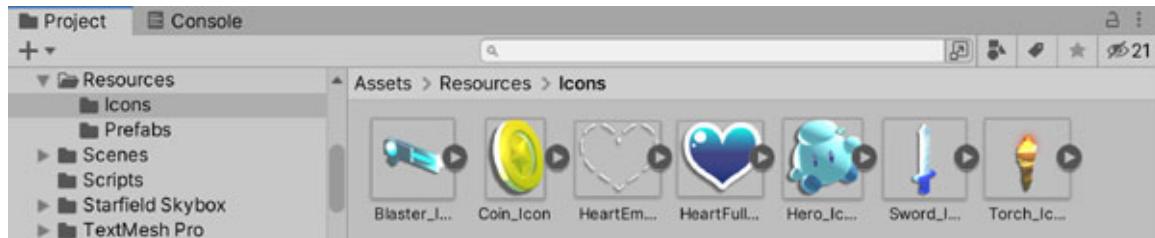


Figure 14.5: Your new Icons folder will quickly fill with game assets on larger projects.

8. All assets brought into Unity have to go through an Importing process, and these icons are no different.
9. If you select one in the Project Window, you can adjust the various Import options in the Inspector Window ([figure 14.6](#)). Since these icons will be used in the UI, you'll want to change the **Texture Type** to **Sprite (2D and UI)**.

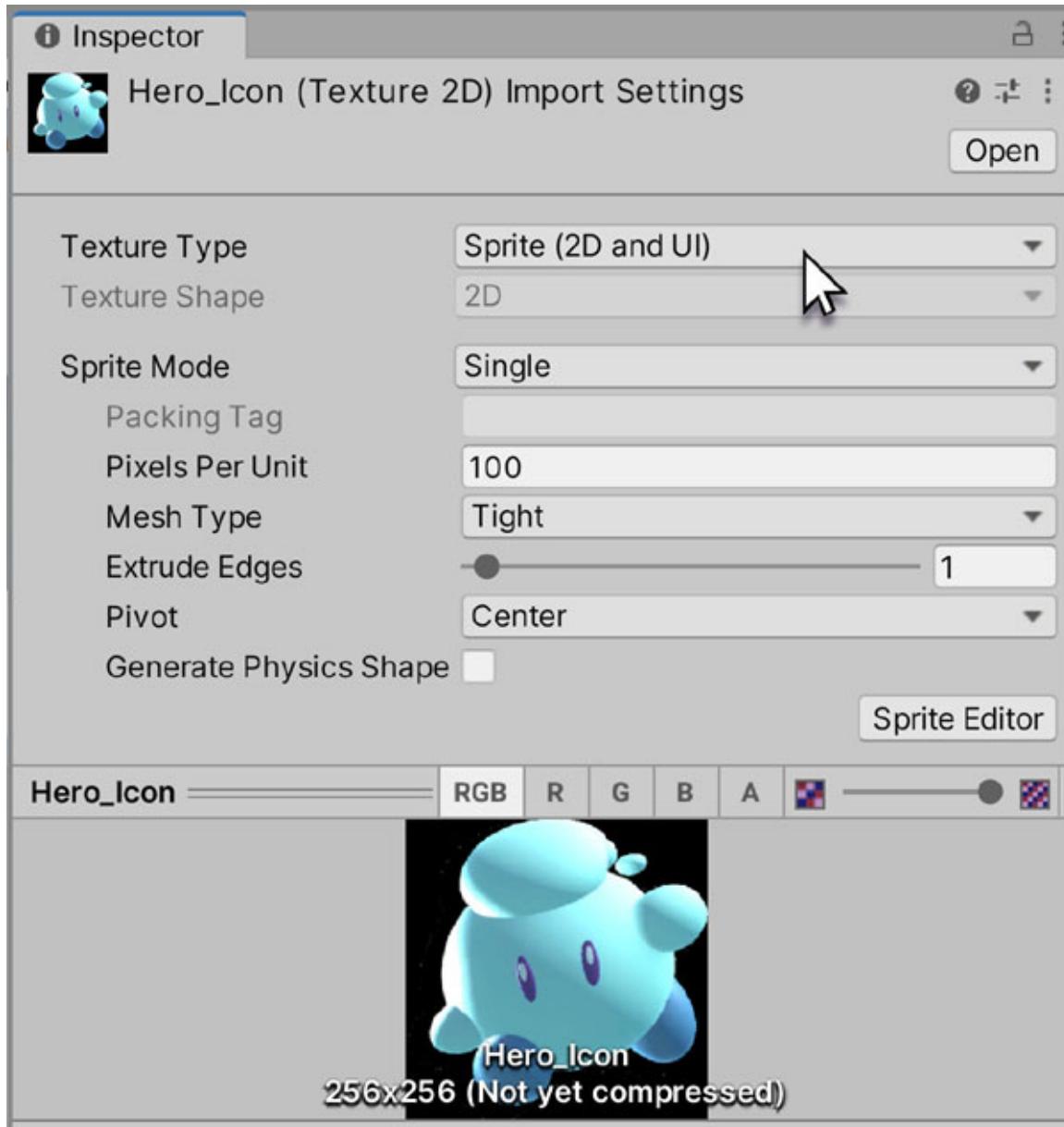


Figure 14.6: When importing assets for the UI, you'll want to make sure the Texture Type is set to “Sprite (2D and UI)”.

10. We can now place Image objects instead of text for our HUD elements ([figure 14.7](#)). By removing the words Health, Coins, and Lives – and replacing them with our new icons – the HUD already starts to feel more polished and professional.



Figure 14.7: Icons instead of text. Still 100% readable, without forcing players to **literally** read.

Unfortunately, even with the icons, our HUD still breaks one of the key UX design rules: *never use the default font*. Let's learn about Unity fonts and how to fix that issue.

Typography

While Icons are the preferred method of quick communication, there are times where you simply *must use text*. Character dialog, item descriptions, and exposition can't be simplified down to a single, tiny image.

In these cases, you're going to need to think about **Fonts**. And as much as **Typography** – *how your text appears to the reader* – is an aesthetic decision, there are several ways your choices here will directly impact the User Experience as below:

- **Picking Readable Fonts:** This sounds obvious, but many games will get hit in reviews because the developer picked a font that looked great on their monster-sized monitor, only to find it completely illegible on smaller screens. Make sure you're testing your UI across multiple screen sizes and resolutions to ensure text stays legible.
- **Multiple Languages:** Supporting foreign languages is an important aspect of considering the User Experience. Ensuring your fonts properly support foreign characters is a key part of this process.
- **Showing Personality:** If you have a dialog-heavy game, you can add personality to your text by assigning different fonts to different characters. This isn't necessarily a UX concern – just a fun polish point.

- **Walls of Text:** We talked about this above, but it bears repeating – screens overloaded with text will overwhelm the player. If you have a long description for something, place it in a secondary popup that requires a button press. It's fine to have areas with some descriptive text, but use them sparingly.
- **Never Use Defaults:** It may be tempting to select the default Unity font as ‘good enough’. This is a quick way to instantly cheapen your game, making it feel like something you ‘whipped together’, even if the underlying gameplay is great. Importing a font is easy, so don't overlook this simple (yet crucial) UX decision.

Let's use this opportunity to step through the Font importing process.

There are several websites to visit when looking for fonts, with [fonts.Google.com](https://fonts.google.com) and fonts.Adobe.com being two of the most reliable ([figure 14.8](#)).

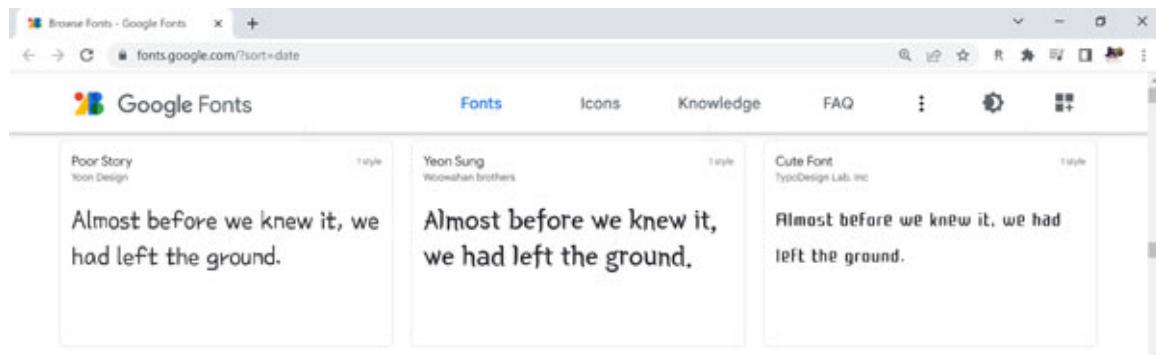


Figure 14.8: Visit [fonts.Google.com](https://fonts.google.com) for a large pool of high-quality fonts to pick from.

1. Once you've selected and downloaded a True Type Font (TTF) or Open Type Font (OTF), add that file to the project.
2. You can now open and convert the font using Unit's **Font Asset Creator**. Toward the top of the Unity editor, click **Window > TextMesPro > Font Asset Creator**. This will bring up a tool ([figure 14.9](#)) that lets you convert a TTF or OTF file into a font atlas, which lets you use the font in any project.

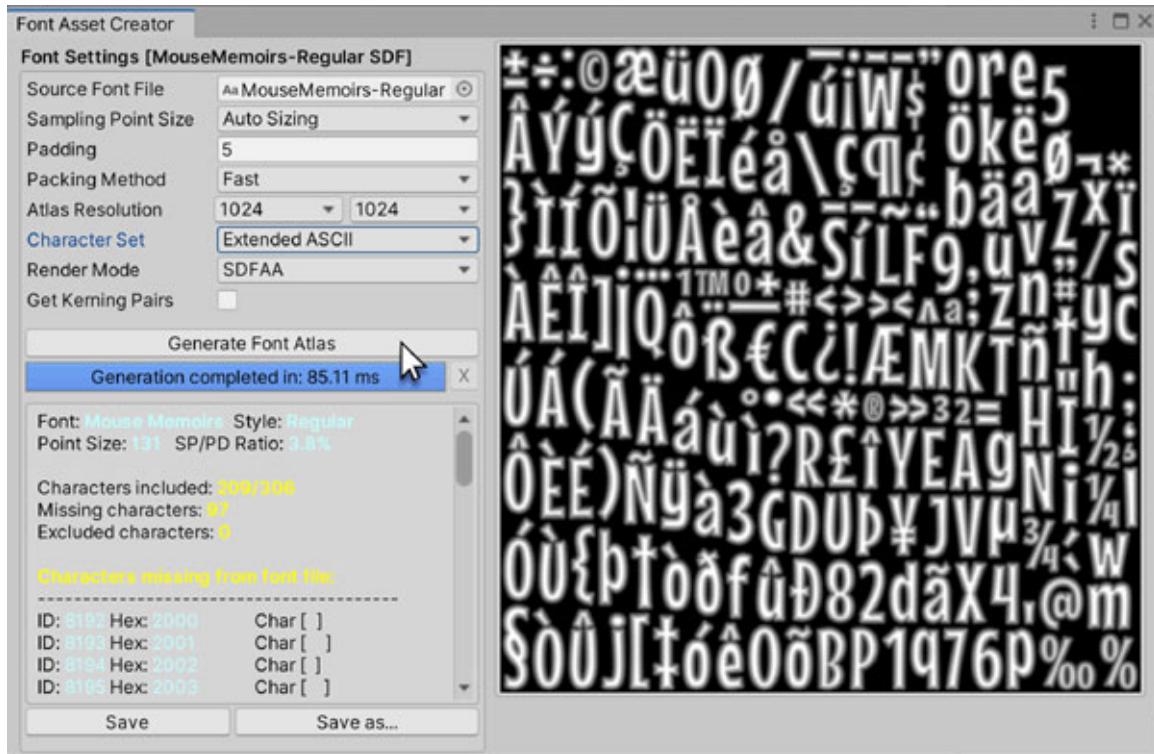


Figure 14.9: The Font Asset Creator takes a font and turns it into a Font Atlas, which is used by Unity to display text across a variety of different formats (UI, 3D Text, etc).

- Once you've converted the Font of your choice, the final step is to hook it up to our text object. Select CurHealthValue, CurCoinsValue, and CurLivesValue, then in the TextMeshPro component, set the **Font Asset** parameter to use your newly created font, as shown in [figure 14.10](#)

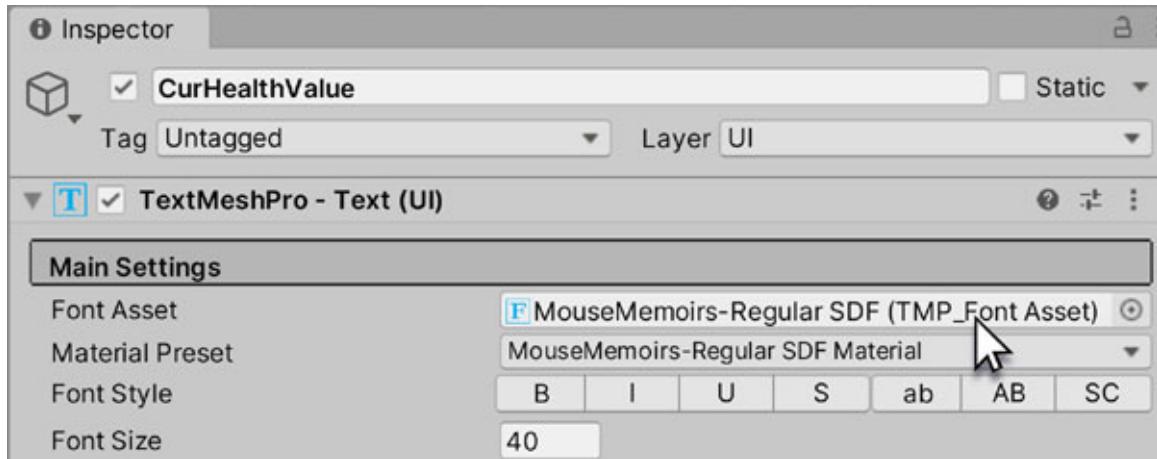


Figure 14.10: Select the HUD text objects and set the new Font Asset.

With the new font in place, and icons replacing the category text ([figure 14.11](#)), our HUD is now feeling more polished and professional.

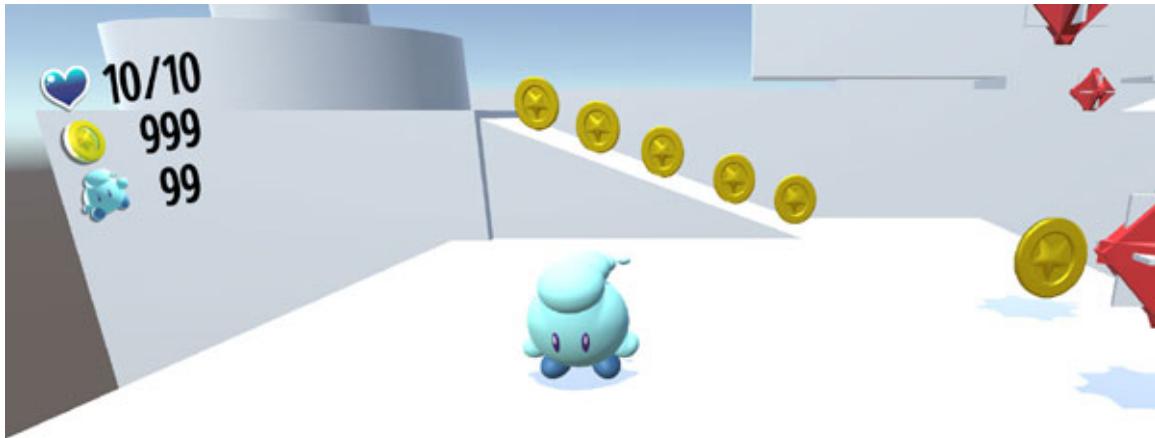


Figure 14.11: Our HUD now has a fancy new font!

There are some other things to consider when choosing fonts for a game you plan on publishing. The most important is **Font Licensing** - the legally-allowed uses of the font you'd like the ship with your title. While fonts available on Google Fonts (and many on Adobe Fonts) can be included in *any* commercial product, other fonts may require a license to be purchased. Double-check the legal usage before using, and shipping, a unique font.

World Space Prompts

Up until this point, our user interface objects have all been in Screen Space - displayed directly on top of our 3D Scene. There's another way a UI canvas can be rendered, called **World Space**, which will place the UI elements directly within the 3D scene itself.

Let's make use of World Space UI to build a 'Helpful Signpost' level object, a common **Onboarding** tool you'll see in many games. We'll make a small wooden sign that will catch players' attention, and when they get close enough to it, UI text will unhide, giving them helpful advice about the surrounding area.

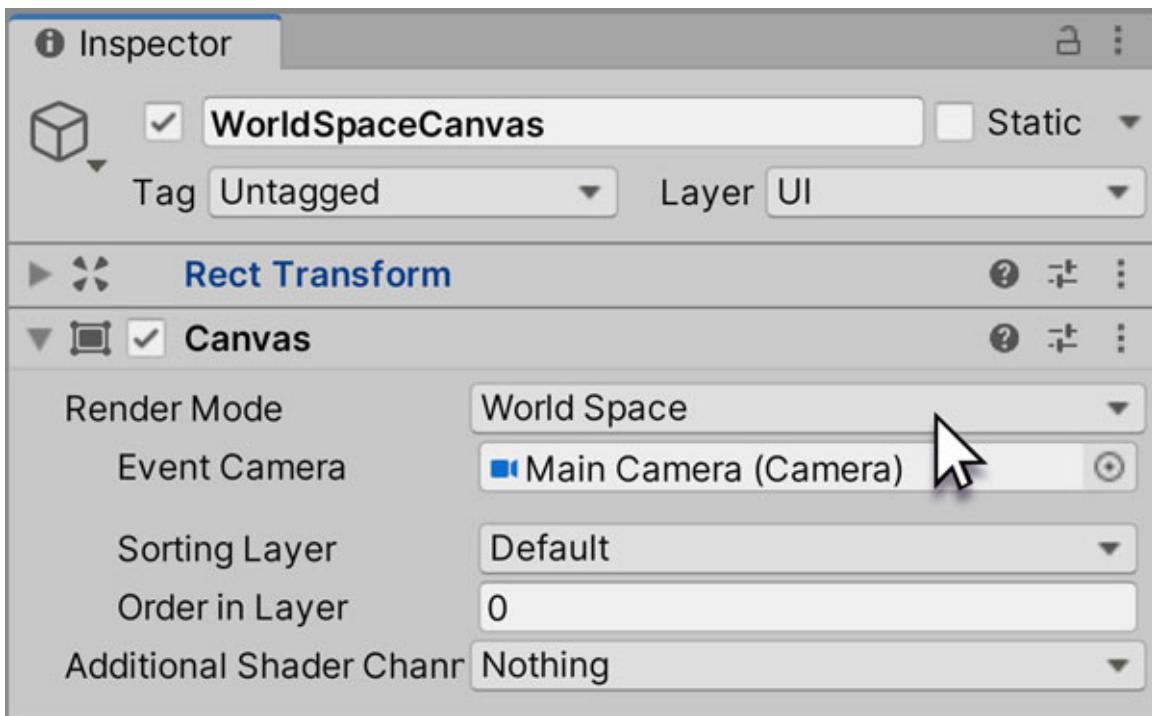


Figure 14.12: By setting the Render Mode of a canvas to ‘World Space’, those UI objects will be displayed in the 3D Scene (opposed to on-top of it).

1. Start with a new Canvas object, and name it **WorldSpaceCanvas**. Set the **Render Mode** to World Space, as in [figure 14.12](#). Because it’s a Canvas object, it can hold all the User Interface objects that we learned previously (images, text, buttons, etc). The World Space camera is especially useful in AR/VR development, or in FPS games that require interactive screens within the levels.
2. For our purposes, we only need two UI objects: an Image background and a Text foreground. First, place an Image object as the child of the canvas. Name it **PromptBG** and set the stretching style to fill the parent ([figure 14.13](#)). Under the Color parameter, set it to a Hot Pink color that’s very easy to see. Once we get sizing and placement looking correct, we’ll change it to a less *obnoxious* color.

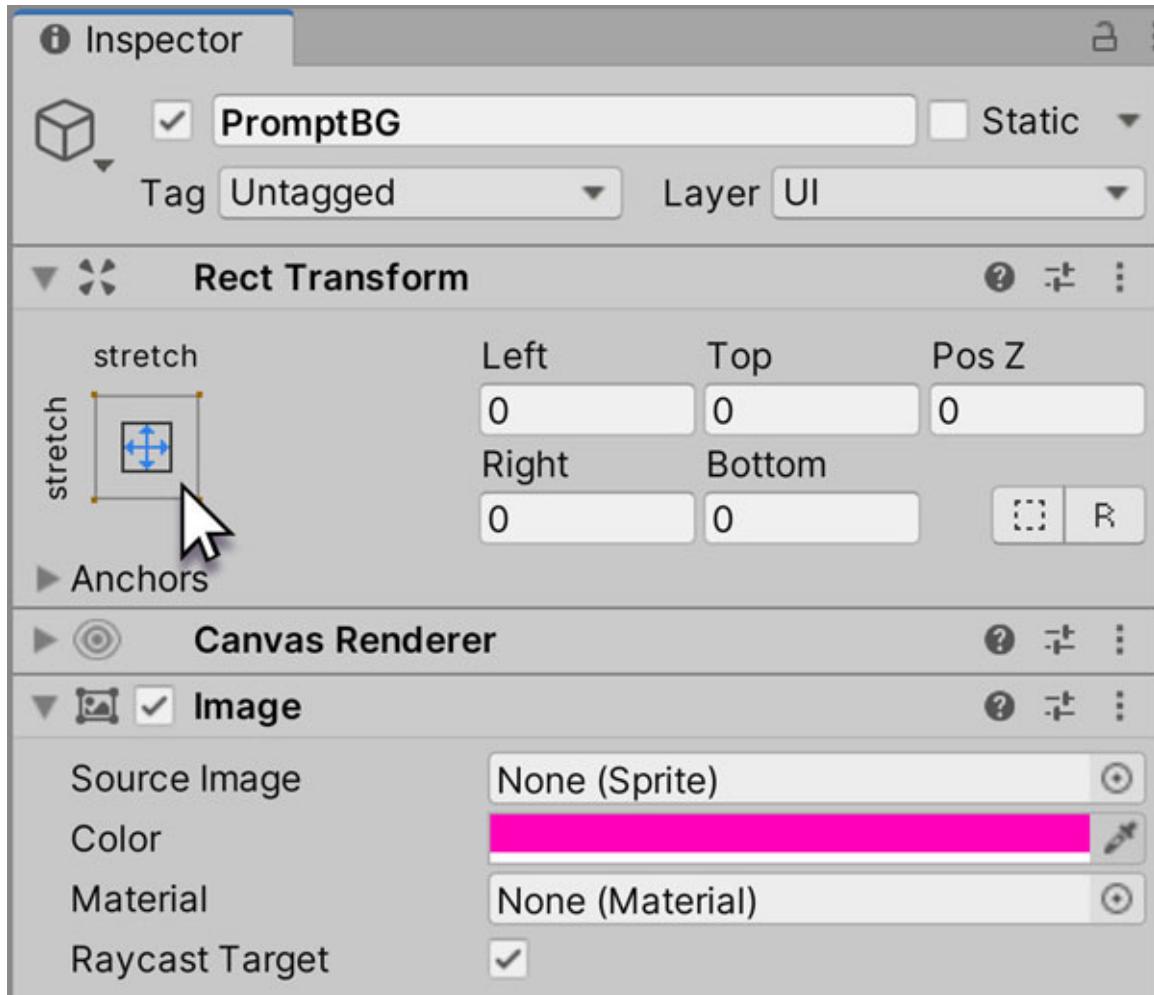


Figure 14.13: Our *PromptBG* object needs to fill to the size of it's *Canvas* parent and be an obnoxious Hot Pink color (for placement and debugging purposes).

This pink rectangle will now fill the size of the canvas, showing you the exact size, position, and orientation of our World Space Canvas

3. If you position the canvas near the hero and press **Play**, you can run around the scene to view the World Space UI in action ([figure 14.14](#)).



Figure 14.14: Our first steps to creating a World Space Prompt.

As you move the hero around, you can see the canvas staying at its position in 3D space. Getting close will cause it to grow in size, moving away from it will shrink it.

Let's now take a moment to create the visible part of our 'Helpful Signpost' - a small wooden sign that we can place in our levels.

4. Use 3D Cube Shapes, ProBuilder, or a combination of the two to make something you're happy with. If you're lacking inspiration, use [figure 14.15](#) as an example of a simple sign that's easy to make.



Figure 14.15: An example of a Wooden Signpost that you can make in Unity. The wood planks are 3D Cubes, the nail is a small Cylinder, and the paper is a ProBuilder Plane that's been 'crinkled', using smoothing groups to sell the effect.

5. Name this object **LevelObj_HelpfulSignpost** and drag the **WorldSpaceCanvas** object to make it a child of the signpost. Set the size and position of the Canvas to match the values shown in [table 14.1](#)

WorldSpaceCanvas						
Pos	X	0	Y	2	Z	0
Width	6					
Height	1					

Table 14.1: The position and size of the WorldSpaceCanvas object.

6. Now that we're happy with the size and placement of our WorldSpaceCanvas, let's change the **Color** of the PromptBG to be semi transparent black. Select the PromptBG object, then in the Image component, set the Color parameter to 0,0,0,180.
7. We'll also need some text to display our 'Helpful Prompt'. Add a TextMeshPro text object as a child of the PromptBG. Set it to stretch to fit within the parent (with a small border) and set the Text parameter to be two lines of text, as shown in [figure 14.16](#)



Figure 14.16: Our text box, which will display up to 2 lines of test to the player.

8. Another important component to add to LevelObj_HelpfulSignpost - a **Box Collider**. We need this to notify our scripts that the user is close enough to the sign to read it. Set it to match the basic shape of the Sign you've created.
9. Our final step is to add a script with text and collision code. Add this new script, named **WorldSpacePrompt**, to the LevelObj_HelpfulSignpost. Open the script in Visual Studio and fill the class with the following code...

```
using UnityEngine;
using TMPro;
public class WorldSpacePrompt : MonoBehaviour
{
```

```

[TextArea, SerializeField, Tooltip ("The text to
display.")]
private string _promptText;
[SerializeField, Tooltip("The TextMesh UI Object.")]
public TextMeshProUGUI _textObj;
[SerializeField, Tooltip("The Prompt Parent to toggle.")]
public GameObject _promptBG;
void Awake()
{
    _textObj.SetText( _promptText );
    _promptBG.SetActive(false);
}
private void OnCollisionEnter(Collision collision)
{
    // if the player collides with this object,
    // unhide up the prompt text
    if ( collision.gameObject.GetComponent<PlayerController>()
    )
        _promptBG.SetActive(true);
}
private void OnCollisionExit(Collision collision)
{
    // if contact with the player ends,
    // then hide up the prompt text
    if (collision.gameObject.GetComponent<PlayerController>())
        _promptBG.SetActive(false);
}

```

This component sets the text, hides it, and waits for the player to come into contact before unhiding it.

- Once the script is filled out, make sure the component parameters are set properly. Drag the PromptText and PromptBG objects to their appropriate fields in the Inspector window. Once complete, the signpost, and associated components, should look similar to those in [figure 14.17](#)

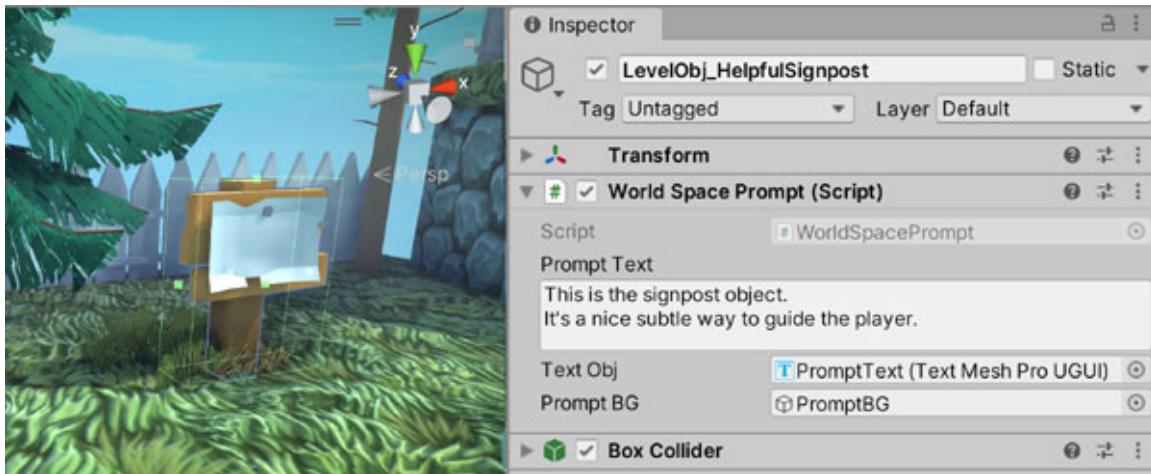


Figure 14.17: Our finalized Signpost, with Box Collider and our newly created Prompt script.

It's time to test our new Onboarding object. When the player sees it from a distance, the text will be hidden. When the player comes up to read it, it will unhide, then hide again when the player walks away.

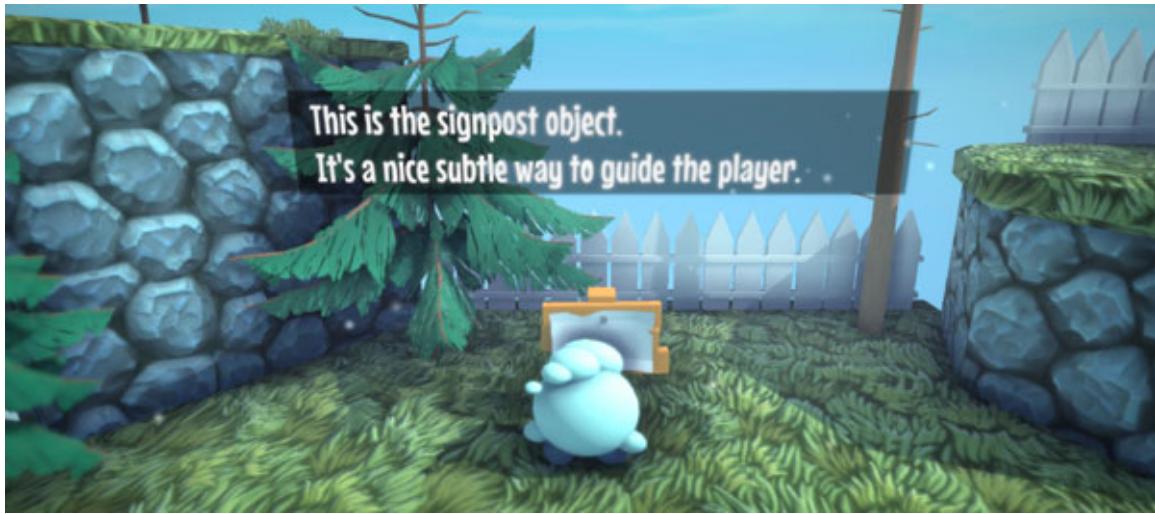


Figure 14.18: Our Signpost gives the player control over what information they want to read.

The signpost is a great reminder of the importance of *subtlety* in our UX. Instead of forcing the player to read every little onboarding prompt, they now have agency over what information they want to see. Advanced players will probably skip most of these prompts, having already mastered the basics in other games. On the other hand, players new to the genre will seek out signposts and be rewarded with useful information and gameplay tips.

By drip-feeding hints to the player, and giving them a choice about what prompts to read and which to ignore, you've crafted a great UX that will give the appropriate amount of information to all skill-levels.

The First 10 Minutes

Now that we've covered the ways that UI can improve the User Experience, let's talk about creating a game that **Hooks the Player**. Specifically, let's focus on those *first 10 precious minutes of gameplay*.

10 minutes to pull the player into your game and not let go.

10 minutes to prove that your game is worthy of their time and money. *10 minutes* to captivate your players with the following systems and concepts...

- **Cutscenes:** If your game has a strong backstory, or even if you just have a simple lead-up to the main action, it's always good to have a compelling Cutscene that performs some basic exposition. Sometimes a simple emotional hook (someone/something needs your help) is all you need to capture the player's attention.
- **Title Screen:** The Title Screen will be the first bit of interaction your player has with your game. Make it impressive!
- **Introductory Stage:** Many level-based games will have an **Introductory Stage** to get the player used to the controls, understand the dangers, and give tips as needed. Just keep text to a minimum. It's a video game - keep the learning fun!
- **Second-to-second Gameplay:** It's important that your movement - the feel of your hero as the run, jump, attack, etc - is as close to perfect as possible.
- **The Broader Threat:** By the end of this 10 minute window, the player should have accomplished a smaller goal, with a larger threat looming on the horizon. They should feel like the game needs them, and to quit now would be abdication of heroic duty.

Our Introductory Stage

Of the items listed above in the First 10 minutes section, creating a great **Introductory Stage** is one of the most important. This is where the player will first have an opportunity to embody the hero and explore your game world. It also sets expectations for the rest of the game, so if you treat your player poorly here, they'll assume similar sun-par design choices across the rest of the adventure.

Note: Remember that poor game design decisions in those initial 10 minutes communicates poor design across the entire game. Do your best to nail this first impression.

With that in mind, let's start building our Introductory Stage.

1. Make a copy of the **SampleScene** and rename it **IntroductoryLevel**. Remove all the level objects, leaving only the UI, hero, GameSessionManager, Lights, and SoundManager.
2. To get the look we want, enable the Post Processor and set the Skybox, like we did in [Chapter 11](#) “A Graphical Upgrade”.

Now, let's create a level that's easy to navigate. At this point in the project, your Prefabs folder should have several objects that we can use to create our level.

The example shown in [Figure 14.19](#) is on the short side, but it can be broken up into 4 areas...

1. **Starting Area:** This is where the player starts. You'll notice there's really nothing to be done here besides move to the left, and that's perfect for the initial introduction. Give the player an area where all they need to do is move the hero around.
2. **Jumping Challenge:** There are a series of three columns that the player needs to jump across. Failure here will not result in death - and safety will be a reoccurring theme in this level.
3. **Climbing Challenge:** There is a small section of platforms that the player needs to climb. By making the end goal higher than the starting point, it gives the player a metaphorical ‘Mountain Top’ to reach.
4. **Physics Challenge:** The final challenge is a sea-saw object that we made back in the Physics Chapter. Again, a fall here will simply force a retreat of the climbing challenge - a less severe punishment than losing a life.

[Figure 14.19](#): The simplicity of this level is completely intentional. Getting used to the basics - and having fun doing it - is the primary goal here.

By zooming in, we can get a clearer picture of what the player is encountering in some of these areas.

5. In [figure 14.20](#), for instance, you'll see the three jumping columns that the player needs to jump across to continue. By placing the coins on the

tops of these platforms, we're signaling the correct path to the player. By placing some ground beneath the platforms, we keep the challenge safe, so falling in won't frustrate the new player.

Figure 14.20: These first challenges should be as frustration-free as possible. Here, death should never be the punishment for failing a trial.

6. This safe area under the platforms is also a great spot to help any players having issues. Let's put a Helpful Signpost down here with some tips (and some inspirational energy), placing this text into the **Prompt Text** parameter...

Making those jumps may look hard, but it's as easy as pressing <color="green">Spacebar</color>!

7. Now, when a player falls in that pit, they will have the option to read the sign and learn about using the Spacebar to jump, like in [figure 14.21](#)

Figure 14.21: The Helpful Signpost lives up to its name, helping players who may not know the nuances of jumping in our demo.

Explaining 'Use Spacebar to Jump' would be overkill if presented to every new player. Most gamers will have some idea what to do here, so it's better to give the player a safe space to attempt the jumps without help. Only when they're in the pit, and possibly in need of that help, should we provide it.

Give the player the room to learn on their own, and only step in with assistance when necessary.

There's something else going on in this prompt that you should find useful. When using text fields in Unity, you can make use of tags to perform various alterations to how it's displayed. We made use of the `<color>` tag, but there are several other useful tags to use when formatting your text, as listed in [table 14.2](#)

<code><color></code>	Colorize the text between two tags.
<code></code>	Displays the text as bold .
<code><i></code>	Displays the text in <i>italic</i> .
<code><u></code>	<u>Underlines</u> a block of text.
<code></code>	Set the font to use for a block of text.

<size>	Adjust the size of your text.
<sprite>	Place any sprite image directly in a block of text!

Table 14.2: Several useful tags when formatting text in Unity.

Moving on from formatting text, let's talk about the subtle ways we can keep the player safe. You'll notice several fence objects throughout the level, as seen in [figure 14.22](#)

Figure 14.22: These fences aren't just for looks. They perform the critical task of keeping the player from accidentally running off the edge of the level. Do everything you can to prevent easy deaths in your Introductory Stage.

By placing these fence objects around the level, you're preventing accidental deaths (which would lead to justified frustration). You can also prevent accidental deaths by placing large Box Collider objects up against the edge of platforms.

If the purpose of this level is to learn the basics in a fun way, our design task is to remove all the things that would prevent fun. Eliminating the possibility of death is often the right call in an Introductory Stage.

Back to the example, once the player gets past the Physics challenge (the Seasaw prefab), they will have reached the metaphorical 'Mountain Top'. The expectation is that something important will be here, but as of now, we reward the player with *absolutely nothing* ([figure 14.23](#)).

With the player finding an empty peak, this is the perfect time to move into our next important UX topic - communicating your **Victory Conditions**.

Figure 14.23: The player has reached the peak, but will find 'nothing' an unsatisfactory and underwhelming reward for the journey.

Achieving Victory

It's important for the player to understand exactly what is expected out of them, both concerning short term and long term goals.

You can imagine the introductory cutscene, or opening exposition, would set up the long term **Victory Conditions** of the game. Defeating a spreading darkness. Rescuing a captured princess. Healing a ruined world. These are all

large, broad, overarching goals that will drive the player initially, but will need to be reinforced with *bite-sized* victories along the way.

Many times these smaller goals will be presented at the start of a level, or perhaps as quests (in a more open-ended adventure), but most games will use some combination of the ideas listed here...

- **Reach A Location:** In platformers especially, moving from left to right is the main driving force. At the end of a level, players are treated to a flagpole, sign, or some marker to let them know this stage has been completed.
- **Item Collection:** By tasking players with collecting X amount of items, you encourage exploration and level navigation.
- **Defeat Enemies:** By tasking players with defeating X number of enemies, you encourage skilled combat and clever tactics.
- **Escort Missions:** While mostly frowned upon, the concept of escorting an NPC from one point to another – all while keeping everyone safe – is an immediately compelling goal. Just make sure you spend enough time working on the NPC AI, otherwise these can lead to considerable frustration.
- **Defeat A Boss:** A boss battle is a great way to spike difficulty and make a clever battle to test the gameplay mechanics introduced up until that point. Just make sure you don't slap tons of health on a boring fight – players hate a **Bullet Sponge** Boss.

With most of these Victory Conditions, conveying the current goal is normally a UI design challenge. A health bar during boss battles. A rolling counter of remaining items to collect. A mini-map showing the enemies to hunt down and defeat.

For our game, however, our problem is less about *conveying* the goal, since players will naturally follow the trail of coins to the end. Our problem is that, once you reach the end, there's literally *nothing there*.

You need to *have* a Victory Condition before you need to worry about *communicating* it.

This is another situation where you can use your creativity to make something for the player to find at the end of this level. The [table 14.3](#) will give you a few inspirational ideas...

Golden Star	Flagpole	Sign	Magical Crystal	Magical Portal
-------------	----------	------	-----------------	----------------

Cat or Dog	Hot Dog	Trophy	Hamburger	Princess
Prince	Treasure Chest	Colorful Gem	Something Lost	Someone Lost

Table 14.3: Ideas for an ‘End of Level’ Item for the player to find, triggering the end of this stage.

While many platformers have a Flagpole or Sparkly Trinket as their goal, we’re going to attempt to do something different, placing a ‘Lost Character’ at the end of the level. We’ll make this character something players are used to, however - a star.

By giving that star some personality, as shown in [figure 14.24](#), we can have some fun balancing ‘something unexpected’ and ‘something unique’.

Figure 14.24: Let’s end our level with a Star - something players WOULD expect - but let’s give that star a quirky personality - something players may NOT expect.

This concept of taking something known, then putting a unique twist on it, is known as **Breaking Expectations**. Some of the most popular games have hooked players by breaking expectations. *Undertale* surprised and captivated players as an RPG where you can win battles through mercy and kindness. The amazing *Mario + Rabbids: Kingdom Battle* put Nintendo’s beloved characters into an X-Com style Tactical Strategy adventure - with Rabbids! Surprising players with new and unique twists on existing concepts is a great way to hook players.

And while our Starperson doesn’t break expectations to the level of those classic games, it is a fun departure from the traditional ‘Golden Star at the end of a Level’. We’ll also do some **Environmental Storytelling** - using objects as props to tell a story implicitly. By building a Tent and Campfire around our lost Star ([figure 14.25](#)), players will infer he probably went for a hike in the area and got lost. Poor guy.

Figure 14.25: This Starperson may look goofy, but their desperate cries for help should enforce a sense of heroism in the player. Saving people – whether they’re lost children, kidnapped royalty, or strange looking Starfolk – is what being a hero is all about.

- With the ‘End of Level’ object modeled, add a Sphere Collider to the object. You’ll now want to add a new script called **SwitchSceneOnCollision**. Open the script in Visual Studio and fill the class with the following code...

```

using UnityEngine;
using UnityEngine.SceneManagement;
public class SwitchSceneOnCollision : MonoBehaviour
{
    [SerializeField, Tooltip("Name of scene to load.")]
    private string _sceneToLoad;
    [SerializeField, Tooltip("Seconds between collision and
load.")]
    private float _transitionTime = 1f;
    private bool _hasCollided = false;
    void Update()
    {
        if (_hasCollided)
        {
            _transitionTime -= Time.deltaTime;
            if (_transitionTime <= 0f)
            {
                // time to load the scene
                // remember: this scene needs to be added
                // to the Build Settings 'Scenes' list !
                SceneManager.LoadScene(_sceneToLoad,
LoadSceneMode.Single);
            }
        }
    }
    void OnTriggerEnter(Collider collider)
    {
        if (collider.gameObject.GetComponent<PlayerController>() )
        {
            // the player has collided with this obj
            _hasCollided = true;
        }
    }
}

```

This script has one purpose - switch scenes after the player collides with this object.

2. Once the script is attached, fill out the **Scene to Load** parameter to be **WorldMap** - as shown in [figure 14.26](#). By ending the Introductory Level

on a sprawling map of your game world, you're showing the player the epic scope of the adventure that lays ahead - an exciting prospect, if you've hooked them properly.

Figure 14.26: The ‘End of Level’ object, with all the necessary components.

3. Press **Play**, and if you did everything right, running into the frightened Starperson will take you directly to the Main Map. You may run into a new Scene Switching issue, however, where the incoming scene looks - *odd*. Specifically, if the new scene comes in looking *very dark*, you may need to perform an extra step.
4. Load the World Map scene, go to the Window > Rendering > Lighting option, and when the Lighting window appears, press the **Generate Lighting** button in the lower right corner, as shown in [figure 14.27](#)

Figure 14.27: If your scenes are coming in with bad lighting, you may need to use the Generate Lighting option in the Lighting window.

5. With the World Map lighting generated, return to the Introductory Level scene and press **Play**. Save the Starperson again, and this time the World Map should load with proper lighting ([figure 14.28](#)). Add some level markers to identify levels that can be traveled to next, and the player will realize the Expansive Adventure that awaits them past the 10-minute mark!

Figure 14.28: The Main Map is another great example of taking a complex system (dozens of levels of various difficulty) and simplifying it for the player - the core objective of UX design.

A Smooth Transition

Another key part of the UX process is identifying the ‘Rough Edges’ of your game - parts that may feel *unfinished* - and spending time *finishing them*. This process of improving an existing system, element, or asset is known as **Polish**, and it’s a great way to make a game that looks and feels *professional*.

We’ve just added an important system to our game, the **Switch Scene On Collision** component. And while it functionally works, it’s current

implementation *feels* rough. The instant transition from the Introductory Stage to our Main Map is far too abrupt.

Let's wrap up our UX work by polishing this rough spot of the experience. By animating a **Scene Transition**, we can smooth out this abrupt swapping of our scenes, using UI elements to fade to black, then fade-in when the new scene is loaded.

1. Start by creating another canvas object in the IntroductoryLevel scene. Name it **UITransitionOverlay** and set the Canvas **Sort Order** to **100**. This will ensure it appears over all the other UI elements.
2. Add two Image objects as children of the new canvas. Set them both to be pure Black (0,0,0,255) and name them **Top** and **Bottom**. We'll be animating them to cover the outgoing scene, then moving them away to reveal the incoming scene.

Use [table 14.4](#) to set the position, rotation, and anchor values of these two elements.

Table 14.4: The transform values for the Top and Bottom transition objects.

3. Our next step is to add a new **SceneTransition** script to the **UITransitionOverlay** object. Open this new script in Visual Studio, and copy the following code into the SceneTransition class...

```
using UnityEngine;
using UnityEngine.SceneManagement;
public class SceneTransition : MonoBehaviour
{
    // the top and bottom objects to animate
    public Transform _topObj;
    public Transform _botObj;
    Vector2 _endTop, _endBot;
    // wait a half second before unhiding
    public float _curTime = -0.5f;
    // by default, transitions take 1 second
    float _endTime = 1f;
    // data for loading a scene post-transition
    public bool _loadScene = false;
    public string _sceneToLoad = "";
    // the singleton instance of this object
```

```
static public SceneTransition Instance;
private void Awake()
{
    // make sure transition objects are visible
    _topObj.gameObject.SetActive(true);
    _botObj.gameObject.SetActive(true);
    // store the single instance of the transition object in
    // this scene
    Instance = this;
    // calculate final transition positions
    float offset = Screen.height * 0.75f;
    _endTop = new Vector2(0, offset);
    _endBot = new Vector2(0, -offset);
}
void FixedUpdate()
{
    _curTime += Time.deltaTime;
    if ( _curTime < 0 )
    {
        // buffer before transition starts
        return;
    }
    else if (_curTime > _endTime)
    {
        // reached the end of the transition
        // is it time to load the next scene?
        if (_loadScene)
        {
            SceneManager.LoadScene(_sceneToLoad);
        }
        return;
    }
    // note: linear lerping is boring, so apply
    // an ease-out animation with Mathf.Pow()
    float t = Mathf.Pow(_curTime / _endTime, 4);
    if (_loadScene)
    {
        // move objects into position then load a new scene
```

```

        _topObj.localPosition = Vector2.Lerp(_endTop,
Vector2.zero, t);
        _botObj.localPosition = Vector2.Lerp(_endBot,
Vector2.zero, t);
    }
else
{
    // transition objects away to reveal the scene
    _topObj.localPosition = Vector2.Lerp(Vector2.zero,
    _endTop, t);
    _botObj.localPosition = Vector2.Lerp(Vector2.zero,
    _endBot, t);
}
}

static public void TransitionToScene( string sceneToLoad )
{
    Instance._curTime = 0;
    Instance._sceneToLoad = sceneToLoad;
    Instance._loadScene = true;
}
}

```

This code does two things. First, when the object ‘wakes up’, it will make sure the top and bottom objects are visible, then animate them outwards, revealing the 3D scene underneath. Second, the static function **TransitionToScene()** can be called from any other script to smoothly transition to another level.

- With the script made, lets hook up the Top and Bottom object parameters, as shown in [figure 14.29](#)

Figure 14.29: The components that make up our Transition Overlay object.

- We now need to tell the Transition Object when it’s time to fade out to the Main Map. Go into the SwitchSceneOnCollision class, find the SceneManager.LoadScene() line of code, and replace it with these two lines...

```

SceneTransition.TransitionToScene( _sceneToLoad );
enabled = false;

```

This will fire off the Scene transition when the ‘end of level’ object is hit. It will also *disable* the ‘end of level’ object, so it doesn’t attempt to re-fire this logic every time the player touches it.

Our transition objects are all in and ready to test, but we’re not presented with a new challenge - our editor has a completely black screen.

6. Because the UITransitionOverlay objects start off in the ‘hidden scene’ state, we’re now presented with a game window where we can’t see what we’re editing. The simple fix is to disable the Top and Bottom objects and set the transition overlay to ignore clicks in the editor. To disable the selection of objects, go to the Hierarchy Window and toggle the pointer hand icon to the left of these objects ([figure 14.30](#)). This tells Unity to ignore the selection of our UI elements when clicked on.

Figure 14.30: Toggle the Hand Icons in the Hierarchy Window to disable the selection of objects in the Unity Editor.

7. With the Transition Object made, it’s time to turn the UITransitionOverlay into a prefab, then place it in the World Map scene. Once both contain the transition object, return to the Introductory Level and press Play. Reaching the end of level object - whether it’s a Starperson, flagpole, or something else - will trigger the fade-out-fade-in effect ([figure 14.31](#)). This smooth transition from one scene to another, while relatively simple to hook up, should make the end of our levels feel considerably more professional.

A small amount of polish can have a big impact!

Figure 14.31: Our finalized transition in action, smoothly fading from our Introductory Level into the World Map.

Conclusion

Building a high quality, frustration-free User Experience requires expertise with almost every gameplay system. UI is an obvious consideration, but sound, VFX, and level design will all prove to be vital tools when perfecting presentation and simplifying the complexities of your game.

Without great communication, it will be difficult to retain players. First they will feel lost in your game, and given enough frustration, you’ll soon lose them

to the multitude of other games available. Polishing and perfecting your game - especially those first 10 minutes, will be critical to hooking and retaining players.

In the next chapter, we'll talk about another way to simplify your design: 2D gameplay. For anyone planning on creating a 2D game, it's necessary to master all the 2D systems and tools Unity provides.

Let's jump into the ultimate gameplay showdown of 2D vs. 3D!

Questions

1. With games that you enjoy, how long does it take to get from the title screen to the main game? How many ‘clicks’ until you find yourself getting impatient?
2. Pay attention to the HUDs other games. Besides improving UX with Icons and Fonts, what else can we do to display only the information needed at a given moment? Are their games that hide and unhide elements based on what’s happening in-game?
3. Besides Flagpoles and Distraught Characters, what other ideas do you have for designating the End of a Level?
4. How could you use the animation system to make ‘Saving the Starperson’ feel better? Can you imagine a way to animate a happy “Thank You!” expression to toggle to when the player collides with them?
5. One useful way to improve the battle system UX is by putting in an ‘RPG Style’ damage values, where every time an enemy takes damage, those values are displayed to the user. We now know how to spawn prefabs and create World Space UI elements, so how would you use these systems to implement a ‘Damage Values’ feature?
6. Polishing your game is an important iterative step. Playing through the current demo, what area do you feel need polishing?

Key Terms

- **User Experience:** Using UI, sound, and vFX to present complex game systems in a way that's simple to understand. Good UX ensures the player always knows What to do, How to do it, and Why it needs to be done.

- **Text Heavy UI:** With complex systems, it's easy to accidentally overwhelm the player with screens that have too much text on them.
- **Icons:** Instead of using text for everything, you should make use of small, easy to read images - known as Icons - whenever possible. Images are quicker to 'read' than text, and simply more appealing to the user.
- **Typography:** The way text is presented to your user. Use various, easy-to-read fonts to display information to the reader.
- **Font Asset Creator:** The tool used to convert fonts from their original format (TTF and OTF) into a Font Atlas, which can be used in Unity.
- **World Space UI:** UI elements that are displayed within the 3D game world - not simply overtop.
- **Formatting Tags:** Unity will accept and parse various tags to format text being displayed. Use `<color>`, ``, `<i>`, `<sprite>`, and several other tags to highlight key phrases being presented to the player.
- **The First 10 Minutes:** A key metric in captivating and retaining the players attention. On average, you have 10 minutes to impress the user before they either keep playing or jump to another game. Polishing this experience is vital to your game's success.
- **Breaking Expectations:** Taking a known concept and putting a unique twist on it. This is a great way to hook your player: allowing them to feel comfortable with something they know, then surprising them with a clever variation on that idea.
- **Environmental Storytelling:** Using props, objects, and vFX in Level Design to subtly tell a story. An invested, detail-oriented player will enjoy seeing these visual clues, piecing together the backstory being told.
- **Polish:** Iterating and improving an existing system or asset, removing the aspects that feel unfinished and bringing them to a more 'finished' state.

CHAPTER 15

2D vs. 3D

Unity is one of the most powerful 3D game engines out there. As you've learned in previous chapters, Unity's tools allow you to create amazing 3D experiences - balancing power, performance, and ease-of-use in an unprecedented way. And every year, the team at Unity is making massive improvements to both the editor and rendering systems, bringing their real-time 3D capabilities ever closer to *movie quality* imagery.

As a real time 3D powerhouse, Unity has grown into an *absolute beast*.

There are some game designers, however, that are less interested in *3D muscle* and more interested in *2D simplicity*. Perhaps you're one of these designers, with plans to develop a sidescroller, puzzle game, or retro adventure. Or perhaps your dream game would use a unique, hand-drawn animation style, similar to the blockbuster *Cuphead* ([figure 15.1](#)).

If you're that kind of game designer, looking to make your own 2D masterpiece, then Unity has you covered. Let's dive right into the tools and systems Unity provides to make *your* 2D game a reality!

Figure 15.1: Unity's 2D tools were used in *Cuphead* to mimic an old-school animation style

Structure

In this chapter we will discuss the following topics:

- 2D vs 3D
- Sprites
- Tilemaps
- 2D Movement
- 2D Animation
- Cinemachine Camera

- The Power of Parallax

Objectives

In this chapter, we'll be covering the large assortment of tools, systems and samples built directly into Unity for the creation of 2D projects. We'll be starting in a completely fresh project, then work our way through the use of Sprites and building levels using Tilemaps. We'll then download a 2D Demo Project to learn about the more complex tools you'll have access to.

By the end, you'll have all the knowledge required to start creating your own amazing 2D game.

2D vs. 3D

When Video Games were first introduced, game designers had a very limited set of options when it came to the *graphical fidelity* of their titles. With a very low resolution and only two color values - Black and White - the earliest games had a very simple aesthetic style. But it didn't matter, because the games being made were *fun*, regardless of fancy graphics and colorful images.

Note: We should make the connection, again, to this fundamental rule of design: your game should be fun, compelling, and have strong player retention before awesome graphics are even introduced. The first Video Games were forced to only use white boxes, and these games spawned an industry. Keep this in mind when creating your own project - start with the basics, make something fun, then build from there. If it was possible then, then it's possible now.

As hardware became more powerful, you saw designers and developers use that horsepower to improve graphics. More colors, higher resolution, and eventually 3D graphics. The 3D images that can now be displayed on phone displays is a testament to how far the industry has come.

Even with the power to make 3D images, sometimes it makes more sense for a game to use a traditional 2D setup. Some of the tradeoffs associated with 2D vs 3D are listed in [table 15.1](#)

Table 15.1: The Pros and Cons when creating a 2D or 3D game.

Our 2D Sandbox

As with the other game development concepts we've covered, it's important to set up an empty 'sandbox' environment as we get our hands dirty with these new tools.

For this more expansive topic, we'll be going a step further and actually creating an entire *sandbox project* to use. Unity provides specific **Project Templates** based on whether you're creating a 2D or 3D project, and based on those templates, it will enable (and disable) various tools associated with that project category.

Since we want access to the full breadth of 2D tools and systems, we need to start a new project with those options enabled.

Let's start by closing Unity and bringing up the Unity Hub.

1. Click the **New Project** button on the home screen, then select the first **2D** option in the list of available templates ([Figure 15.2](#)).
2. Set the project name to be **2D Sandbox**, then press the **Create Project** button in the lower right-hand corner.

Figure 15.2: By creating a new project from a 2D template, your editor and project will be set up with the tools and systems required for easily creating a 2D game.

Once your new Sandbox project is ready, it's time to head over to the Unity Asset Store to find a 2D package to use.

3. You'll find many options, but it's suggested that you download a retro, Pixel-Art package to follow along with, like the SunnyLand asset, as shown in [figure 15.3](#)

Figure 15.3: The Unity Asset Store has many 2D packages to choose from.

Once you've imported these assets into the project, it's time to learn about the backbone of Unity's 2D system: Sprites.

Sprites

3D games use models to represent characters, items, and environments. These objects can range from hundreds to thousands of polygons, with modern graphics requiring several textures on top of the mesh to get the high-fidelity look players expect.

2D assets, on the other hand, are made from two basic objects - a flat square and an image texture. When combined, these objects are known as a **Sprite** - the core object of a 2D game scene. The simplicity of sprites allows you to create a game with *just* textures, opposed to the multi-step process of modeling, texturing, and animating 3D objects.

2D games also have the benefit of utilizing **Sprite Batching**, a method of rendering hundreds of thousands of images without breaking a sweat. If you want to make a game with *tons* of objects on-screen at the same time, you may want to consider going the 2D route.

You can easily find sprites in your project by searching for them specifically.

1. In the Project Window, click the **Search By Type** button, and select **Sprite** ([Figure 15.4](#)). This will list all the sprites from the Asset Package we just imported.

Figure 15.4: Use the Search By Type options to find all assets of a specific category. While we're searching for Sprites, this can also be used for finding Prefabs, Meshes, AudioClips, and more!

*Importing Sprites: You can also import your own sprites directly from image files that you've authored. Use the **Import New Asset** option, then select a png, JPEG, bmp, or even PSD (Photoshop format). This will add the file into your project. Just set this asset's **Texture Type** to **Sprite** and it'll be ready for action!*

2. Once you have sprites listed in the Project window, simply drag them into your scene to build your environment ([Figure 15.5](#)). Since sprites are GameObject, they have all the Transform data you'd expect. Move them, rotate them, and scale them as you see fit.

Figure 15.5: Building a scene out of sprites is as easy as dragging them from the Project window and up into your Scene.

Now remember: while we're learning about tools for 2D games, Unity is *still* a *3D Engine*. The default camera is set to 2D mode, but you can toggle that at any time on the top of the Scene Window.

3. Press the **2D** button to toggle between the 2D and 3D camera modes, as shown in [figure 15.6](#)

From various angles, you can get a better view of how objects are ordered, and perhaps fix any **Z-Fighting** issues (the flickering of 2 or more sprites that share the same Z-position). This method of creating 2D games within a 3D engine will give you many interesting opportunities as you build your game.

Figure 15.6: Sprites are just flat, 2D objects in a 3D scene.

Back to our 2D objects - let's look at the **Sprite Renderer** component, the script that makes Sprites so flexible.

4. Select a sprite that you've placed in the scene, and scroll down in the Inspector Window, as shown in [figure 15.7](#). The Sprite Renderer is where you'll be specifying the display parameters of this object.

Figure 15.7: The Sprite Renderer is a key component in Unity's 2D system.

As you can see in [table 15.2](#), the Sprite Renderer script gives us all the parameters we could need for tweaking how a sprite looks on-screen.

Sprite	The 2D Sprite texture being displayed on this object. Images have to be imported with Texture Type set to 'Sprite'.
Color	The color tint of the sprite. By default, sprites are tinted white, which displays the colors as they appear in the texture.
Flip (X/Y)	Toggle The flip parameters if you want to mirror the sprite horizontally, vertically, or both.
Draw Mode	Should this sprite be drawn normally (Simple), or should it be scaled, or repeated, based on slicing data (Sliced, Tiled).
Mask Interaction	If there's a mask in the scene, this parameter sets how the sprite will interact with that mask (None, Visible Inside Mask, Visible Outside Mask).
Sprite Sort Point	Whether the sorting of this sprite is done from the object's Center or Pivot point.

Material	Sprites can be given a special material to use for rendering, but most will look fine with the default.
Sorting Layer	Sprite can be given a specific sorting layer for grouping, and display, purposes. First sprites are z-sorted by layer, then by ‘Order’.
Order in Layer	A value that sets the z-order of a sprite, drawing an object in front, or behind, the others.

Table 15.2: The parameters of the Sprite component.

And while sprites can be imported one image at a time, the real power of the Unity Sprite system comes when multiple images are on the same sheet, allowing those objects to be **Batched** for a big performance boost. Let’s look deeper into how we can organize our 2D images into *Sprite Sheets*.

Sprite Sheets

In the world of graphic design, one of the most time consuming processes is the **Slicing** of images: breaking a large image into smaller images to use in a website, application, or game. Previously, slicing was a process that could easily eat up much of an artist’s time. Luckily, Unity has powerful tools for slicing up an image and to create an asset known as **Sprite Sheets**.

Sprite Sheets are single images that contain many sub-images. In Unity they’re used for everything from defining UI assets to breaking up the sprites of a Tile-based environment.

Let’s take a moment to step through this process - cutting up an image and turning it into a **Tilesheet** (square sprites used for level creation).

1. Start by searching your project for a Sprite sheet image to use. This will be a single image with a grid of other images on it, similar to the asset shown in [figure 15.8](#)

Figure 15.8: A Tileset texture will look something like this, with several sprites clearly broken separated on a single image.

2. Select this image to bring up the Import Settings in the Inspector window. Change **Sprite Mode** to **Multiple**, then press the **Sprite Editor** button.

Figure 15.9: Select ‘Multiple’ as your sprite mode to turn an image into a Sprite Sheet.

3. The Sprite editor is where you’ll be defining the sub-objects of a Sprite sheet. You can do this manually by simply dragging a rectangle around any Sprites you want to use in your game. If the sheet has been properly authored, however, you can also use automated options for slicing up your sheet.
4. To automate the creation process, select the **Slice** option at the top of the window to bring up our options ([figure 15.10](#)).
5. Set the Type to **Grid By Cell Size**, then enter the pixel size of your grid. A red grid will appear to show you how the tool will be slicing up the image. If you’re happy with the alignment of the grid, press the lower-right **Slice** button to start the process.

Figure 15.10: Unity can automatically Slice and Organize the sprites in your tilesheet.

Unity will then automatically slice up valid sprites based on the settings you entered. By default it will discard any empty sprites in the sheet, as well as overwrite any existing sprites defined in this file.

6. If the process was successful, you will be able to click on the individual tiles to see their sprite information displayed ([figure 15.11](#)).

Figure 15.11: A successful slicing of our sprites. Our tilesheet is now ready for use in a Tilemap!

Tilemaps

If you’ve played a game from the 8-Bit or 16-bit eras - whether it was a sidescroller, top-down adventure game, or even JRPG - you should be familiar with **Tilemaps**. Tilemaps were how levels were made for games before the introduction of 3D. They’re a grid-based data format that stores sprites used in a given level. Creating environments from many small objects (known as Tiles) let early developers build large levels within the hardware’s memory constraints.

Nowadays, the use of Tilemaps has less to do with conserving memory and more about the rapid creation of levels with a retro aesthetic quality, as shown in [figure 15.12](#)

Figure 15.12: Sidescrollers - especially retro-style ones - often use Tilemaps for level creation.

With our Tilesheet sliced up, we're ready to start making our Tilemap.

1. Place a Rectangular Tilemap grid in your scene by Right-clicking the hierarchy window and selecting **2D Object > Tilemap > Rectangular**.
2. Then, from the top of the editor menu, select **Window > 2D > Tile Palette**. Select the **Create New Palette** option to bring up a subwindow ([figure 15.13](#)).

Figure 15.13: Before we can use our sprites, we need to convert them to a Tile Palette.

3. Within this ‘Create New Palette’ subwindow, keep all options set to their default, and press **Create**. After pressing the Create button, it’s going to ask you where to place the new palette file. Create a folder in your Assets directory called Tiles, then place the new Tile Palette there (2D Sandbox > Assets > Tiles).
4. It will then prompt you to “*Drag a Tile, Sprite, or Sprite Texture asset*” into the Tile Palette window. Find the tileset you just Sliced up, and drag that into the Tile Palette popup.
5. Once Unity is done processing the tiles, the Tile Palette window will now be displaying a grid of tiles, as seen in [figure 15.14](#)

Figure 15.14: Our tile palette, with a grid of Tiles Sprites to place.

6. You should notice a Toolbar at the top of the palette window ([figure 15.15](#)). This is much like the Terrain editor tools, where each option lets you perform a different useful task when designing an environment.

Figure 15.15: The tools for making a Tilemap, as well as the active Tilemap that you’re editing.

7. You can use these buttons to switch between the various tools, or toggle the selected tilemap that you want to edit. When the palette is

visible, you can also use hotkeys to toggle between the selected tool ([Table 15.3](#)).

Table 15.3: The Tilemap tools, and the hotkeys to quickly swap between them.

8. Using the **Tile Palette Toolbar**, select the Paint tool, then select one of the tiles from the palette. You can now click on the Tilemap grid to place that tile wherever you want ([figure 15.16](#)).

Figure 15.16: Use the ‘Paint’ tool to place the selected tile in the Tilemap.

One issue you may encounter (and as seen in [figure 15.16](#)) is that the scale of the tiles may not match the size of the tile grid. There are two ways to fix this. They are as follows:

1. The *first* option is to adjust the grid cell size ([figure 15.17](#)). Select the Grid object, and in its Grid component, adjust the the Cell Size until the tiles fit properly

Figure 15.17: Because the tiles were made at a 16x16 resolution, our first possible solution is to set the Grid’s Cell Size to match that.

In some cases this is the proper fix, but when dealing with *very small* images - like pixel art - this will also decrease the size of your collision objects, possibly decreasing the accuracy of physics calculations (high speeds and small colliders don’t play nicely together)

2. In these cases, you can use the second option, which is simply to bring in your images at a lower Pixels Per Unit. Select the tileset image to bring up the Import settings, then change the Pixels Per Unit value until the tiles fit nicely in the grid ([figure 15.18](#))

Figure 15.18: Because these pixel art tiles were created at a 16x16 resolution, we’ll need to set the Pixels Per Unit to 16.

3. With the tile sprites fitting properly in the grid, it’s time to get creative. Place some ground, ladders, foliage - any sprites from your

Tile Palette that you feel will make the environment look interesting ([figure 15.19](#)).

Figure 15.19: Variety is the key to a well-crafted environment.

We can also make our environment look more interesting by adding a non-interactive Background Tilemap.

4. In the hierarchy, right-click on the **Grid** object and add a **2D Object > Tilemap > Rectangular**. Name it **BGTilemap**. You can now select the BGTilemap in the Tile Palette, letting you place tiles behind the main Tilemap sprites.

If you find that your BGTilemap objects are actually appearing in FRONT of the other Tilemap, you'll want to tweak the display order of your **Tilemap Renderer**, the component that manages how a tilemap gets displayed.

5. Select the BGTilemap object, and in the Tilemap Renderer, set the **Order in Layer** parameter to **-1**. Just like with the Sprite Renderer, this value will determine what tilemaps are displayed in front of others. By setting it to -1, we want it to be rendered behind any tilemaps with larger values (15.21).

Figure 15.20: A Background Tilemap gives your level more depth, and gives you a place to do some non-interactive environmental storytelling.

The Tilemap systems allow you to quickly create and iterate on levels with a limited number of assets. However, as we talked about at the start of this section, this *look and feel* is undeniably *retro* ([figure 15.21](#)). This may be what you're looking for, but other designers may need a more *modern* 2D aesthetic.

Let's pull up a demo project to learn the *advanced 2D tools* provided by Unity.

Figure 15.21: Add some polish with additional sprites - sky, trees, and rocks. Set them to appear behind all the tilemaps to finalize your 2D tile-based environment.

Studying The Lost Crypt

It's time to jump into the more advanced aspects of Unity's 2D tools and systems, but to do that, let's first grab a project made to showcase these tools: **The Lost Crypt**.

1. Bring up the Unity Asset Store in your browser (assetstore.unity.com) and search for "Lost Crypt". It should be one of the first options that comes up. Once selected, press **Add to My Assets**, as you see in [figure 15.22](#)

Figure 15.22: This demo project does a great job showing off the advanced 2D systems available to you in a way that's both fun and informative.

Overwriting Projects: Since the Lost Crypt is a full project asset, importing it into the 2D Sandbox will overwrite all the Tilemap work we've just done. This is fine, since the rest of our chapter will focus on the features used in this Sample Project. However, if you wish to keep playing with tilemaps and sprites, and do NOT want to overwrite that work, make a fresh project now to import the Lost Crypt into.

2. Once you have the Lost Crypt project downloaded and imported, open the **Main** scene and take a look around ([figure 15.23](#)). In the Hierarchy Window, you'll see how they've organized the scene, with empty objects called Cameras, Lights, Environment, etc. serving as the storage containers for various game objects. Storing child objects like this is a great way to stay organized as a scene gets complicated.

Figure 15.23: The Lost Crypt project uses Unity's 2D systems to create a beautifully polished demo.

3. After you've looked around to get a basic idea what the scene looks like, press the **Play** button to start the demo. Use the Arrow Keys to move and Spacebar to jump. Run to the rightmost side of the scene, grab the scepter, then run to the left. It may be short, but you should have noticed an impressive amount of polish throughout the demo. Here's a list of the most noteworthy aspects...

- **Smooth Camera:** From the slow pan in when you start, to the transition to a wide shot when the item is grabbed, the camera movement throughout the demo is highly cinematic.
- **Character Animation:** The heroine you’re controlling is well animated, with both elements that have been hand-animated (arms and legs) and elements that use physics to dynamically move (her ponytail).
- **Curved Environment:** The hills you’re running across aren’t just straight paths - they’re hills with a subtle curve to them, making the environment feel very organic.
- **Environmental Particles:** You’ll have also noticed dust motes and fiery torches made out of particles. These were created using the same Particle System we learned about in previous chapters!
- **Parallax:** You should notice both foreground and background elements, all moving at faster or slower based on their distance from the camera. This extra depth is known as Parallax, which we’ll talk about towards the end of the chapter.

2D Physics and Movement

Much like the 3D physics systems of Unity, 2D physics and movement are completely driven by components. If you select the object “Sara Variant”, you’ll see both the physics objects (**Rigidbody 2D** and **Capsule Collider 2D**) as well as a custom Character Controller 2D script, which handles the movement and animation of the heroine ([figure 15.24](#))

Figure 15.24: Just like our 3D hero, this 2D character has a Collider object and a custom made Controller script to handle movement of the Character.

Looking through the available components for our gameObjects, you should notice that almost every physics related script also has a 2D counterpart, like the 2D colliders in [figure 15.25](#)

Figure 15.25: 2D Colliders - just like 3D colliders, but with a 3rd axis that stretches to infinity.

Other than the slight variation of component name, the systems are almost identical. Triggers, onCollision functions, gravity, mass...everything you know about 3D physics is applicable here.

2D Animation

Animation is a design tool that, when used properly, can add considerable polish and professionalism to any game. When we first learned about animation, it was in the context of a 3D hero made out of individual objects. By animating those objects (hands, feet, body), we could make the hero run, jump, and attack. This was a process called **Object-Based Animation**, and it worked well for that example.

Sometimes, however, you'll need to animate a character more complex than a small, heroic sphere. When your game is filled with animals, monsters and humanoids (as shown in [Figure 15.26](#)), you may need to use a more *flexible* form of animation, **Skeletal Animation**.

Figure 15.26: The Heroine of the demo animates with an impressive smoothness. Even her long braided hair flows with every change of movement.

Skeletal Animation is a process where, instead of directly moving and rotating *objects*, you move and rotate **Bones**. These bones are associated with objects, which will be deformed in an organic way. Skeletal Animation allows for legs that *bend*, hair that *flows*, and torsos that can *hunch*.

There are some terms associated with Skeletal Animation that we should go over before diving into the tools. Those terms are as follows...

- **Deformation:** When a sprite image is skewed, stretched, and squished in real-time.
- **Bones:** Objects used to deform a given character or object. Bones are only displayed in the editor.
- **Rigging:** The process of associating bones with the objects they will deform.
- **Sprite Skin:** The component that holds the bones that affect a sprite.

If you open the **Sara Variant** prefab, you'll get to see the skeletal system in action. All the long white objects shown in [figure 15.27](#) are Bones. These are all associated with certain sprites, and all animation is done by changing the position and rotation of these Bone objects - never by touching the sprites that make the character.

Figure 15.27: The bones are represented by the long, white shapes that cover the character. These are used to stretch and deform the sprite they're attached to.

The ponytail is a great example of how the Skeletal animation system works. There is only one Ponytail sprite, but there are several bone objects that can be moved and rotated to adjust how the hair is displayed. Take a moment to move around the bones in this prefab to see how the attached sprite is affected. As bones move and rotate, the character will as well.

Another benefit of bones is that, because they are gameObjects themselves, they can be animated with keyframes.

Bring up Unity's Animation Window, select the Sara Variant object, and bring up her **sara_run** animation. You'll see a 1:00 second animation with several keyframes, all set for the leg, arm, and torso bones ([Figure 15.28](#)). Scrub the animation timeline to see how animating these bones directly affects the character sprites.

Figure 15.28: Bring up the Animation window to see how the position and rotation of the bones can be animated using keyframes - just like our Hero in the Animation chapter.

You will notice that the bones in the Ponytail object aren't listed. This is because those bones actually use **2D Hinge Joints** - the 2D version of the joints we used [Chapter 6](#) "The Physics of Fun". By making the bones of the ponytail linked by Physics Joints, Sara's long hair falls and flows based on gravity ([figure 15.29](#)). This is a great way to make use of physics in your character animations: realistic movement, with no hand-placed animation needed!

Figure 15.29: Bones can make use of the 2D Physics components to animate and move in a more automated way (opposed to being animated via keyframes).

An important note: Unity also allows for 3D skeletal animation, but those animations need to be created outside the Unity toolset. By using Maya, 3D Studio Max, or Blender, you can create 3D skeletal animations for your characters and import them.

If you're looking to make a 2D game, however, know that Unity gives you everything needed to create impressive 2D skeletal animations directly from the editor.

Cinemachine Cameras

As you investigated the Lost Crypt scene, you may have noticed a small red Gear/Camera icon close to the player's starting location. You may have also noticed that there were *several* cameras in the scene, where we traditionally only used *one*.

This is because the demo used Unity's cinematic camera system, known as **Cinemachine**. You can unroll the Cameras group in the Hierarchy window ([figure 15.30](#)) to see all the camera related objects in the demo.

Figure 15.30: That tiny Gear icon indicates a given Camera is under the control of the Cinemachine system, which allows you to easily transition from one camera to another.

With the Cinemachine components, you can easily transition between cameras. You can set these transitions to move at different speeds, automate effects such as 'shaky cam' (making a camera feel like it's being held by someone) and limit the camera bounds directly from the Cinemachine parameters. It's a powerful tool that the Lost Crypt makes great use of.

1. Select the **CM Vcam – Near** camera so we can learn about one of the key Cinemachine components: the **Cinemachine Virtual Camera** ([figure 15.31](#))

Figure 15.31: Virtual Cameras store all the data used by Cinemachine to transition between different camera views.

2. Virtual Cameras are how you set up the various cameras that the Cinemachine system will be transitioning between. When editing a Virtual Camera, you can press the **Solo** button (at the top of the

component) to see that camera in the Game view. Some of the other important parameters are listed in [table 15.4](#)

Save During Play	Select this if you're doing a lot of tweaking while the game is running. If enabled, all changes made in run-time will be made to the component (opposed to changes being reset when the game is stopped).
Priority	If the Cinemachine logic is trying to move between multiple cameras, it will use the Priority value to determine which to actually use (Lower Value means Lower Priority).
Follow	The Object to follow (if the 'Body' parameter has logic that requires an object to follow).
Look At	The Object that this camera will try to look at (based on the 'Aim' parameter).
Standby Update	How often this camera should update when the camera isn't currently 'Live'.
Lens Ortho Size	The Lens settings of this camera.
Transitions List	A list of actions to trigger when this Virtual Camera is set to 'Live'.
Body	The main method of movement for this camera (follow an object, move along a set path, etc).
Aim	The main method for determining what this camera should look at.
Noise	A setting that adds some random positioning and rotation to this camera. Can be used for a Screen Shake or a 'Handheld Camera' effect.

Table 15.4: Several of the key parameters available when tweaking a Virtual Camera.

3. While the Virtual Cameras will store the various views that you can switch between, it's the **Cinemachine Brain** component that determines which camera should be **Live** (enabled), and which cameras should be disabled.
4. Select the **Camera - Main** object to see the various parameters available in the Cinemachine Brain component ([figure 15.32](#))

Figure 15.32: The Cinemachine Brain should be placed on the Main Camera object. It controls the current camera that is being used in the scene, and handles all the heavy lifting when transitioning from one virtual camera to another.

Since the ‘brain’ does the majority of work here, and each of these parameters will help achieve the cinematic look you’re going for, let’s examine these options closer in [table 15.5](#)

Show Debug Text	Display the current status of the Cinemachine Brain (current transition information, ect).
Ignore Time Scale	Cinemachine logic will ignore changes to the current time-scale, so if you have a scene that’s sped up, or slowed down, the camera will move in normal time.
World Up Override	Override transform that will change the up-axis for this camera (Y axis is ‘Up’ by default).
Update Method	How often, and when, Cinemachine virtual cameras get updated.
Default Blend	The smoothing function for when Cinemachine transitions from one virtual camera to another.
s (Blend Seconds)	How long it takes, by default, to transition from one camera to another.
Custom Blends	An option list of transition overrides between two cameras.
Events	Events that can be triggered in certain situations (when the camera cuts, and when a camera is activated).

Table 15.5: Parameters you’ll be using when hooking up a Cinemachine Brain.

True to its word, the Cinemachine system lets you produce some exceptionally cinematic camera movements with little effort. It can also be used when making 3D games, allowing you to easily transition between various cameras with just a few components.

The Power of Parallax

There’s one final 2D effect that we want to cover, and that’s the concept of **Parallax**. This is a natural occurring phenomenon where objects further from a viewer will move slower, and objects closer will move quicker.

Parallax is a great way to give extra depth to your 2D scene, and there are two easy ways to implement it.

The first way is to simply set up your 2D scene to use a Perspective camera, then place distant objects further away from foreground elements, as shown in [figure 15.33](#)

Figure 15.33: You can mimic the Parallax effect by moving distant objects further back in the scene.

The second option when implementing a Parallax effect is to create a new component that moves objects a bit more, or a bit less, based on a Parallax multiplier. That's actually what the Lost Crypt team did, creating a script called **ParallaxLayer.cs** that was attached to several objects. Open that file to see the straightforward implementation, which takes a multiplier and adjust the position based on change in camera position.

If you want to see the impact of Parallax, you can put a **return** statement at the top of the **LateUpdate()** call ([figure 15.34](#)).

Figure 15.34: You can disable the demo's Parallax effect by simply leaving the *LateUpdate()* function early. Toggling the effect lets you appreciate its impact.

The early *return* will ignore the effect entirely, showing you how flat and static the Lost Crypt environment looks without that depth. Reenabling the effect allows you to see the importance of depth in your 2D environments.

Parallax is an easy and straightforward way to polish the visuals of any 2D game.

Conclusion

While Unity has become a true 3D powerhouse, it's also proven itself unstoppable in the area of 2D. Developers looking to recreate the retro magic of old-school 2D games, or perhaps make an animated adventure that nobody's seen before - will have all the tools they need in Unity. Sprites, tilemaps, physics engines, 2D Skeletal Animation, and Cinematic cameras: it's all there, all flexible, and all incredibly easy to use.

With a strong understanding of the systems available for both 2D and 3D games, we're now going to take a step back for a broad look at the *types* of games you may want to create. Since Unity has the power to make Sidescrollers, RPGs, Shooters, and any other style of game, it's important to categorize, inspect, and learn about the **Genres** that players have come to enjoy, and that designers enjoy creating.

Questions

1. What are some of the benefits of creating a 2D game instead of a 3D game? What are some of the drawbacks?
2. Sometimes, if you paint sprites into a Tilemap, the sprites will appear too small. What are the solutions to this problem?
3. True or False: You can only have one tilemap in a scene at a time?
4. Tilemaps have been a useful tool for creating games for the last 40 years. What are some of the benefits, and drawbacks, of using tilemaps in a modern game?
5. If sprites intended as background elements are appearing in the foreground, what parameter do you adjust to fix this?
6. True or False: Bones can have physics components attached to them, allowing them to move based on the physics system (opposed to keyframe-based movement).
7. If you see a Red Gear/Camera icon in a scene, what does that represent?
8. True or False: The Cinemachine system is ONLY for 2D projects. You cannot use it for 3D games.

Key Terms

- **2D Project:** A game project in Unity that uses flat sprites for characters, items, and the environment.
- **Sprites:** A flat plane with an image on it. The sprite is the basic building block of 2D graphics.
- **Z-Fighting:** When sprites share the same depth position, then will start to flicker (ie. fight) as the rendering code struggles to determine which object is in ‘front’.
- **Sprite Sheets:** A single image that contains several sprites.
- **Slicing:** The process of taking a single image and cutting it up into its sub-images.
- **Sprite Batching:** Drawing hundreds (or even thousands) of sprites in a single call, significantly improving performance.

- **Tilemaps:** A grid of sprites that define a level or environment.
- **Tile Palette:** A collection of sprites that can be used to fill a Tilemap.
- **2D Physics:** Much like 3D physics, the 2D physics system lets you use components to set the shape of a 2D object. You can then attach a 2D RigidBody component to have that object interact with gravity and other 2D objects in the physics simulation.
- **Object-Based Animation:** A method of animation where you set keyframes for the individual objects that make up a character.
- **Skeletal Animation:** A method of animation where bones are attached to objects, and as the bones are moved, the sprites (or mesh) that makes up the character is deformed. Useful for animated characters that need to bend and flex (animals and humanoids).
- **Cinemachine:** A Unity system that manages the transition between different camera views. Use it to create a more cinematic experience with the player.
- **Virtual Camera:** The Cinemachine cameras that will be blended as the game determines a change in camera is required.
- **Parallax:** The feeling of depth that comes from seeing distant objects moving slow, and close objects moving quickly. This is a useful effect for making 2D games feel less ‘flat’.

CHAPTER 16

Mastering the Genres

During the early years of game development, designers were faced with a blank canvas of possibility. Almost every idea was new, and only by implementing those ideas in a game - then testing those games in the market - could designers sort the *good* concepts from *bad* ones.

Luckily, with over 50 years between the first arcade games and the modern industry, there have been enough successes and failures to see what *types* of games players enjoy playing. And when these broad concepts are found to have a lasting appeal, that gameplay combination is organized, categorized, and labeled as an official video game **Genre**.

Many genres you'll recognize by their acronyms. RPG, or *Role-Playing Games*, take the player on an epic journey, leveling up through multiple battles to save a Kingdom, World, or Universe. FPS is another well-known Genre, which stands for *First Person Shooter*. These games put you behind the barrel of a gun, shooting your way through fast paced, adrenalin-fueled levels.

There are, of course, countless other genres and subgenres. As a game designer, it'll be important for you to understand the production implications of creating a game in one Genre over another. By understanding what makes each genre unique, as well as appreciating scope differences between them, you'll be one step closer to mastering the game development process.

Structure

In this chapter we will discuss the following topics:

- Concerning Scope
- Sidescrollers
- Top-Down Adventure

- FPS - First Person Shooters
- 3rd Person Games
- RPG - Role-Playing Games
- Strategy Games
- Puzzle Games
- Creative Juxtaposition
- Player Expectations

Objectives

In this chapter, we'll be diving into the various video game Genres that are commonly used in the industry. You'll visit the Unity Asset Store to look through the demo projects available for these genres, allowing you to get a head-start on the development process.

We'll also discuss about managing scope, and the different ways that you can design a clever game that stands out amongst other titles. By the end, you'll have a strong understanding of player expectations across various genres, and the work required to get these games made.

Concerning Scope

Before jumping into any specific Genre, it's important that we acknowledge **Scope** first.

Scope is a broad concept that measures all the coding, art, writing, testing, and marketing needed for a given design. It's a metric for understanding the work required to get your game from the conceptual phase to completion.

Different game genres will each have a different *scope* (for example, making a successful FPS will require more work than an Endless Runner). There are many reasons for this as follows

- **Player Expectations:** When a genre is popular, players will have pre-existing quality and content expectations. Since common genres will have a large pool of possible players, it'll be tempting to design a game with a popular genre in mind. Be warned, however - your game will be constantly compared to those previous games within the genre.

If these games were made by AAA studios, it'll be difficult to stand out with a limited, healthy scope.

- **Solution:** Be clever and creative about your use of a genre. Don't try to match existing games with what *others* do well. Instead find a clever gameplay gimmick that *you* can do well. Creative thinking at the design stage can make any game stand out.
- **Art Requirements:** It can be tempting to jump right into a big, awesome, art-intensive genre (3D Platformer, RPG, FPS). It's also an easy way to burn-out quickly, getting excited about a game design that requires too much work on the asset creation side.
 - **Solution:** Pick either a genre smaller in scope, or pick an art style that doesn't require a massive amount of AAA 3D graphics. Time yourself making a test object, then multiply that by the number of objects in your game (+25%, to give yourself some breathing room). You'll quickly figure out if the asset requirements for your game are out of scope.
- **Code Complexity:** There are certain systems - Multiplayer, Cross-Platform Mods, Complex AI - that will require a lot of debugging time, which will drain your time and motivation. You can also over complicate a simple system through poorly written code, quickly implementing a feature only to realize the code is hard to read and difficult to build upon.
 - **Solution:** Try to keep your gameplay design limited to systems you're comfortable writing. You can also download existing systems that do the heavy lifting (*Mirror*, for instance, is an amazing and robust networking API). But always take the time to write clean and well documented code. Assume you'll be coming back to these scripts over the next few years. Will '*future you*' understand the code written by '*present-day you*'?

Considering the scope of a possible game design, and being honest about the work involved, will help to set your project up for success. The best way to ensure your game gets done is to keep game development fun. And nothing kills the fun of development like realizing a design is bloated and hard-work has to be tossed-out.

Focus on limiting scope and doing a *great* job with a small concept, opposed to doing an *OK* job on a wider concept.

Sidescrollers and Platformers

Two genres that need almost no introduction, Sidescroller and Platformer games have been a staple of the industry since the early 80's. Starting with blockbusters such as Donkey Kong and Jumpman, players loved navigating a hero through levels filled with tricky platforms and deadly hazards (spikes, pits, fireballs, and so on).

Games such as Super Mario Brothers took the platforming genre and added camera movement, allowing for long levels and more enemies. Factor in power ups, secret areas, and a wide variety of environments (water levels, underground caves, lava-filled fortresses) and the ‘Sidescroller’ genre cemented itself as the quintessential video game experience.

Because of its longevity, designers have been able to play with the Sidescroller/Platformer formula for almost *40 years*. In that time, several subgenres have emerged. The player still navigates a nimble hero around dangerous levels, but clever designers have expanded upon this idea in interesting ways, like the **Metroidvania** subgenre, as shown in [figure 16.1](#)



Figure 16.1: *Ori and the Blind Forest* - an award winning Metroidvania-style Sidescroller built in Unity.

- **Metroidvania**

A sidescroller that focuses more on exploration than conquering ‘levels’. The player is given an expansive world to find their way through. Backtracking is important, as well as gaining upgrades that allow you to reach previously inaccessible areas.

- **Endless Runner / Jumper**

Stripping the sidescroller down to its simplest form of interaction (often a single button press to *Jump*) the Endless Runner / Jumper subgenre pits the player against a constantly moving camera and levels that can randomly build themselves, allowing for ‘Endless’ gameplay. This is a style of Platformer perfectly suited for Mobile.

- **SHMUP**

While SHMUPs (Shoot-Em-Ups) typically have the player controlling a vehicle instead of a character, this genre has its roots grounded in the Sidescroller style of game. Typically piloting a spaceship from left to right, the player has to use precision to hit incoming enemies, dexterity to dodge incoming bullets, and tactics to manage upgrades.

If you’re planning on creating a game in this genre, you’ll find several starter projects and game templates.

The first, which expands on the Lost Crypt demo, is called the **2D Game Kit**. This expansive, well integrated toolkit adds enemies, UI examples, moving platforms, trigger systems, and boss battles to what you’ve already seen in the Lost Crypt. The team at Unity even made an entire written manual to go over, in depth, the tools provided especially for this demo.

If this lines up with your design goals, the 2D Game Kit can be downloaded - for free - from the Unity Asset Store ([figure 16.2](#)).

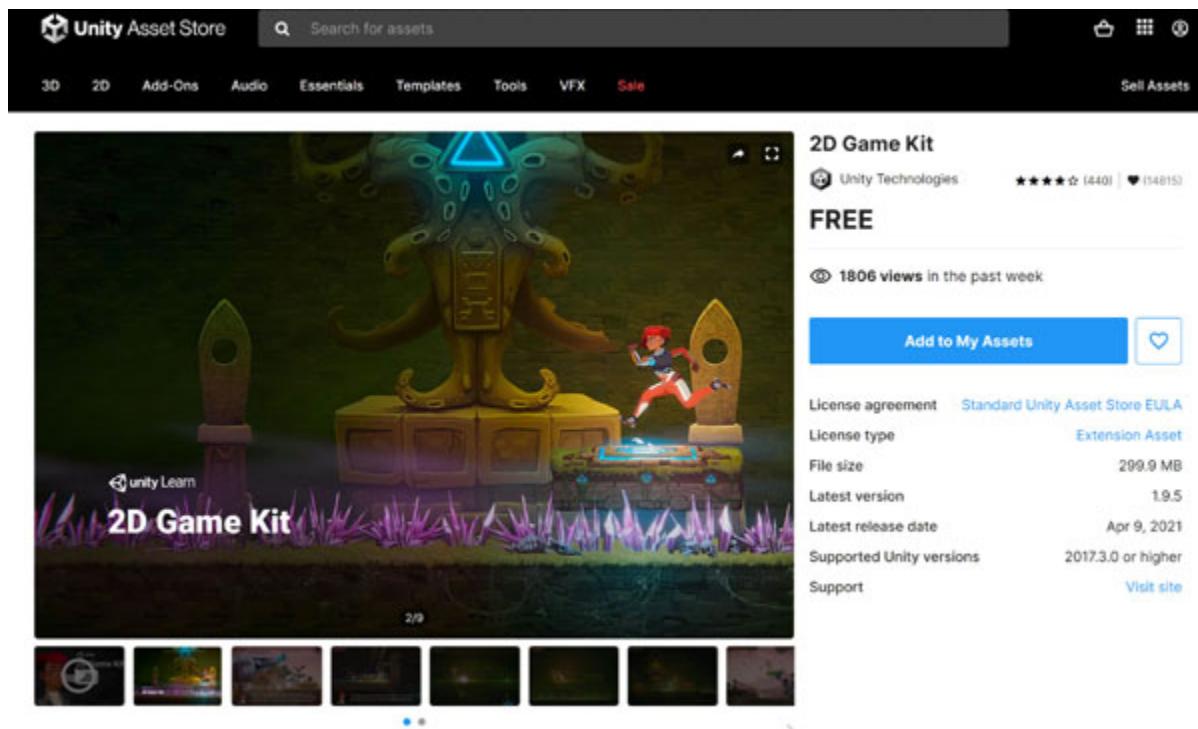


Figure 16.2: The 2D Game Kit is a powerful set of tools for any Sidescroller or Platforming game project.

Another powerful Asset Package for creating 2D Sidescrollers and Platformers is the **Corgi Engine** ([figure 16.3](#)). While this comes with a cost (unlike the included samples from the Unity team) the Corgi Engine has been used to ship many games, and continues to be updated with awesome new features.



Figure 16.3: The Corgi Engine continues to be updated - a great foundation for any Siderscroller or Platformer you'd want to build.

A Note on Paid vs. Free: We've been making heavy use of free assets so far, but it's important to point out the benefits of the 'Paid' options on the Asset Store. If you find a highly rated asset, you can look through its release history to see all the updates and improvements being made. Not only do these serve as a strong foundation to build upon, you also get the benefits of a passionate team keeping the asset fresh with new tools and systems. Free samples, on the other hand, tend to serve as a foundation where you'll be building out the additional functionality. Both methods have their benefits, but sometimes you'll find the 'Paid' option will save you more long-term time and effort.

When it comes to the *Scope* of a Sidescrolling / Platformer game, there are a few things to keep in mind which are listed below:

- **Competition**

There are a LOT of Sidescrollers on the marketplace, so your game will have considerable competition. Pick a strong gimmick (or several)

to get noticed.

- **Movement Perfected**

Perfect movement is *vital* for Sidescrollers. You should finalize the move set of your hero before starting on anything else.

- **Mechanics First - Level Design Second**

The design and implementation of clever gameplay mechanics should come before you start to build your levels. Once the mechanics have been tested in a sandbox environment, and deemed to be fun, deep, and engaging, you can move into the sandboxing of levels. And, as always, remember to white-box your ideas in sandbox scenes.

- **Landmarks and Unique Challenges**

Sidescrollers are typically made with Tilemap based environments, which are easy to create. This allows you to make an expansive world - lots of levels, with minimal asset-creation effort. Just make sure areas are designed with unique challenges and landmarks, otherwise levels will start to feel like ‘more of the same’.

While Sidescrollers are plentiful throughout the industry, they also give you an opportunity to put your own crazy spin on a genre that people know and love. And chances are, if you use the Sidescroller/Platformer genre to make something unique and clever, they’ll love your game, too!

Top-Down Adventure

Another classic genre, around since the late 70s, is the Top-Down adventure. First making an appearance on the Atari 2600 with the genre defining “Adventure” ([Figure 16.4](#)), these games give the player a top down view of a sprawling world. Mazes, enemies, puzzles and more fill the environment, forcing players to master both their combat and puzzle solving skills.



Figure 16.4: The birth of a genre: ‘Adventure’ for the Atari 2600.

Since “Adventure” first enthralled players, Top-Down Adventure games have grown and matured, with Unity being used to create several high profile additions to the genre.

The entralling 2022 **Tunic**, for instance, used the power of Unity to make a beautiful, intricate, and clever adventure across a charming 3D Isometric world ([figure 16.5](#)).



Figure 16.5: The team at Isometricorp Games used Unity to build the beautiful Top-Down Adventure game **Tunic**.

Just like with side scrollers, designers have had years to iterate on this classic formula. Traditionally, a Top-Down Adventure game will have an overworld for a player to explore, with sub-levels to find and explore as well. Weapon and armor upgrades can be found, or purchased, and clever puzzles will test the players logic and reasoning, as well as their skills with a weapon. New subgenres have grown from this basic idea, as designers have pushed this gameplay in clever directions...

- **Dungeon Crawler**

Where the typical Top-Down Adventure balances Overworld and Underworld exploration, Dungeon Crawlers throw players deep into deadly Labyrinths. Combat takes priority as the player fights off hordes of dungeon-dwelling creatures. A victorious player will emerge from the depths with the most coveted of prizes: MASSIVE LOOT. Loot allows the player to upgrade equipment and crawl even deeper into the depths.

- **Roguelike**

A cross between an Adventure game and an RPG, the **Roguelike** subgenre has players fighting their way through randomly-generated dungeons of increasing difficulty. This is often done in turn-based fashion, with the added excitement of **Permadeath**: once the player dies, they have to start the entire game over from the beginning.

If you’re looking to create a Top-Down Adventure game, the Unity Asset Store has some great packages to get you started.

Directly from the Unity Team, there’s a free 2D Top-Down tutorial called **2D Beginner - Tutorial Resources** ([figure 16.6](#)). It will walk you through some of the concepts of a Top-Down game, help you implement them, then leave you with several useful scripts to use on your own adventure.



Figure 16.6: The 2D Beginner package from the Asset Store will guide you through some tutorials to get you started towards your own Top-Down Adventure game.

If you want something a bit more fleshed out, there's an asset for purchase simply called **TopDown Engine** ([figure 16.7](#)). Made by the same team behind the sidescrolling Corgi Engine (which we mentioned in the previous “Sidescrollers and Platformers” section) the TopDown Engine gives you a large assortment of tools and systems. Their team is continuously updating this package, so you'll definitely get your money's worth here.



Figure 16.7: The TopDown Engine asset will give you a strong foundation for any 2D or 3D Adventure you'd want to make!

As for the scope of creating a Top-Down Adventure game, it really depends on how large you want to make the world. Traditionally, for a reasonably paced adventure, players will be expecting the following...

- **6-8 Dungeons**

You'll want to make enough underworld content to keep the player hooked. Each dungeon should have puzzles, new items, and a compelling boss battle.

- **Interesting Overworld**

The overworld of a top-down adventure should be fairly large, but not so epic as to get lost. A good rule is to partition sections of your world map, keeping them unreachable without late-game items. This ensures the player stays confined to 'safe areas' until they have the tools and experience to venture into escalating dangers.

- **Townsfolk**

While you don't need to be heavy-handed with the story, you'll need at least one town, or a handful of inhabited outposts, to have people to chat with. These can be quest givers, healers, salespersons, or just NPCs looking to chat about the state of the world. Without these interactions, however, the player will struggle to appreciate the looming threat. Always remind players of the evil that needs to be fought back - sparking the heroic drive to keep adventuring.

- **Exciting Items**

Whether it's weaponry, skills, or new items, players will be looking for interesting upgrades that result in fun new gameplay mechanics. Use these mechanics to craft interesting puzzles and clever battles. A constant flow of new mechanics will keep the player excited for what they'll experience in the next dungeon.

While potentially more involved than a Sidescroller, the Top-Down Adventure is certainly a game genre that a single developer can tackle. Just make sure to prototype those dungeons and get lots of playtesting feedback to make sure your world design is properly guiding new players.

Once you have all the important pieces in place, Top-Down Adventures are one genre that players can't get enough of!

FPS - First Person Shooters

As computers reached the point where real-time 3D graphics were possible, one of the first genres to emerge from this exciting new tech was the First Person Shooter. With games like Wolfenstein and Doom leading the charge, players were thrown into 3D arenas with only their wits, their guns, and a limited supply of ammo.

They may look antiquated now, but these first FPS games treated players to a revolutionary new way to play games. The mind-blowing graphics and the intense, lightning-fast gameplay cemented the genre - and 3D - as a permanent fixture of the Video Game landscape.

Since their humble beginnings on the PC, the basic, single-player FPS has evolved into some interesting subgenres, including the following:

- **Arena Shooters**

Instead of being a story driven, level based experience, some of the most popular shooters now are Multiplayer Arena Shooters. These games drop players into an environment, with various rules (King of the Hill, Capture the Flag, etc) and let players fight it out until one player (or team) stands triumphant. Arena games tend to be over quickly, making the core gameplay loop fast and fun.

- **Survival**

Players are thrown into a deadly situation where the threat of starvation is just as real as any creatures trying to kill you. The Survival sub-genre places the player against the environment, forcing them to Hunt, Gather Resources, and Craft the equipment needed to outlast a more primal set of dangers.

- **Walking Simulator**

The term may have started as an insult, but the concept of a ‘Walking Simulator’ - a FPS that foregoes combat in favor of a rich story - is something that many players have come to enjoy. Often presented as a mystery that you’re slowly unraveling, the player gets to explore a world rich in environmental story-telling.

- **Open-World**

By ditching the level by level structure of the original FPS games, developers found that Open-World environments infused the genre with a thrilling sense of exploration. The open-world FPS comes with some technical hurdles (draw distance concerns, Real-time loading and unloading of assets, etc.), but when done right, the open-world FPS can be a captivating experience of setting out into the unknown.

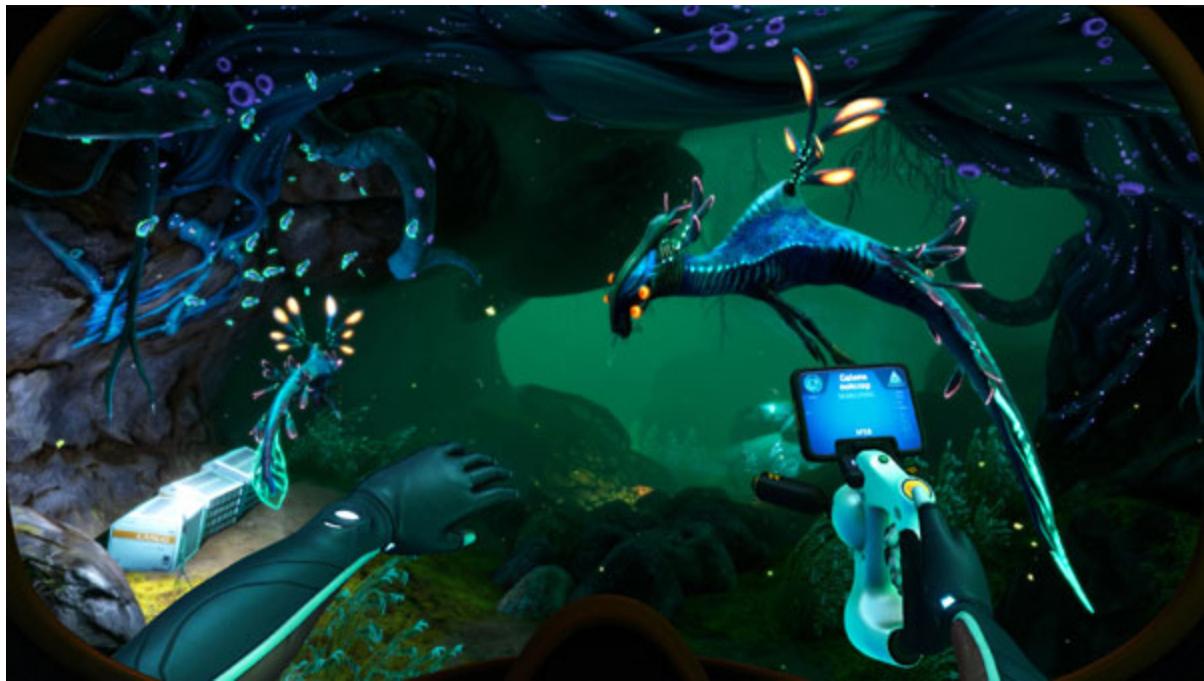


Figure 16.8: *Subnautica*, the Award winning First Person Survival game from Unknown Worlds Entertainment, was developed in Unity.

If you’re looking to make a first-person shooter, the Unity Asset Store has everything you need to get started. There are both free and ‘paid’ project templates that you can make use of, and there are always new 3D assets being added, specifically designed with this genre in mind.

The top free asset is a project created by the Unity Team called **Creator Kit: FPS**, as seen in [figure 16.9](#)

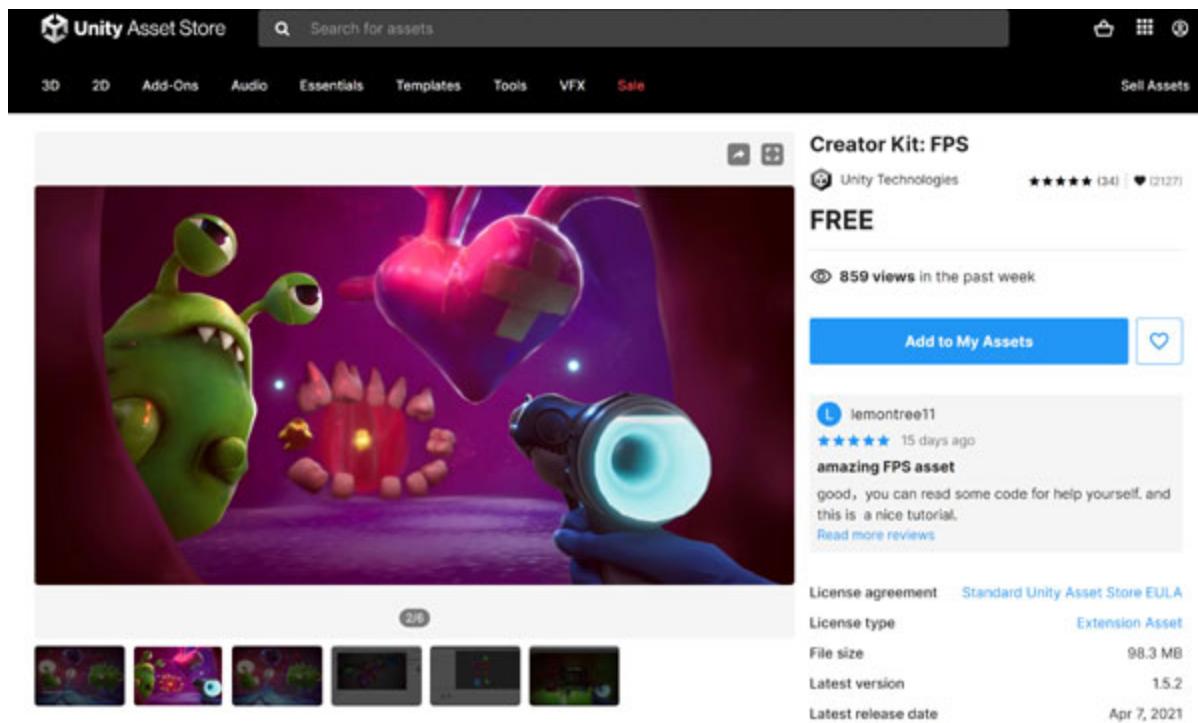


Figure 16.9: The FPS Creator Kit gives you the scripts, prefabs, and basics needed to start building your own FPS.

As we start talking about 3D-specific genres, it's important to note the increase in scope that comes with a 3D game. In general, you can do a lot more, with a lot less, in 2D. On the other hand, 3D graphics require more time and effort to hit the quality bar expected by players.

If you do go solo developing a 3D game (such as an FPS) you'll want to pick an art style that is both visually appealing and allows for the quick creation of assets. This may be a cartoony style, a retro polygon-heavy style, or even a **Hand-Drawn Art Style**, using shaders to pull off an appealing 'brush stroke' look.

With 3D games, one of the easiest traps to fall into is an overwhelming asset creation pipeline. Remember to make test assets early in the process. Use these as markers for how long the rest of your assets will take to create. If there's an asset intensive art style that you really want to use, is there a way to narrow your scope to require fewer assets?

Like with all aspects of a project's scope, everything is a balancing act. You can have *many* simple assets or a *few* complex assets. Designing a game that requires *many complex assets*, however, will lead to frustration and burnout in the long run.

Third-Person Games

Another genre born from 3D technology, Third-Person games place the camera behind your hero as they explore expansive polygonal worlds.

Sometimes the action in these games feels more like a platformer, with the running and jumping challenges of the 2D era re-imagined for the 3rd dimension. Other times, the action will feel more like a shooter, placing a heavy emphasis on weapon loadout, ammo management, and precision aiming.

Figure 16.10: The team behind Yooka Laylee used Unity to create a Third-Person Game overflowing with personality and clever gameplay.

The tools in Unity are perfectly aligned with creating this style of game, with several robust demo projects available showing off the engine's capabilities. If you bring up the Unity Asset Store, then search for **3D Game Kit** ([figure 16.11](#)), you'll find an expansive and impressive project that gives you extra tools and components to start your Third-Person game project.

Figure 16.11: The 3D Game Kit is a massive project that expertly demonstrates how Unity provides all the tools needed for a Third-Person game.

As with most genres, there are a few things players will be expecting when playing this category of game which are as listed below:

- **A Deep Move-Set**

The **Move Set** of your hero are the actions they can perform in the world. Jumping, Running, and Attacking are the obvious moves, but additional actions like Wall Jumps, Backflips, Grappling, Mountain Climbing, Dodge Rolls, and Deflecting Projectiles are great options when designing ways for the player to interact with the world. Players will expect a deep and unique move set when they pick up a new game in the Third-Person genre, so finalize this critical aspect before going too deep into Level Design.

- **Great Level Design**

Of course, when creating a game like this, high-quality Level Design is extremely important. It can't be restated enough, however: you have to respect the time and iteration required to transform *good* ideas into *great* levels. Use White Boxing to block out levels and test broad ideas. Only after your playtesters are having fun is a level ready to be marked as 'complete'.

- **Camera Control**

One common point of frustration in Third-Person games - for players and developers alike - is how to handle the camera in awkward situations. As you craft your levels, you'll likely spend significant time tweaking how the camera works in tight corridors and confined spaces. Wherever possible, use open areas to give your camera the necessary space to situate itself, but don't be surprised if player feedback contains camera-related complaints.

The last thing to mention, just like with FPS games, you'll want to be careful when planning for the creation of art content. Since Third-Person games are 3D in nature, it'll be easy to accidentally overextend yourself, designing a game that requires a ton of assets. Plan your Third-Person game accordingly, and try to pick an art style that doesn't lead to frustration as you struggle to create the required content. Because if there's one game category that can devour assets, it's the Third-Person genre.

RPGs - Role Playing Games

If there is one genre that's favored by storytellers, it's the RPG, or Role Playing Game. Much like the dungeon master in traditional, table-top RPG sessions, game designers love to build worlds, heroes, villains, and adventures for others to enjoy. In this way, It's only natural that the RPG would become the genre of choice for designers looking to tell an epic story.

RPGs are also interesting because they're not necessarily tied to a specific style of gameplay. There are top-down RPGs, first-person RPGs, and even side-scrolling RPGs. Combat can be real time, turn-based, card-based, or even shooter-based, like in the newest Fallout entries. The hero may be a lone adventurer, a party of adventurers, or an entire army defending their kingdom.

As a game designer, you have a lot of creative freedom when building an RPG. Use any of the genres in this chapter as a starting point to start crafting your Role-Playing Game with.

If you're looking for a more traditional, top-down example, however, Unity has an RPG demo project but you can download and make use of. Like with the other genre sample, go to the Unity Asset Store and search for an asset called **Creator Kit: RPG**, as shown in [figure 16.12](#)

Figure 16.12: This creator kit goes over the basics of creating a top-down RPG

The only scope concern comes from the amount of writing you'll do to fill dialog, combat, item description, and general narration required to tell your story. If you're up to this challenge, however, the RPG genre it's a great way to tell a story - and build a world - that the player gets to truly immerse themselves in.

Strategy Games

If you're looking to test the brain power of your audience, a perfect category for your game is the **Strategy Game** genre.

Descending from board games such as GO, Chess, Risk and others, the strategy genre slows down the action in favor of methodical, tactical planning. Often set up as one player against several AI opponents, strategy games tend to be **Turn-Based**, where each player makes their moves in a predetermined order.

Strategy games can be broken down into the following subgenres...

- **RTS**

The real time strategy subgenre forgoes the turn based pacing in favor of something more action packed. Resource gathering, large armies, and big explosions are all part of what makes RTS Games so addicting.

- **4x**

The 4X strategy subgenre stands for explore, expand, exploit, and exterminate. These games have the player exploring a randomly

generated world, expanding a growing empire, trading with other nations, and conquering those nations if necessary.

- **Tower Defense**

A simplified take on the RTS genre, this category of games have you setting up a path of defenses to destroy waves of incoming enemies. Balancing tactics and puzzle solving, the Tower Defense subgenre is a great way to get players thinking strategically across a gauntlet of increasingly difficult levels.

If you're interested in trying your hand at creating a strategy game in Unity, a great place to start is the **Tower Defense Template** asset ([figure 16.13](#)).

Figure 16.13: Get started in the Strategy genre with the Tower Defense Template Asset.

Strategy games are an interesting genre to design for. Instead of testing your skills designing compelling levels and challenges, the job of a Strategy designer lands mostly in the area of Pacing, Balance, and AI. You'll need to create interesting units for the player to command, and clever arenas for players to explore and battle within, but most of the player feedback will be concerning the pacing of the game and whether defeating the AI is too easy, or hard.

The Strategy genre is one where tweaking numbers is a huge part of the process, so keep all your data organized and easy to iterate upon.

Puzzle Games

For thinking games that are more casual (and less explosive) there's the highly successful Puzzle Game genre. On mobile, the puzzle genre is one of the most popular categories of game, with chart toppers like Candy Crush, Cut the Rope, Angry Birds, and Threes! On consoles, games such as Tetris, Puyo Pyuo, and Bust-a-Move include a bit of twitch-centered gameplay to balance the 'brain game' aspects.

Whatever type of puzzle game you prefer, know that it's a great genre to tap into when building your game.

If you want to get started, Unity has a great sample project on the Asset Store called **Creator Kit: Puzzle** ([figure 16.14](#)). While it's more of a

‘Mouse Trap’ style demo, it’ll give you some great ideas for how Unity can be utilized to create a fun, addictive Puzzle game.

Figure 16.14: This Puzzle project will get you thinking about how to create your own clever creation.

One of the best aspects of the Puzzle Genre is that, when considering scope, it’s pretty easy to manage what tasks need to be completed. Your game will require some well polished assets, of course, but in general, all the hard work is in Puzzle Design (making the levels) and Gameplay Development (creating new mechanics to build levels around). Puzzle games are the perfect genre for someone looking to get started as a game developer.

Creative Juxtaposition

Throughout this chapter, we’ve talked about the importance of creating a clever gimmick or unique gameplay to set your game apart from others in a similar genre. This could be an interesting system or mechanic that’s never been seen before, or perhaps incremental advances to existing mechanics.

One way to tackle this ‘clever gimmick’ problem is through the use of **Juxtaposition**. This method of combining two wildly different concepts will produce a unique game that may, at first glance, appear mismatched (and possibly bizarre). Upon closer inspection, however, players will find an experience where contrasting ideas actually come together in a complimentary and exciting way.

With Juxtaposition, you can shock and surprise Gamers with a concept that’s brazenly unique, then impress them with a solid hook and unique gameplay.

Let’s look through some modern examples of Juxtaposition in video game design, starting with the amazing FPS **Superhot**.

When talking about player expectations in the FPS genre, we used adjectives like *intense* and *adrenaline-fueled*. Gamers pick up a first person shooter expecting the heart pumping, non-stop thrills of an action movie - that they’re the star of.

With FPS games, players expect *non-stop* action...but what about a FPS game where *stopping mid-action* was a *key mechanic*.

This is what the Superhot team did with their entry into the First Person Shooter genre. Here is a game where you have explosive weaponry and enemies changing you from all sides, as expected. Unlike other FPS games, however, you have a brilliant gameplay mechanic where *time only moves if you move*. When you're stopped, everything and everyone around you stops. If you move slowly, enemies and bullets will also move at a snail's pace.

Figure 16.15: Superhot - a FPS where time only moves when you do.

This Juxtaposition between the expected high-intensity gameplay of other FPS's, and the slower, deliberate, tactical gameplay of Superhot, gave players something new and unique to master.

The other game we should talk about came from a partnership between Nintendo and UbiSoft. This game - *Mario + Rabbids: Kingdom Battle* - was a juxtapositional masterpiece. Not only was this the first time the Mario characters and Rabbid characters came together, they did so in a completely unexpected genre: a Turn-Based Strategy game.

Figure 16.16: *Mario + Rabbids: Kingdom Battle* was a delightful, unexpected surprise for many gamers.

Whenever possible, you should try to surprise and delight gamers with something they haven't seen before. By understanding the basics of a genre, you can use Juxtaposition to play with the core mechanics in interesting ways!

Player Expectations

While it's important to have unique gameplay, ideas, and surprises for the player, it's equally important to get the *basics* right. Being mindful of **Player Expectations** - especially in an oversaturated industry - will be an important part of hooking players in that critical *First 10 Minutes*.

There are several learning opportunities that'll help you recognise what players are looking for. Only after you *understand* expectations can you then *exceed* them!

- **Read Reviews**

If your game fits into a pre-existing genre, chances are a digital marketplace will list **Similar Games** you can learn from. Bring up professional and user reviews to read and absorb. You should quickly determine the parts of a game that are loved, and which parts came up short. Use reviews across multiple games to determine what players will be expecting in a given genre.

- **Play and Learn**

There's no better way to dig into a genre than to play a *ton* of games. Big AAA games, small Indie games, and everything in between. Try to approach each title as a player would, and if you feel something is/isn't working, take note of it. Use playtime as an opportunity to research a genre, and you'll quickly build a deep well of gameplay ideas to use in your own designs.

- **Imitation and Iteration**

When tackling the blank page of a new design, it can be easy to get overwhelmed trying to make every idea new and unique. When struggling to craft a design solution, you'll be surprised how many times the answer is to simply 'do what worked before'. By taking a solution from a previous successful game, then iterating on it, you can be sure your design choices lineup with Player Expectations.

The Imitation Game: As a creative thinker - someone looking to make a game that's new, clever, and unique - the idea of using ideas from other games may sound unfulfilling. Know that the industry, as a whole, is built on the iteration of great ideas from many sources. If a game is released, and the team has developed a solution for a common problem (camera controls, morality systems, quest and mission management, etc), you'll see the entire industry adopt and iterate on that solution. As long as you're not stealing and recreating games in their entirety (which is absolutely frowned upon in the industry) your adoption of ideas will allow you to match player expectations. And if your implementation is unique enough, perhaps give other studios a fresh take to consider!

By polishing the basics and meeting Player Expectations, your game will have a solid foundation, ready for those crazy, unique ideas that will set

your game apart.

Conclusion

Designing your own game can be immensely fun, incredibly satisfying, yet - at times - feel insanely difficult. While we'll continue to talk about ways to reduce these difficulties, simply identifying and understanding your *genre* will help to narrow focus and give your design a strong initial foundation. Instead of creating a whole new experience, you're simply iterating on the amazing games that came before it.

With a better understanding of how Unity helps you to master these genres, it's time to take a look at the various platforms you'll be creating games for. Whether you're targeting PC, Mac, Mobile, Consoles, or AR/VR, Unity makes it easy to take one project and deliver it to gamers on any system.

Questions

1. In what ways can an over-sscoped design be harmful? Even if the project scope is reduced in the middle of development, what harm can it still do to the project or team?
2. Why is it important to understand the genre you're working in?
3. How can you determine what current player expectations are for games in a given genre?
4. How can Juxtaposition be used to create an interesting game design?
5. What are some of the ways it can be better to use a paid asset over a free one?
6. The time and energy spent on making assets can bring a project's momentum to a halt. How can the selected Art Style change the speed that assets are created?

Key Terms

- **Genre:** A style, or classification, of gameplay. Genres are collections of specific features, mechanics, and interactions that come together in a compelling way.

- **Scope:** The art, coding, design, testing, and marketing work required for your game. Designing with genre in mind will help narrow the focus, helping to keep scope from getting too unwieldy (and overwhelming).
- **Permadeath:** A gameplay rule where, if the player dies once, they have to start the entire game over. This is often set as a difficulty option before the game starts.
- **Turn-Based Gameplay:** Any game where moves are made one player at a time, giving each participant time to carefully study the game board and determine the best possible move. Often used in genres where strategy and tactics are important.
- **Player Expectations:** todo
- **Hand-Drawn Art Style:** A 3D art style that uses shaders and post-process effects to make your assets look hand-drawn, resembling a painting, comic strip, concept art, or some other pleasant aesthetic. Using shaders in this way will often take the burden off of needing to create AAA-quality assets, since any object will get the proper ‘look’ by simply using the shader.
- **Move Set:** All the moves your hero can perform in a game. Jumping, Crouching, Wall Jumps, Backflips, Dodge Rolls - there are countless ways to make the player feel like an agile superhero in your game. Make sure all these moves feel great before moving onto level design (perhaps in a Movement Sandbox scene).

CHAPTER 17

Platforms and Publishing

We live in a time where video games are *everywhere*.

Awesome new games are released daily across mobile, console, and PC/Mac. It's not just traditional systems, - you'll find gaming in abundance in places you wouldn't expect.

Going to the movies? Before it starts, there may be an interactive game on the big screen to play vs. the rest of the audience. In the passenger seat of a car? Large-screen infotainment systems have several games to enjoy. Learning something new in school? Educational web-based-games teach new concepts in an exciting, approachable, and familiar format.

Video games are everywhere - which means you have tons of opportunities to get your creation in front of players.

The problem with such a wide variety of platforms is that each system has its own unique way of performing basic tasks. Handling input, displaying graphics to the screen, File IO, and even writing code are all core development concepts that can have wildly different implementations. You're forced to spend time learning 'how to do things', when that effort would be better spent *making an awesome game*.

This is where Unity excels. Allowing you to create one project and deploy it across a wide variety of platforms. No more re-inventing the wheel when porting a game - simply make some simple code changes, select your platform of choice, and Unity handles the rest!

Structure

In this chapter we will discuss the following topics:

- Your Platform of Choice
- PC / Mac / Linux
- Mobile Games

- Console Development
- WebGL
- AR / VR
- Publishing
- Publishers

Objectives

In this chapter, you'll be learning all about the different platforms that you can target in Unity. You'll learn about the nuances of these platforms, with their various benefits and detriments. We'll also get into the process of taking your final game and publishing it, whether you're teaming up with an official publisher, or going the 'self-publishing' route.

By the end, you'll have a greater understanding of the types of games that work best on given systems, and hopefully be closer to identifying your preferred platform when designing your own game.

Your Platform of Choice

A **Platform** is the combination of hardware and software that will run your game. You can always count on a platform to consist of 3 major systems: user input, display output, and CPU. These three systems are the backbone of the interaction cycle: the display shows the player something, input allows the player to react, and the CPU facilitates communication between the two.

When choosing a target platform, the *real* excitement comes from the *unique* aspects of various platforms. Clever forms of input, immersive forms of output, and even the power of the CPU can be leveraged to create wildly different games.

Let's say you want to make a very casual game that people can play at any time. In this situation, you'll want to target a Mobile platform (iPhone, iPad, or Android devices). Players can enjoy your game by simply taking out their phone, encouraging gameplay in short, fun bursts and helping to increase retention. If a deep, immersive environment is your main focus, there's no better platform than VR, where powerful headsets surround the player with the sights and sounds of your game world.

PC and Mac gamers have the benefit of Cutting-edge Hardware. Owners of a Nintendo console are often looking for amazing family-friendly content. And web-based games can be used by corporations to give people a fun mini-game to enjoy when visiting their website.

There are many things to consider when picking your platform of choice, so let's break down the specifics of the various gaming systems.

PC / Mac / Linux

The easiest platforms to develop for are, by far, the PC, Mac, and Linux systems. This is because the hardware you're targeting is the same as the hardware you're developing on. No special wires or dev tools required - the development platform IS the target platform.

This sounds simple enough, but the benefits are wide reaching.

Quick iteration is a *must* in game design, and it doesn't get any quicker than being able to test directly on the same OS as the one you're targeting. Making changes and compiling a new build for other platforms can take 5x - 10x longer than the quick iteration times here.

Debugging is also significantly easier, with an easy integration with the Visual Studio debugger, allowing you to place breakpoints and do a deeper dive into what could be causing issues when they arise.

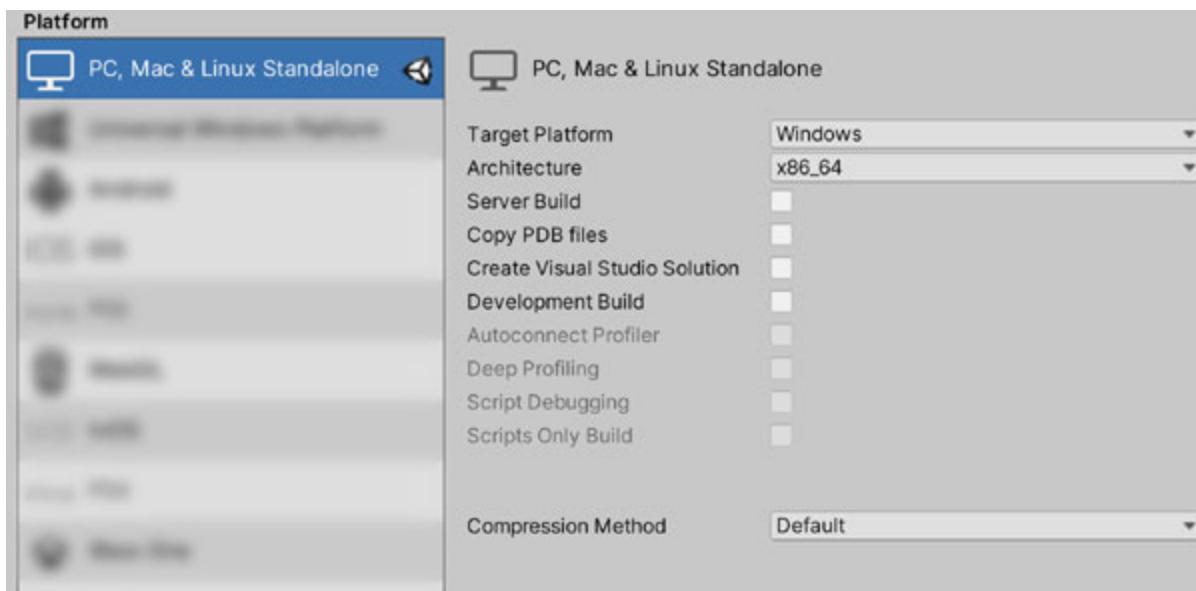


Figure 17.1: The Build options available when creating a PC, Mac, or Linux Standalone build.

As with all platforms, creating a standalone build for these PC / Mac / or Linux will give you access to a slightly different set of platform features. Let's investigate the specifics...

- **Input**

- Keyboard
- Mouse
- Controllers (Optional)

- **Output**

- Monitors of various resolutions and ratios. Designing your UI around the 1920x 1080 resolution *should* ensure a usable layout across all monitors.
- Some users may have a multiple monitor setup.

- **Pros**

- Easy to test and debug builds.
- Wide adoption of target hardware (lots of people own PCs and Macs).
- Steam marketplace makes it easy to sell games to players, and it comes with a powerful API (Workshop, Achievements, Multiplayer, etc).

- **Cons**

- Can be tricky developing for a wide variety of possible setups (CPUs, memory, GPUs, etc).
- Can't count on a player having a controller, if your design requires one.
- The PC gaming market is very saturated, making it hard to get noticed.

- **Other Notes**

- Since computers can be upgraded and tweaked by the user, you'll need to support a wide variety of specs. Make use of an options screen to toggle graphics settings, letting the player disable /

enable features to get the game running at the best possible framerate.

Mobile Games

The mobile game market, as we know it today, is a relatively new phenomenon. Smartphones capable of the same graphical fidelity as consoles and PCs have only been around for the last five years. Being able to take a 3D game - designed for a more powerful system - and simply deploy it to your phone is a pretty remarkable feat. Many larger studios have tapped into this flexibility, like the team at Blizzard, who used Unity to produce the amazingly addictive Hearthstone across multiple platforms ([figure 17.2](#))



*Figure 17.2: Blizzard Entertainment used Unity to create the legendary **Hearthstone**, which was launched across multiple platforms.*

To develop for mobile, you'll first need to pick your platform of choice. As of today there are two major competitors in the mobile space, the iPhone and Android series of devices.

Each of these device families have their own development ecosystem: the hardware required to develop on (a laptop or PC), the hardware required to

deploy onto (a phone or tablet), and an online, cloud-based portal for managing titles and developer tools.

If you prefer the Apple family of devices, which include the iPhone, iPad, and Apple Watch, then you'll need to grab your Mac and set up an account at developer.apple.com. This is the portal you'll be using for uploading and managing apps made for these devices.

Developing for iPhone will require a Mac based computer, an iPhone or iPad device to deploy and test on, as well as developer credentials and an Apple developer account, which will cost you approximately \$120 a year. If there was an argument to be made *against* developing for the iPhone/iPad family of products, it would probably be the upfront cost, especially if you lack the necessary hardware. It can quickly get expensive for a humble developer.

Once you have everything you need, however, including app certificates and the proper dev tools, developing for Apple devices is a clean and straightforward process.

To access the most up-to-date information on making iPhone builds and deploying to Apple hardware ([figure 17.3](#)), visit docs.unity3d.com/Manual/iphone-GettingStarted.html

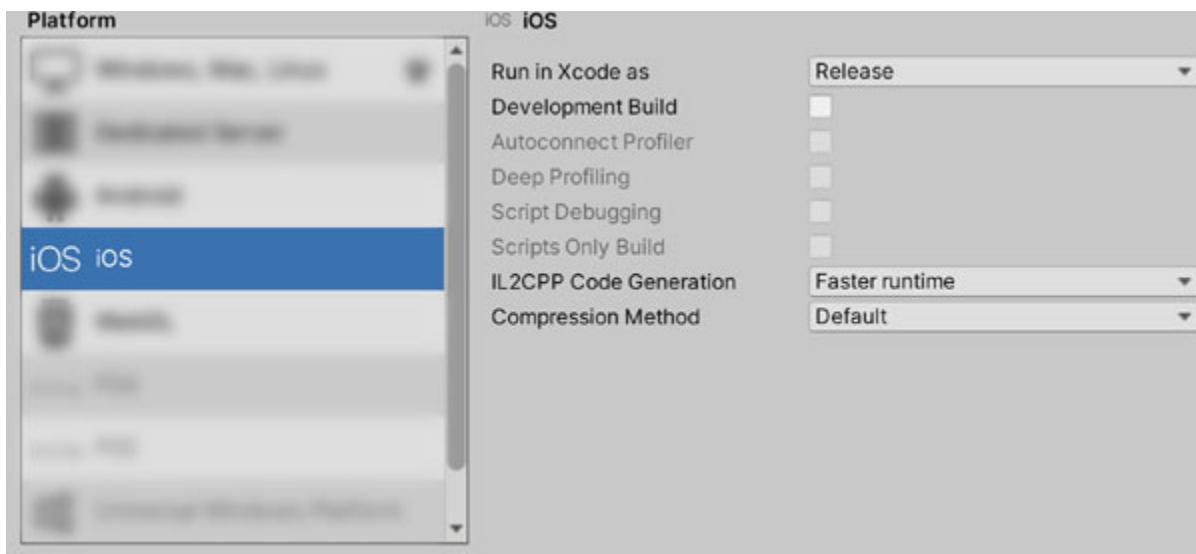


Figure 17.3: Your build options when making an iPhone / iPad build of your game.

Android devices give you a less expensive alternative to mobile development, since you can pick up a secondhand android device relatively

cheaply. And you can develop on either PC or Mac, so there's no specific computer requirements for deploying.

Because the android line of phones is based on an OS, and that OS has been used across hundreds of different devices, the challenge with developing for Android is going to be one of compatibility. **Compatibility** is the process of making sure your game works on the widest range of hardware options. Compatibility is a concern on any platform where there is a wide range of both high end and low end devices that can play your game.

Since it's impossible to test on *every* device, it'll be up to you to either hire a testing facility that can check compatibility across devices, or you'll need to get your hands on some baseline hardware to test. This requires getting one or two high-end devices, and a handful of low-end devices (and possibly some mid range ones) to make sure your game runs properly across a wide spectrum of hardware.

Deploying to an Android device is possible directly from the Unity editor ([figure 17.4](#)), so if you're interested in getting started on Android development, visit docs.unity3d.com/Manual/android-BuildProcess.html for the most up-to-date setup instructions.

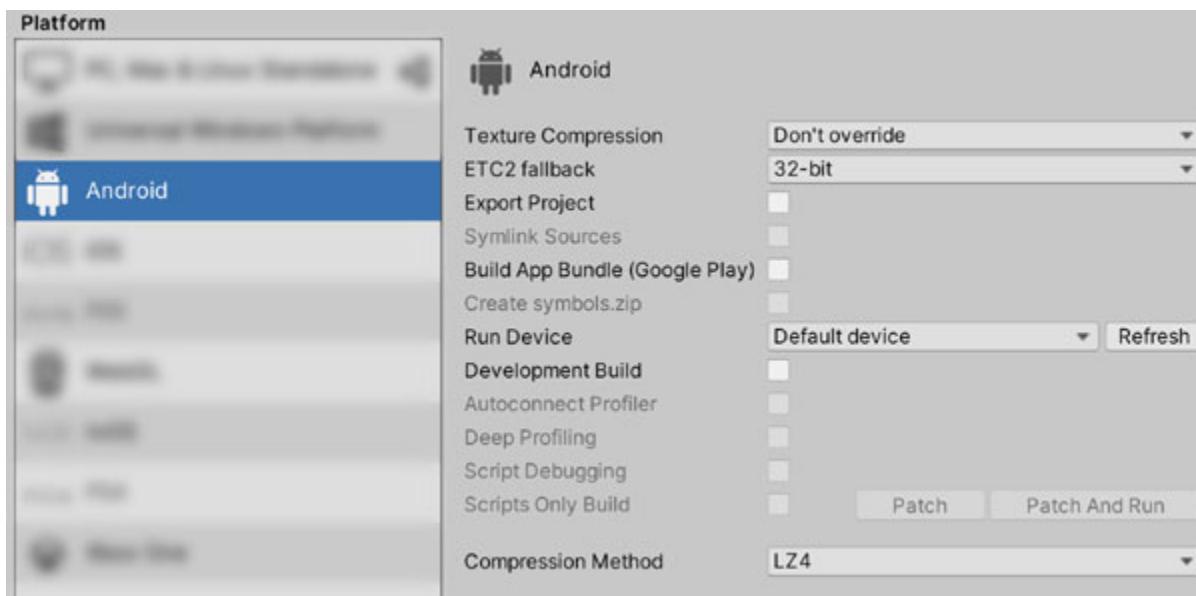


Figure 17.4: Building and deploying a build to an Android device.

When it comes to designing for iPhone, iPad, or Android devices, there are some unique hardware features that you should be mindful of which are as listed below:

- **Input**
 - Touch Screen
 - Cameras
 - Microphone
 - Bluetooth Controllers (Optional)
 - Pen (On Some Models)
- **Output**
 - Smaller screens of various resolutions and ratios (ranging from 1280x720 up to 2778x1284)
- **Pros**
 - Almost everybody has a phone, so the possible user base is gigantic.
 - The app management tools provided are great for putting out updates, managing sales, etc.
- **Cons**
 - Mobile app markets are extremely saturated. Takes hard work, strong marketing, and a bit of luck to get noticed.
 - Most players are looking for **F2P** (Free to Play) games. Some game designs don't work well with this monetization model.
 - Limited input options, with no tactile feedback.
 - Designing an interface where you can tap the screen without obfuscating is challenging.
 - Designing UI screens that resize nicely across all resolutions and ratios can be complicated.
 - Testing on a device can make debugging difficult.
- **Other Notes**
 - Mobile devices are considerably more powerful nowadays, but since you'll still want to work on a wide range of devices, an options screen (with graphical quality toggles) is recommended.

Console Development

If you were a kid that grew up playing video games on your couch, then *this* is probably the section that got you excited about Unity. With Unity, you can make one project and deploy it to the console of your choice. Whether that is the Xbox, PlayStation, or Nintendo Switch, Unity gives you everything you need to make your *console debut*.

But first, a bit of gaming history.

For the last 30 years, making a game on a console was fairly difficult, and not just from a development standpoint. High quality standards and expensive hardware requirements set a high barrier for entry for anyone looking to create a game. Factor in the manufacturing cost of physical cartridges, and the thought of a small team making a console game would've been unthinkable. And for a solo developer, *impossible*.

Thanks to Unity, as well as the introduction of digital downloads, small development teams (and even solo developers) have the opportunity to create games and release them on their favorite console.

And while Unity makes this a straightforward process, there are still some caveats when looking to develop games for the XBox, Playstation, or Switch.

The first, and most important to remember, is that you'll still need to get a Dev Kit that you can deploy and test on. A **Dev Kit** is a special version of a console made specifically for developing on. This includes extra tools for debugging and accessories for transferring builds from your computer to the console. The price of these development kits has come down considerably: what used to cost tens of thousands of dollars will now only cost *hundreds*. That's still a steep hardware cost for some developers, but it's certainly more reasonable than before.

Second, know that part of the application process is to pitch the game you're making. You'll need to prove that you have a high-quality game in the works (not just vaporware) and can be trusted within the private development forums.

To help in this process, you may want to first release a title on a less restrictive platform (PC or Mobile) to both prove yourself as a serious developer, and help get the funds needed to purchase a Dev Kit.

- **Input**
 - Controllers with up to 12 buttons, 2 analog sticks, and rumble support.
 - Several controllers support motion-controls
 - Bluetooth Keyboard and Mouse (optional)
- **Output**
 - Games are displayed on the players television set, traditionally at a 16:9 aspect ratio.
 - The Switch has an ‘Undocked’ mode, where a 1280x720 mobile screen is used.
- **Pros**
 - Fewer games are released on consoles, so it’s easier for your game to be seen in the store.
 - Knowing that the user has a controller can make it easier to plan UI and input systems.
 - Traditionally, only one title is running at a time on a console, so you have the entire muscle of the platform available for your game.
- **Cons**
 - Higher barrier to entry than PC, Mac, and Mobile.
 - Some consoles are less powerful than others, so it can be difficult to get graphics looking the same across all three platforms (if a multi-platform release is part of your plan).
 - Before you release on a console, there is a substantial checklist of testing requirements known as the **TRL** (Technical Requirements List). You’ll have performed and verified all the items in the TRL before you can upload your game. This can take weeks, or sometimes months, to implement these requirements. Factor this into your schedule when developing a console game.

WebGL

Online gaming may not have the draw that it had 15 years ago, but there are still plenty of sites looking for gaming content that can be played through your browser.

When exporting a project from Unity to be played on the web, developers can target the WebGL platform. This will package the game in a lightweight format that can be uploaded to a server and played with a small bit of HTML code. Like with other platform options, WebGL can be selected in the Build window, as shown in [figure 17.5](#)

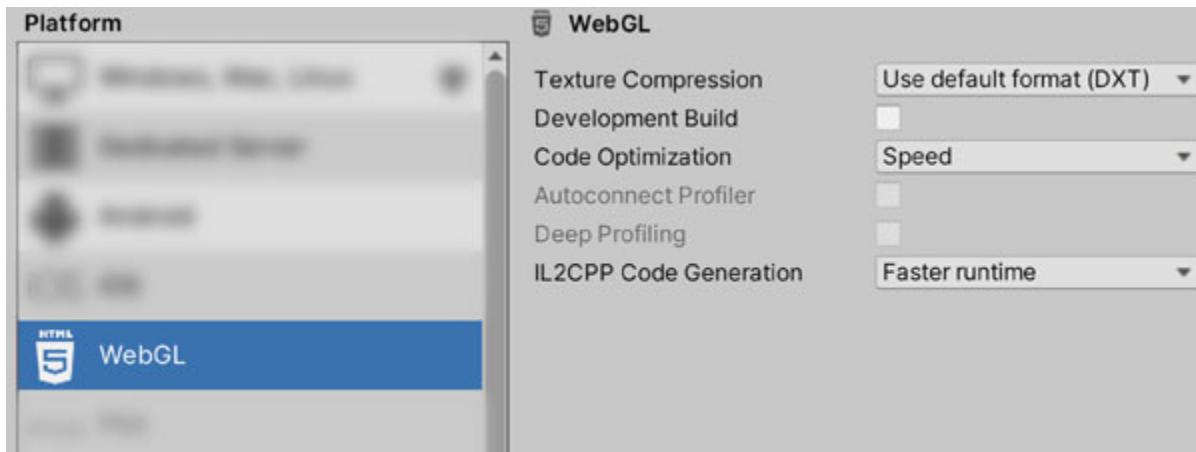


Figure 17.5: Unity's WebGL Build Settings.

While there are considerations to make here, they're primarily concerning size and performance, since WebGL games need to be downloaded and run on a single-threaded process. This list below covers some of these considerations...

- **Input and Output**

- Same as the device running the Browser that the WebGL game is running in.

- **Pros**

- A quick way to get a game into the hands of anyone. Once the game has been uploaded to a site, simply send them a link.
 - Easy way to target PC, Mac, and any platform that can run a WebGL package in its browser.

- **Cons**

- Single threaded.
- Limited Monetization options.
- Larger projects can have long loading times (especially if running off a slow server).

- **Other Notes**

- If you’re looking to sell your game, WebGL is probably not the main platform you want to target. However, if you can use it to make a demo, it can be a great marketing tool to give players a sample of your game.

VR/AR

One of the newest platforms to reach mass adoption are the various headsets used for VR (Virtual Reality) gaming.

The amount of progress that’s been made in this area has been astounding. There was a time just a few years ago we’re any serious VR headset would have to be tethered to a powerful gaming PC, with motion sensing units on two corners of the room you were playing in. It took a lot of time to set up, and early hardware was prone to miss calibrations and firmware issues. But for all the headaches that could come out of a VR session, the technology was undeniably incredible. Being able to move your head in 360° - as well as your hands and body - and actually feel like you’re in a 3D world was unlike anything most players had ever experienced.

Fast forward to today, where cameras on the headset can perform inside-out tracking, allowing gamers to play in 3-D space without needing physical sensors to track their position. Advances in graphical chipsets also mean that certain VR headsets no longer have to be tethered to a large, expensive PC. Standalone VR has brought the technology to the masses, and games such as Beat Saber and Job Simulator ([figure 17.6](#)) have used Unity to immerse and delight gamers in a way that can’t just be described - it has to be *experienced*.

Because Unity was at the forefront of this VR/AR revolution, game developers interested in the technology will reap the rewards of a well integrated set of VR tools available to anyone using Unity. If you have a

Meta Quest headset, all you need is a handful of downloads and project templates to get started making your virtual creation.



Figure 17.6: The award winning VR game *Job Simulator*, by Owlchemy labs, was developed using Unity.

Some considerations to make when building for VR include...

- **Input**
 - Motion-Sensing Controllers with buttons and analog sticks
 - Hand-Tracking with gesture recognition
 - Eye-Tracking (Experimental)
 - Voice Commands (Experimental)
- **Output**
 - Two Screens, one for each eye
 - 3D Sound
- **Pros**
 - Complete and instant player immersion. Users are placed in a 3D environment they can explore and interact with.

- Movement and interaction comes naturally. For instance, in most games, crouching is a button. Here, you simply *crouch*. Very intuitive.

- **Cons**

- Some headsets are uncomfortable for players who need glasses.
- Headsets can get heavy, and hot, after prolonged play sessions.
- Some experiences can be too real (and *too scary*) for certain players.
- To get the best graphical fidelity, you still need a powerhouse gaming PC to connect to.

Publishing your Game

Since we're talking about the various platforms that you can release a game on, it's important to mention what's involved in actually Publishing your game to a digital marketplace. **Publishing** is the act of getting your completed product into the hands of players. With digital storefronts, this means organizing and uploading the screenshots, text, trailers, and the playable build.

Different platforms have different ways of handling this step, but as listed above, there are several assets that every marketplace will require as mentioned below

- **Executable Package**

The final build that gets exported out of Unity is one of the most important things you will be uploading when it's time to publish. The executable that you upload will have different names based on the platform: bundle, package, executable, app. But they're all talking about the final build of your game - a version that has been thoroughly tested and is ready for public consumption.

- **App Icon**

This is the image that is displayed to players representing your game. Do not shrug this off or take the creation of this asset lightly. This artwork is the first thing that players will see when they go to start up

your game. It needs to reflect all the personality of the game you created.

- **Game Description**

Since almost every digital store has a spot for a game description, you'll need to develop several paragraphs that describe your game in an exciting way. While screenshots and trailers are better tools for hooking a player, once they're interested in your game, they're bound to want to learn more. Inevitably, they'll head to the description to get a better understanding of your game. Make sure this includes all the unique gameplay and clever hooks. If you do everything right, it'll leave them excited to jump into your game world.

- **Screenshots**

Because video games are a visual medium, it's important to have around six well-crafted screenshots from your game to entice players. Study the rules of good composition, and put in cheat keys to allow UI elements to be toggled to present your game in the best way possible. Start with 20 screenshots that you like, show them to friends, family, and other gamers to see which images resonate the best.

- **Trailer**

Nowadays, most marketplaces will let you upload a video trailer to set as the initial media shown to players on the store. You *can* skip this part, but it's highly recommended that you *don't*. The trailer is a flashy way to pull players into your game world with quick cuts of well-crafted gameplay shots, some pre-rendered animations, and amazing music and sound effects. Overlay text to describe the high-level, unique aspects of your gameplay. You have 30 to 60 seconds to excite the player, so make sure you keep this video full of interesting action and visuals. As you make your game, be thinking about the trailer and what scenes, gameplay elements, animations, and effects you want to highlight. One minute of amazing footage is all you need, so make sure it's perfect!

Teaming Up with Publishers

Creating a great game is only part of the work required to launch a successful title. As we discussed about publishing, you hopefully got a

sense for the other player-facing content that needs to be created and managed. Screenshots, trailers, and marketing text will be your first interaction with potential players. Getting these things wrong – especially in today’s crowded marketplace - can ruin your chances before your game has a chance.

All of these tasks, while required for publishing, fall under the broader category of *marketing*. **Marketing** is all the publicity done to get players excited about your game. Without the proper amount of **Hype** (player excitement), it’ll be hard to get traction selling your game.

Because marketing is so hard, and requires a lot of you time and effort to do right, you may want to consider teaming up with a publisher. Publishers are large, external partners that will take your game to the next level - helping to craft the best marketing message that will get people hyped about your creation. They will often spearhead and pay for all the marketing activities, and because of their size, will have better connections to the industry. They can get your game reviewed by large sites, get advertisements in place, and build up a large social media following. With publishers handling all the non-game creation stuff, it allows you to focus on what you do best – making an awesome game.

There is one drawback to this set up: publishers require a percentage of your games profits. This is understandable, since advertisement and marketing campaigns are costly. It’s also understandable that you’d be cautious about giving up too much after working so hard.

Consider this, however: if a Publisher can double your profits, then aren’t they worth 40-50%?

Just be sure to read contracts, check to see if you’re giving up IP rights, and make sure that the percentages aren’t too lopsided. There are several Indie-friendly publishers out there that, while they can’t represent tons of games, the games that they *do* publish get all the marketing love and affection you would expect. It never hurts to have a talented group of professions rooting for your success. And if they can double, or triple, your sales, then certainly they’re worth 40-50%.

If you’re enjoying the game making process, but get overwhelmed with marketing and publishing tasks, you may want to team up with a fair and qualified Publisher.

Conclusion

It's easy to take the multi-platform capabilities of Unity for granted. When something works this seamlessly, it hides the complex and difficult work that is being automated.

Just 10 years ago, if you wanted to make a game for mobile, console, or for the web, you would have to learn entirely different coding languages, new input and output APIs, and new file handling systems. The fact that Unity manages (and obfuscates) all the particulars of these various platforms is a testament to the talent of the Unity team.

Because of this amazing tool, any game developer with enough time, energy (and possibly cash) can create and release games for the platform of their choice.

You now understand the available platforms to develop for and tools to develop with. You've learned about the genres players have come to love, and design concepts to put your own clever spin on those genres.

Our final exercise has us taking a serious look at the *game design process* as a whole. Finally - it's time to tap into your creative side as you *design your own game*.

Questions

1. What are the three major systems that you can count on all platforms to have?
2. What are the benefits of targeting a Mobile device for your game?
3. What kind of games genres work well with the Free to Play monetization model?
4. What are the benefits of targeting a VR device?
5. Name a platform you could target if your design requires a controller?
6. What are the benefits of teaming up with a Publisher? What are the drawbacks?
7. What's more effective for building hype - a written description, screenshots, or a trailer?

Key Terms

- **Platform:** A system that you can play your game on, consisting of input systems, output systems, and a CPU. Different platforms will often offer unique forms of input and output, making different platforms ideal for different types of gameplay.
- **Compatibility:** Is the process of making sure your game runs on high-end hardware, low-end hardware, and everything in between. The more hardware configurations you're compatible with, the more players can enjoy your game.
- **PC / Mac / Linux Builds:** Standalone builds can target Windows, Mac, and Linux OS'es. Great for more involved games that require time, attention (and perhaps a keyboard).
- **Mobile:** Mobile builds can target iPhone, iPad, Apple Watch, and Android devices. Great for games that should be played on the go.
- **F2P:** ‘Free to Play’ games cost nothing to download, but will have a store where you can make purchases. Most mobile games use this method to get as many users as possible, but note that F2P doesn’t fit nicely with more traditional genres.
- **Consoles:** The XBox, Playstation, and Nintendo Switch game systems. You can target these platforms if you need the player to have access to a controller.
- **Dev Kit:** A dev-only version of a console. You have to get approved to purchase one, but the price of these devices has dropped considerably in the last two generations.
- **WebGL:** The browser-based platform that you can deploy Unity projects to.
- **VR/AR:** Virtual Reality and Augmented Reality. Platforms that use headsets to put the player in 3D worlds, or display holograms in front of their view.
- **Publishers:** Organizations that can help you market and publish your game.
- **Marketing:** The process of creating a message about your game that excites players. This can be through Screenshots, text, trailers, or any other number of efforts to engage potential players and get the word

out about your game. If there's one aspect of game development that gets overlooked by small teams, it's the marketing effort required for success.

- **Hype:** The player excitement built from your marketing efforts. You can build hype by releasing videos, building a social media community, Johnny podcasts, website editorials, or through good old fashion advertisements.

CHAPTER 18

From Concept to Completion

All great games come from a unique creative vision. Unity gives you everything you need to bring that vision into reality, but clearly envisioning a game can be as difficult as implementing it.

Up until this point, almost everything we've built has been based on the topic of a given chapter. The examples you've implemented haven't given you much opportunity to be creative. You've learned the tools by using the tools. Now it's time to learn successful game design by *designing your own game*.

The games industry is saturated with new titles being released daily, so your core competitive advantage is a unique vision and personal creativity.

In this final chapter, you'll be learning how to hone your creative vision, mastering game design from *Concept to Completion*.

Structure

In this chapter we will discuss the following topics:

- Preproduction
- Design docs
- Paper Prototypes
- Concept art
- Mock ups
- Project management
- Play testing
- Analytics
- Prioritizing feedback
- Milestones, Momentum, and Morale
- Early access builds

- Marketing
- The Gold Master...and Beyond!
- Mastering Game Design

Objectives

In this chapter, we will be turning our attention from learning development tools to learning about game design and project management.

By the end of this final chapter, you'll have a firm understanding and appreciation of the pre-production stage, where brainstorming, creativity, and detailed documentation will build a solid foundation for the rest of your project.

You'll also be learning the tools and process of improving your game based on Player Feedback. By mastering the cycle of design, development, feedback and iteration, your project will be set for success.

Preproduction

“Anything worth *doing* is worth doing *well*.”

This phrase holds true with any task, and since Video Game development involves *many* tasks, it's especially appropriate here. If you have the passion, drive, and vision to spend months (if not years) on your own video game, then doing it ‘well’ should be one of our top goals.

Since we'll be focusing on *quality* over *speed*, our first step is to dive into the **Preproduction** process. Preproduction is the phase of development that has you describing your game using words, pictures, sound...whatever it takes to get the vision out of your head and into a format that others can decypher.

A ‘Team Leader’ Note: Clear articulation of your vision is especially important when working on a team. There will be many opportunities for a shallow design to be misinterpreted, so be as thorough as possible when describing gameplay, story, UI, and other aspects of your game. If you’re a solo developer, having a deep and detailed outline is still vital for organizing thoughts and separating the ‘ideas’ phase of

development (“Preproduction”) from the ‘implement’ phase (“Production”).

The pre-production phase can last as long as you want. It allows you to go wild imagining the game you want to make. Draw, write, daydream. Close your eyes and imagine playing the game that you’re creating. What genre of game is it? Is there a hero? If so, what does the hero look like? What does the hero sound like, and how do they move? What’s the clever gameplay hook that will set your game apart?

The more of these questions you can answer, the more concrete your vision. It’s also important that Preproduction is done away from your coding tool of choice. To the best of your ability, *no code should be written at this stage*, leaving your decision making to be more creative, and less analytic. Remember, the stronger your vision, and the more time spent in the Preproduction phase, the better your game will be.

Now that you understand - can hopefully *appreciate* - the importance of preproduction, let’s go over the key activities related to this critical stage of development.

Design Docs

One of the most important activities in the Preproduction process is the creation of **Design Docs**. Design Documentation contains your game design vision in written form, laying out story, gameplay, world landmarks, enemy stat spreadsheets - anything and everything associated with your game.

Before sitting down to start documenting your design, you’ll first want to pick the best tool for the job. Some designers prefer to have a notebook on hand at all times, letting them jot down ideas when inspiration strikes. Many studios, especially with multi-person teams, will make use of cloud-based documentation tools like Google Docs (as shown in [figure 18.1](#)). By creating your design and storing it in the cloud, you have instant access to it at any time. This is a useful way to collaborate, iterate, and ensure the details of a project are always a few clicks away.

Figure 18.1: Many studios use the Google Drive apps to write and store design docs. Since it’s a cloud-based solution, your game design can be viewed, edited and discussed anytime, anywhere.

There are many aspects of a game's design that you should cover in your documentation. These include the following:

- **Gameplay Overview:** A brief 1-2 page overview of your game is a great way to solidify your high-level goals. What is the genre? Who is the hero? How many hours of gameplay are you expecting? Include an **Elevator Pitch** - a focused, 3-5 sentence explanation of your game that instantly excites the listener. The overview doc should give incoming team members a good idea what kind of game is being made.
- **Mechanics:** Each major gameplay mechanic should have their own bit of documentation. This should explain the intention behind the mechanic and different ways that the mechanic can be used in the game design process. You can imagine a power-up having a document categorizing different ways it can be used in level design. Or perhaps your game has several hero classes with different abilities. Writing a doc for each class would be a perfect way to organize your design into its core mechanics.
- **World Building:** If your game has an important back story, it's vital to get this written down so all team members have access to it. Understanding the details of the game world will help writers create better dialogue, artists create unique assets, and designers create clever, thematic gameplay elements. Quality world building starts with a deep and detailed World Design doc.
- **Characters:** The characters of your game are often the heart and soul of your story. You cannot be too thorough in your descriptions, listing both gameplay specific abilities and personality quirks that have no effect on gameplay. The more character (and characters) you can infuse into your game, the deeper the connection you'll make with players.
- **Stats and Numbers:** Because so much of a game's pacing and feel is determined by the numbers and values used, it's important to have a place where these numbers are listed and explained. Many spreadsheets will even allow you to write code to export columns of data as XML, which ensures the game data is always synced with the design documentation.

- **Implementation:** Sometimes there are gameplay systems that are so complex that their implementation needs to be documented early in the process. Defining classes, file formats, data pipelines, and system flowcharts will ensure your game is properly architected. Implementation docs can also be written as systems are hooked up, giving incoming developers insight into how, and why, complex systems were implemented.
- **Style Guide:** To make sure all the artists are producing cohesive assets, you may want to create a style guide that outlines rules for the creation of UI and in-game art. The style guide will often include concept art and inspirational pieces. The larger your art team, the more important it is to have a guide the reference. Visual continuity is one of the most important factors of a professional-looking game.
- **Marketing Ideas:** As you're designing your game, it's only natural to start having ideas for the various ways you want to build hype. Have a marketing document ready to fill with ideas that come up during development. While advertisement and marketing is done closer to release, it's never too early to start brainstorming ideas.

A design doc is also a **Living Document**, meaning that the specifications and ideas within them can be altered and changed as the project is worked on. As your game comes together and you gather player feedback, you may come to realize that an idea you had early in the development cycle doesn't quite fit. Since it's a living document, the lead game designer has the power to alter the design and work with the team to get new ideas turned into new tasks. Don't be afraid to alter your design simply because it's been written out in the design doc. When building something as large and complex as a video game, you have to be flexible.

Game Design Task #1: Do you have a game design you've been thinking about, but never started in earnest? It's time to take that first step, which is to either grab a design notebook or cloud-based app and write, write, write! Every day, write about the gameplay, levels, hero abilities, enemy attacks. Write out quest ideas, dialog ideas, and anything else you feel is important. The more of your design you commit to paper, the easier it'll be to keep writing. The first step is always the hardest!

Paper Prototyping

There are some game designs that don't even require a computer to start playtesting. Board games, RPG's, puzzle games, and card games can all be tested with a game design tool called the **Paper Prototype**. By making a physical Version of your game to be played with dice, cards, and hand-drawn assets, you can quickly test the fun factor of a given game design.

Figure 18.2: Use dice, paper cutouts, coins - anything you can get your hands on to build a Paper Prototype of your game.

There are many benefits to paper prototyping, as listed below:

- **No Coding Required:** Paper prototypes can be made and enjoyed away from the computer. This allows you to focus more on the design and less on development and implementation.
- **Speed of Iteration:** If you come up with a way to make your game better, it's easier to iterate on a paper prototype than needing to rewrite code and create new assets.
- **Instant Feedback:** By sitting down and playtesting with friends, you can instantly tell which parts of the game are working. Excitement versus boredom. Clear rules versus confusing ones. Bored sighs and confused looks have a way of revealing faults, whereas smiles and laughter quickly identify strengths in your design.
- **Test Anywhere:** Instead of needing a computer, controller, and other electronic playtesting equipment, a paper prototype can be tested anywhere there's a table and friends.

If possible, always use Paper Prototyping to play your game at the earliest stages. Just like with White-Boxing, if it's fun in a simple form, your game will be even greater in its final form.

Concept Art

Establishing the visual design of your characters and world is an essential goal of Preproduction. To do this, you'll want to develop **Concept Art** - the sketches, designs, paintings, and rough ideas for heroes, enemies, items and environments.

Concept art can be used to visualize ideas and ensure what you're imagining translates into iconic gameplay assets. Remember, it's fast to iterate through pencil sketches of a hero, like the ones in [figure 18.3](#). It's much harder to iterate once a character is modeled, textured, rigged and animated.

Figure 18.3: Concept art is a great way to play with different ideas quickly. It takes a fraction of the time to sketch an idea instead of modeling and texturing it.

Use the pre-production phase to solidify the design for your visual assets, and work with the team members to make sure what's being proposed works with the rest of the game. Some things to look out for include the following:....

- **Odd Shapes**

In general, you want your characters to be able to navigate the world without colliders getting snagged on level geometry. A crazy design for an enemy may look cool on paper, but check to make sure it'll work in-game. Once something complex is rigged, you may run into a lot of headaches that could have been avoided with asking questions in Preproduction.

- **Reusing Skeletons**

When creating your assets, you can re-use animations by building characters that use the same skeleton. When designing a new enemy, NPC, or hero, check to see if you can tweak proportions to fit them onto an existing skeleton. They may not end up using ALL the same animations (you can often get by with a unique Idle and Walk Cycle) but it's a great way to prevent animator burnout.

- **Identifiable Colors and Silhouettes**

Make sure you're designing these assets at the size they'll be on-screen. From the in-game camera angle and distance, you'll want to make sure the Color and Shape of every unique asset is properly readable. Games require the player to assess situations quickly, so the outline of an enemy or color of a projectile should be instantly identifiable.

- **More Color Rules**

It's a good idea to come up with bright, easy to see colors that can signify different gameplay rules. Perhaps design enemies so weak ones have a visible blue element, strong enemies are wearing red, and brutal foes wear purple. Or maybe you want color to show how objects can be interacted with. These rules of visual communication are up to you - but set them early in the design process, and follow them *obsessively*.

Game Design Task #2: *It's time to get sketching! Grab a sketchbook and, every day, doodle some assets for your game. Characters, items, level ideas. You can spend hours on these designs, or they can be stick figures. The goal is to get your ideas out into the real world, where they can be examined, analyzed, and discussed.*

Mockups

Another important part of the preproduction step is the creation of **Mockups**: hand-crafted images that show the game as you're envisioning it.

These can be as loose or as detailed as you want, but the important thing is to have a visual representation of key game screens and moments ([figure 18.4](#)). Which elements are displayed in the HUD, where the camera will be positioned, art style, UI screens, etc.

Figure 18.4: A mockup is a hand-crafted image of what the game is supposed to look like in its final form. Try to match the intended art style - a retro game mock-up should use pixel art, for example.

Think of this as a visual target. As assets are created and the game comes together, these mockups will give you a guideline to make sure the look of the game, and the information provided to the player, match your vision.

Like other aspects of the pre-production process, mockups are likely to be tweaked and iterated upon. That is expected, and even encouraged, since improvements made from feedback ensure you're listening to players.

Some other concepts associated with Mockups, and the iteration of your visual style, include the following:...

- Previz

While a mock up is traditionally a still image, you can also render an animated pre-visualization (or Previz). By communicating your game vision using video, you can solidify aspects such as movement, animation, particles, and screen transitions. Like a mock up, it also gives you an agreed upon target, almost serving as a mediator as questions come up. When asked “How should this work?” or “How should this look?”, the answer can often be “Let’s consult the previz”. A great use of your Preproduction time.

- **Paintovers**

A paint over is when you take a screenshot from your game, bring it into an art program, and *literally* paint over it. Because it’s so much faster to add detail and make changes this way, you can quickly determine what new assets are needed, or what visual changes should be made, as you determine how to improve the visuals in your game.

- **Art style**

The mockup should also indicate what visual style you’re using in your game. Are you going for a realistic look, a cartoony look, or are you making a retro pixel art game? Any mock up or pre-visualization you create should reflect this.

By creating Mockups, Previz videos, and Concept Art, you’ll build an invaluable library of reference imagery, leading your team into the production phase with a visual roadmap towards the final product.

Game Design Task #3: When you close your eyes and imagine your game, it probably has some distinct features. Perhaps a hero standing at a certain location? A health bar and various elements in the HUD? Enemies or NPCs in the distance? Your next task is to start sketching this vision to create a Mockup of your game. Once you’re happy with the rough images, you can jump into Unity (or your art tool of choice) to bring together an image that resembles what you’re imagining. If you get frustrated, scrap it and try again. Most creative tasks take a few tries to perfect, and when it comes to the look of your game, it’s a crucial element worth perfecting.

Getting Creative

One skill you'll be utilizing throughout preproduction is **Creativity**: the use of your imagination, knowledge, memories, and even subconscious to form new and unique ideas. It's an easy concept to describe, but a difficult one to master.

Getting into a creative state of mind is *necessary* when formulating new gameplay concepts that will excite the player and make your game stand out. The problem is, creativity isn't something that you can simply *turn on*. Sometimes creative ideas fall like rain, other times, getting creative is like summoning water from the desert (literally called a "Creative Drought"). You can never be certain when the creative spark is going to strike, and while it's elusive and a bit unpredictable, there are a handful of concepts useful for harnessing your creative potential as listed below...

- **Open and Closed States of Mind:** There are two modes of creative thinking when tackling a problem, Open and Closed. When you're in Open mode, you're actively searching for a solution. This is where the majority of creative decision making is done. All ideas should be considered as you brainstorm as many solutions as possible. Closed mode occurs after a solution is decided upon and it's time to work. Don't let possibilities distract you as your team implements based on the agreed upon plan of attack. On a large project such as a video game, you'll be switching between these modes repeatedly.
- **Brainstorming Sessions:** One of the best ways to get the creative juices flowing is to sit down with a piece of paper (or a note taking app) and jot down all the ideas that pop into your head. Good ideas, bad ideas, and everything in between. If you think it, write it down. You may only end up using 2-5% of what you write, but by getting your thoughts onto paper, it's easier to organize your options and see what's working, and what's not.
- **The 5 Senses:** When faced with a creative dilemma, it can often be useful to find different ways to stimulate your senses. Use taste, touch, and sound to immerse yourself in the problem and find a source of creative inspiration. For example, if you're creating a level in a country you've never visited, spend an evening preparing a meal and listening to traditional music from that location. It's a simple (and often delicious) way to get into a more creative mindset!

- **Randomization:** There are websites online that will allow you to randomize the names of people, places and things. When creating an enemy, character or location, the inspirational spark you need may be a randomly generated name or phrase.
- **AI Helpers:** If you're looking for artistic inspiration, you can make use of new AI image generating tools. Enter a prompt and see what is randomly generated. It may just give you the creative spark you're looking for.
- **Get up and Move:** It can be hard to stay creative when sitting in the same spot for too long. Going for a walk, a drive, or a bike ride will relax your creative muscles and give the process a little room to breathe.
- **Letting it Simmer:** If you're still struggling to come up with the perfect idea, there's no harm in pushing the current query to the back of your mind and returning to it later. This is a great way to get your creative subconscious involved, with clever ideas hitting you like lightning in the shower, or coming to you in a dream. In this way, even the hardest creative problems can - *miraculously* - solve themselves.

Getting creative can be difficult, but struggling to find great solutions to hard problems will give you an advantage over games that use common solutions.

Creative thinking is a powerful tool in the Preproduction stage. Developing great ideas early will prevent lost time later in the project. You may feel the urge to jump into Unity and start developing, but the longer you resist that temptation and stay creative, the more mature your vision will be, and the better your game will be when completed.

Game Design Task #4: What makes your game special? It's a question that players are going to ask, so make sure you have a good answer - or perhaps several! Use the above tricks to get into a creative mindset and come up with several ways that your game is unique. Perhaps you'll want to use Juxtaposition to combine two different ideas into something fun and new? Perhaps you want to take some mechanics from another genre and see how they feel in your game? As long as you're doing something new and creative, players are going to notice!

Project Management

With pre-production materials in hand, it's time to start implementing your ideas in the production phase. Here is where you break down your vision into tasks to be hooked up by you, your team, or outside contractors.

This method of organizing and leading a large project through tasks, goals, and outcomes is known as **Project Management**.

Because there's so much to juggle when managing a project, it's useful to have some rules in place. One of the first, and most important, is called the **Iron Triangle** rule ([figure 18.5](#)). Here, any goal can be broken down into three main areas of importance: Quality, Time to Complete, and Cost. The rule states that, at most, you can choose *two* of these areas to prioritize. Perhaps you need a high-quality intro movie created. You can either have it made *fast*, or *cheap*, but not *both*. Or perhaps you need something quickly and inexpensively. The Iron Triangle dictates that, in this case, you can expect quality to suffer.

Figure 18.5: The Iron Triangle. You can prioritize two of these categories, but not all three.

Accepting this rule, and managing the trade offs between the three priorities, will come up repeatedly when building your game.

Another useful project management axiom is known as the **80/20 Rule**, which states that **80%** of your outcomes come from **20%** of the inputs. In other words, there is a *small* set of *very important* tasks, and a *large* set of *unimportant* tasks.

As the project manager of your game, it's your job to sort and prioritize vital tasks over trivial one. You'll still *eventually* want to work on the 'less important' tasks - which tend to fall into the *polish* category - but make sure you're getting the major features implemented first.

These two rules will help you in the most important goal of a project manager - *scheduling tasks*. You'll need to determine *what* gets worked on, *when* it gets worked on, and what the *quality expectations* are.

The Mile-High View: If you're leading a team, it may be tempting to give yourself a large task list to move the project forward. Resist this temptation - your job is to oversee the project, not to implement systems

and create major assets. The team needs you looking at the Mile-High View, where you’re judging the game as a whole. Your focus should be broad: playtesting builds, coordinating tasks, managing scope and team morale. Task-implementation requires a narrow focus - which is not what the project needs from leadership. This obviously depends on the team size - a solo dev has to be the one implementing features - but assume that mid-sized to larger teams need leadership focusing on the management aspects of production.

To help you out, there are plenty of web-based project management solutions out there. Look into apps like Trello and Jira ([figure 18.6](#)), which will let you fill up a task board, set up milestones, and keep you (and your team) focused and productive.

Figure 18.6: Jira is a great tool for organizing tasks and managing your project.

Game Design Task #5: *Organizing your project is necessary, even if you’re a solo developer, so take the time to create a Jira or Trello account (or some other project management tool). You want a single spot for categorizing the work that needs to be done, so establishing this now is a great way to get comfortable creating new tasks and marking them as complete. In fact, your first task should be “Set Up Project Management Board”. Once done, set that task as complete. Feels great to finish tasks, doesn’t it?*

The Vertical Slice

When managing large Video Game projects, it can be easy to get overwhelmed by the magnitude of work that needs to be done. While the pre-production phase does a great job laying out your roadmap, it can also have the negative effect of *appearing* insurmountable.

We know this isn’t true, though. The tools we’ve learned in Unity give us everything we need to quickly implement almost any idea. Nothing is insurmountable, and so we simply need to start with something *small* and *focused*. This method of taking a game design and quickly implementing a playable, fun, and visually interesting prototype is known as **The Vertical Slice**. By identifying a small section of the game to implement and playtest

quickly, you can start your project fueled by confidence and excitement (instead of being overwhelmed by the workload).

Once you have a vertical slice built, it's time to hand it off to playtesters.

Play Testing

Getting regular feedback on your game from trusted play testers is an important part of the game design loop. As the creator of the game (and possibly lead developer and artist, too) you're far too close to the project to do proper playtesting. Certainly you can test for bugs, but the question of "Is this a good game?" has to be answered by external sources.

When getting ready to set up playtesting sessions, there are several rules to follow as listed...

- Compile a list of trusted playtesting friends from whom you can rely on honest, quality feedback. Search for internal playtesters that will give praise and constructive criticism in equal doses. It's good to know what *is* working, but the most actionable feedback will be of the *negative variety*.
- Send regular testing builds of your game and key milestones to make sure that the game that you're envisioning is the game they're experiencing.
- Cycle through playtesters, giving ample time between feedback sessions. This way, testers can see significant improvements being made and will feel their feedback is being considered.
- If your testers aren't being paid, be patient and courteous when feedback doesn't come back at the speed you'd prefer. Playtesting burnout, or underappreciation, is a quick way to lose testers (and anger friends).
- Recorded, in-person sessions are a great way to conduct playtesting for your game. Since it can be hard for a playtester to articulate everything they're feeling in written feedback, watching them play is certainly the better option. Experiencing nuances in their facial expression, or subtle frustrations as they struggle, will give you insight that would normally be hard to parse. Whenever possible, get some lunch and plan some in-person playtesting sessions!

- Don't forget to add playtesters to the credits. Ask how they would like to be listed, and make sure you shower them with appreciation. These are normally friends and acquaintances taking time out of their schedule to help your game succeed. Thanking them properly is an important part of the process.

Playtesting is very important to the overall quality of your game. You may be proud and excited about how your game is coming together, but playtesters may find it boring, or confusing. Don't take their feedback *personally*, but you *should* utilize it to make your game better.

Remember, the public will be *much* more harsh than your friends. It's better to get critical feedback now than bad reviews upon release!

Game Design Task #6: *It's never too early to start building that list of playtesters. Send out emails, or message friends directly, to see if they'd be interested. Have the list ready, and give them a heads up when a build is ready to test. If they're still interested, send it over. The more outside feedback you receive, the better your game will be!*

Analytics

Another useful (and unintrusive) way to gather data from play testers is by using **Analytics**. Analytics are chunks of data, sent during a players gaming session, that you can collect - and later analyze - to determine if the game is being played as you'd expect. You can collect data on how fast levels are completed, how many times a player dies in a given spot, what weapons they prefer to use, etc, and it's all done without interrupting the player's experience.

By amassing information about the players, as they're playing it, you'll have the necessary insight to make balance and design tweaks without the players needing to complain.

Unity provides some powerful Analytics tools, which are a part of their *Gaming Services* platform. Log into the Unity website, click your user name in the upper right corner, and select Organization Settings. One of the options in the left hand column will be **Analytics**. Press that tab to get started ([figure 18.7](#)).

Figure 18.7: You can access the gathered Analytics information from the unity web portal.

Paying for Cloud-Based Services: Gathering Analytics is a great way to capture gameplay from all the play sessions, but because that data has to be stored on a server somewhere, it's important to remember that most Analytic services will come with a small fee. For Unity, the first several million events will be free, so using it for your initial playtesting sessions will not cost you anything. Once your game becomes more popular, you may see a cost incurred, but even that will be a small price to pay for the insight you're gathering.

Prioritizing Feedback

Once you have a nice amount of data from play testers, it's time to start prioritizing feedback.

The first important step is to separate bugs, complaints and suggestions. Bugs are obviously the most important feedback to pay attention to, since errors in your code can lead to game breaking issues that will ruin even the best experience. Sometimes these are small bugs, but most of the time they'll be issues that need to be fixed for the next milestone.

You'll also be getting complaints from players about parts of your game that are either confusing, too hard, or sometimes even too easy. Take these frustrations seriously, especially if more than one tester has brought them up. Sometimes this means changing or removing a part of the game that you have a deep fondness of. If testers say something isn't working, then it's probably *not*, and you'll either need to change that feature or remove it.

Killing your Darlings: When receiving feedback in a creative industry, you may come to realize that, for the sake of the overall project, one of your favorite elements needs to be removed. This can be a hard situation to stomach, since it involves throwing out and reworking aspects of your project that you felt more important. Never be afraid to trim the fat and reduce scope! If multiple trusted outside sources dislike something about your project, chances are the public at large will also dislike it. Sometimes you have to swallow your pride and admit that something you love needs to be cut. It will hurt in the moment, but you'll come to appreciate the decision in the long run.

You may also get detailed suggestions from play testers about tweaks to the design that they would make. This form of feedback is a bit more tricky to navigate, and while it should be considered, never feel obligated to prioritize external suggestions. A clear concise game vision can quickly turn into a bloated mess if too many random ideas and features are thrown into the mix. Only absorb suggestions that fit perfectly into your overall vision, but always thank play testers for their ideas - it never hurts to have others considering ways to improve your game.

Once you've read through the feedback and have a good understanding of how the issues need to be prioritized, you'll want to jump into your project management software (Jira, Trello, etc) to turn the feedback into official tasks. Use the prioritization options to mark important issues as highest priority, back-burner items to low priority, and everything in-between.

Collecting and acting on player feedback is a critical part of the game development cycle. Fix issues and implement new features quickly so you can continue to get feedback. Great games come from constant iteration.

Game Design Task #7: Once players get to play your first milestone (whether it's a vertical slice or white boxed demo), take their feedback and see how it lines up with your design. Prioritize fixing any issues that came up multiple times, and move on to the implementation of more levels and content. Plan your next build, And give yourself some extra time as wiggle room, ensuring you're never stressing out about hitting an arbitrary date. By improving your game based on player feedback, you're on your way to making something great!

Milestones, Momentum and Morale

No matter the team size - whether you're leading a large group or you're a solo team of one - keeping an eye on both progress and excitement is vital to the completion of your game. If a project is taking too long, artists and developers will start to lose faith in your vision, which can lower output and exasperate frustrations across the entire team.

You can bolster a happy team *and* keep the project on track by managing these 3 M's: Milestones, Momentum, and Morale.

- **Milestones**

You can't talk about project management without underscoring the importance of milestones. As we've learned in previous chapters, a Milestone serves as a bookend to a focused period of implementation. It allows you to wrap up one area of your game before moving onto others. By establishing and hitting specific milestones, you and your team will always feel like you're making progress.

- **Momentum**

With the constant, focused implementation of features required to hit your milestones, you and your development team will build up momentum in your output. Once familiar with the tools and code, tasks will move faster and you'll find yourself getting high-quality work done at a quicker pace.

- **Morale**

It's important to keep your team's excitement and happiness levels high. There's a lot of work to get done, and an upbeat, engaged team will rise to the occasion. Teams with high morale will produce quality work, filling in design gaps by eagerly contributing ideas. Even if you're the *only* team member, always be paying attention to your personal morale. **Burnout** is real, and is destructive to any creative project. Always prioritize the mental health of you and your team.

Remember: It's easy to create a large and exciting design doc filled with cool concept art and mock ups. It's also easy to get overwhelmed by a colossal vision that expects 100% output at all times and no mistakes to be made. When managing your project, set reasonable milestones that allow for steady progress. As the vision is brought from *idea to reality*, the team will automatically get excited as their hard work pays off.

In this way, hitting Milestones leads to a boost in Morale. Stronger Morale improves Momentum. And Momentum produces exciting Milestones. This 'slow and steady' cycle is a positive feedback loop that ensures nobody burns out, and the project is completed on time and at a high level of quality.

Game Design Task #8: Take your game design and map out all the milestones needed to get it to the finish line. Use your project management software to estimate how long this will take. If it's going to take longer than you feel comfortable with, can you trim down scope to

reduce the burden? Remember, it's better to make a quality game with a small scope than create a sloppy, half-finished game with a large scope.

Early Access

With the invention of digital marketplaces comes flexibility in a game's release schedule. For early console generations, games would be completed and tested to be as bug free as possible. Once an extensive QA process was complete, the final build would be burnt to a physical cart (or disc) and released to the public. Players had high quality expectations and were more interested in playing games than being a part of the creation process.

Now that games are released digitally, however, it opens the door for earlier releases. These **Early Access** builds are put out for users to enjoy in a half-finished state. With players becoming play-testers, you instantly build a community of excited users that play new milestones and give lots of great feedback.

And while this approach has been used to create plenty of amazing games, an Early Access release schedule has several pros and cons as below

- **Pros**

- Get to build a community early.
- Get to start selling your game early.
- Lots of feedback and ideas from early adopters.
- When creating a PC/Mac build, many testers means various hardware configurations are being tested.

- **Cons**

- Players may get impatient if a project starts dragging.
- Players may complain about the rough state.
- You'll have to be careful about listening to feedback - it may be good and thorough, but it may not actually line up with your vision.
- Harder to hide big marketing surprises when the game has been out for months.
- The official release may feel underwhelming to you.

- Less design flexibility - removing large features may be right for the game, but could anger the community.

The “Cons” list may look long, but don’t let that deter you from releasing a game early to get feedback and start building hype ASAP. With so many games out there, assume it’ll take a while for word of your game to get out there. Early Access is a great way to build your user base and include eager players in the process!

Marketing

Once your project is far enough along, it’s time to start considering **Marketing**: reaching out to potential users through advertisement, social media, and other channels to build hype for your big release. And whether you’re self-publishing your game, or you’ve teamed up with a publisher who will formulate an official marketing plan, there are some great ways that Unity can help in your efforts.

First is the creation of marketing art assets. Just like when we rendered icons directly in the Unity editor, you can set up scenes to create compelling artwork and animations to use in your marketing efforts. By making use of in-game assets, you can quickly create an interesting composition, like the one arranged in [figure 18.8](#)

Figure 18.8: In the Unity editor, you can set up interesting scenes to help produce marketing art and assets!

You can also use Unity’s Gaming Services to create an advertisement campaign, getting your title in front of other players ([figure 18.9](#)). While this option is mobile only, it can be a great tool for building a user base across Android and iOS devices.

Figure 18.9: Use the Unity web portal to set up an advertisement plan to grow your user base.

Outside of Unity, there are several interesting ways to build up excitement for your game.

- **Join Online Communities**

The Indie game dev community is a great group of developers. Join activities - such as #ScreenshotSaturday on Twitter - to put images of your game out there and start building interest in your project.

- **Create an Online Community**

By creating a Discord server or personal forum for your game, you can build up a trusted group of players that you can brainstorm ideas with, or simply chat with about random life stuff. It's always great to have a solid community backing you up!

- **Participate in Podcasts**

There are some great indie podcasts out there, and if your game looks interesting, most would love to have you on to chat.

- **Find a Local Meetup**

It's great to hang out with fellow developers. See if your area has a local IGDA Chapter (Independent Game Developers Association). If they don't, perhaps you could find some like minded developers and start your own meetup to show what you're working on. Even if it's just to get feedback and grow your circle of connection, meetups are a great way to keep your morale high.

Game Design Task #9: Once you have some marketing artwork that you're happy with, take it and turn it into a shirt, a mug, or stickers. While the digital art forms are fun, there's something about holding something physical that makes your project feel more real.

The Gold Master...and Beyond!

It's time to look ahead to the future of your project.

There will come a time where your main task list is empty, play testers are thoroughly enjoying your game, and you start to think that - perhaps - it's time for the *official release*. Hitting version 1.0 is a huge milestone, one that is equal parts *exciting* and *terrifying*.

When preparing your **Gold Master** - the official *final build* that will go out to players - there are some important tasks to remember which are listed below...

- Update version to 1.0, both in build settings and in any text fields.

- Remove any early access or beta version watermarks.
- Dedicate at least 2 weeks to playtesting. It's important to verify that no show stopping bugs have slipped into this build.
- Make sure the build is set to Release mode, which will give a nice performance boost.
- Be prepared to hang out on forums as people start asking questions about your game. Making yourself present for these conversations is a great way to win fans.
- Plan a release party, or at least a release dinner, to take a moment to celebrate your accomplishment. Releasing a game is a big deal, so remember to make a big deal out of this monumental day!

The weeks leading up to, and directly following, launch will be both thrilling and terrifying. This is the moment that you've been working towards, so while it's immensely satisfying to see all of your effort pay off, there can also be a bittersweet feeling of "What now?"

With older games, the release of a title marked the end of that project. Developers and studios would be at a crossroads, and it would be time to start thinking about the next big game.

Digital distribution changes all that. Modern players have come to love titles that continue to put out content well past version 1.0. In fact, it's easy to look at the official launch as a starting point. All the major features are implemented and the game has been released - It's time to think of ways to make your game even *better!*

With this in mind, it's a good practice to consider putting out Updates for a year or more after the official release. **Updates** are versions of your game that fix bugs and include new features that weren't in v1.0. It's a way to keep players engaged and excited, and gives you opportunities to continue marketing, bringing in even more users.

Releasing the Gold Master is no longer the end of a project, but simply a new beginning. Let your passion and excitement spill over into post-release content, and players are sure to take notice!

Game Design Task #10: Even in the preproduction stage, you should be thinking ahead to post release content for your game. Don't try to cram all your best ideas into version 1.0. Instead make a great core

experience that leaves the players wanting more, then use updates to give it to them! Make a post-release doc that you can fill with ideas for updates once the game has been officially released. Remember, in game design, the end is often just a new beginning.

Conclusion

You now have the tools, skills, and understanding required to become a *master game designer*. Unity gives you a powerful editor and flexible systems perfect for making any game you can envision. And by working your way through this book, you have the game design knowledge to bring your vision from *concept* to *completion* - and beyond!

Just like games themselves, the road to mastery takes you from the *unknown* to the *known*. It requires dedication and practice. It requires the honing of skills through trial and error. And in the case of Mastering Game Design, it requires the designing, building, and releasing of *awesome games*.

In this book we went *broad* in our learning, covering many game design ideas as they lined up with Unity's tools and systems. Now, you get to take that knowledge and go *deep* - building on what you've learned to create your own game, expanding and sharpening your skill set as your project goes from vision to reality.

With every new concept you encounter comes *knowledge and growth*. Never stop learning! With every new struggle you overcome comes *strength and perseverance*. Don't succumb to frustrations! And with every new milestone you release, you'll move closer to that ultimate goal of *game design mastery*.

We're in a golden age of game design.

What are *you* going to create?

Questions

1. In what ways can the Proproduction process save you time and effort once you start developing your game?
2. Why is it useful to write your Design Docs in a cloud-based tool?

3. True or False: Once an idea has been written in a Design Doc, it can NEVER be changed.
4. What are some of the game genres that can be tested using a Paper Prototype?
5. In what ways are Previz videos more useful than a Mockup?
6. When you're in a Closed creative state, should you be coming up with new ideas, or implementing the ideas already agreed upon?
7. True or False: You only need a Preproduction phase if you're on a team. If you're a solo developer, you should jump right into development.
8. The Iron Triangle rule of Project Management states you can focus on two out of three priorities when working towards a goal. What are these three priority categories?
9. When assigning development tasks on a mid-sized team (or larger), you may be tempted to give yourself tasks to help the team out. Why is this a bad idea?
10. How can Project Milestones, Team Morale, and Development Momentum work together to form a positive feedback cycle?
11. In what ways can an Early Access release help your project? In what ways can hurt your project?
12. When promoting your game, what are some of the channels you have to make connections and show off your project?
13. Why is post-release content important?
14. In this book we went broad on many topics. Is there a topic you want to go deep into?
15. What kind of game do you want to make? What is the clever hook that will set it apart?

Key Terms

- **Preproduction:** The earliest phase of game development, where you're organizing and outlining the details of your vision.
- **Design Docs:** The written design of your game. This should break down gameplay mechanics, character, world building, story, and

anything else you feel is important to lay out before coding and development begins.

- **Elevator Pitch:** A 1-minute, 2-5 sentence explanation of your game. Needs to be brief and instantly captivating. Use this in an overview doc to get incoming team members excited for your vision.
- **Living Document:** A document that can change if necessary. Design docs are ‘living’, in that they will lay out initial design plans, but should be altered if feedback comes in that part of the game needs improvement.
- **Concept Art:** Sketches, painting, sculptures, and any kind of art that defines the look of various assets in your game. Heroes, enemies, environments, UI elements, items - everything important should be sketched and thought through in the Preproduction process.
- **Mockups:** Hand-crafted images that show - at an early stage - what your game will look like. Should visualize HUD/UI elements, characters, environments, and art style.
- **Paper Prototype:** Using paper cutouts, dice, and other items to make a physical version of your design to play with friends. While this only works with certain genres, it’s a great way to test and iterate a design before jumping into code.
- **Creativity:** Using your imagination to form new ideas. In an oversaturated industry, pushing for the most creative solutions will help your game get noticed.
- **Open State:** A state of mind where you’re looking for the best, more creative solution to a problem. Keep exploring possibilities until you (and your team) are happy with a great idea that you can all agree upon.
- **Closed State:** Once a creative solution is determined, it’s time to stop thinking about all the possibilities and start implementing *the one that was agreed upon*. It may be possible that this solution still needs work, or perhaps it was the *wrong* solution, so implement quickly so you can verify how an idea fits into the broader project. You may need to return to the Open state if more work needs to be done.
- **Project Management:** The process of leading a project through tasks and goals to reach a given outcome - in this case, a completed game.

- **The Iron Triangle:** When working towards a goal, you can prioritize Quality, Cost, or Time. At most you can prioritize two of these, but never three. Always consider the tradeoffs between these areas when managing a project.
- **Tasks:** The itemized work that needs to be done to implement and complete your game. These may be implementing features, designing content, improving tools, hooking up assets, fixing bugs, and planning marketing events. Try to get every task, no matter how small, organized in your management system. Making tasks as done, and seeing your ‘total completed task’ number go up, is immensely satisfying.
- **Momentum:** By becoming familiar with the systems being used - as well as keeping morale high - you can gain momentum. This allows for tasks to be done quicker, and at a higher quality level, than earlier in a project.
- **Morale:** The happiness levels of your team - and yourself. Often directly related to output. Don’t let mental health go overlooked.
- **Burnout:** When you, or a team member, is pushed to exhaustion. Often requires an extended leave to recuperate, and many times has lasting effects. Work hard to keep your team happy and healthy, and they will reciprocate.
- **Vertical Slice:** A small, polished, fun demo of your game.
- **Playtesting:** Sending builds of your game to outside sources for testing. Will give feedback, which can be used to iterate and make the game better.
- **Analytics:** Sending information about how a game is being played. Give you data to analyze about play sessions, which can be used to make tweaks in future builds.
- **Early Access:** Putting out an early, unfinished version of your game for players to purchase and give feedback on. A great way to build a community that becomes part of the creative team.
- **Gold Master:** The final v1.0 build of your game.
- **Updates:** Released versions of your game beyond 1.0, often including bug fixes, new features, and fresh content. A great way to keep players engaged.

- **Launch Party:** When you reach a big milestone, like the release of your game, you need to celebrate. This can be a party, dinner, or some other event that builds comradery and appreciation for how far you've come. At the end of any hard-won achievement, it's important to feel appreciated. "It was hard work, but you did it. Great job!"

Index

Symbols

2D Animation [432, 433](#)
2D Beginner [451](#)
2D Game Kit [447, 448](#)
2D Hinge Joints [434](#)
2D physics and movement [431, 432](#)
2D projects
 versus, 3D projects [414, 415](#)
2D Sandbox [416, 417](#)
3D Game Kit [456](#)
3D Sound
 basics [276-278](#)
 for Weapons [282, 283](#)
 on items [279-281](#)
3D Space
 filling [325-331](#)
3D Studio Max [308-310](#)
80/20 Rule [495](#)

A

Achievements [372, 373](#)
AdjustCurHealth() function [65](#)
advanced combat mechanics [260, 261](#)
AI Actions [225, 226](#)
AIBrain class [78](#)
AIBrain component [222-224](#)
AI Component
 setting up [228-232](#)
Aim Assist [261](#)
Air Juggling [261](#)
Ambient Occlusion effect [316](#)
Ammo [260](#)
Analytics [497, 498](#)
Anchor Presets [107-111](#)
Anchors [107](#)
animations
 implementing [212-215](#)
Animation View window [199, 200](#)
 _applyToTarget [77](#)
Arena Shooters [453](#)
Artificial Intelligence (AI) [220](#)
art style [492](#)

Audio Clips
importing [273-275](#)
Audio Components [266](#)
 Audio Listener [266](#)
 Audio Source [267, 268](#)
Audio Design [265](#)
Audio Mastering [283](#)
Audio Mixers
 mastering [283, 284](#)
Audio Rolloff [278](#)
Awake() function [38](#)

B

Basic Bullet [80-85](#)
basic collider shapes [161, 162](#)
 box collider [162](#)
 capsule collider [162](#)
 sphere collider [162](#)
basic enemy [220-222](#)
basic hero
 creating [193-198](#)
BlackBG object [107](#)
Blender [308](#)
Bloom effect [316](#)
Bones [433](#)
Bottomless Pit [358](#)
BreakableBrickBlock [297](#)
Breakable Weapons [261](#)
Breaking Expectations [402](#)
bugs [150](#)
 buggy bullets [151, 152](#)
 bullets forever [153](#)
 endless pit [150, 151](#)
Build and Run option [155, 156](#)
Building Blocks [141, 297-300](#)
BulletObj_Basic [81](#)
Bullet Sponge Boss [401](#)
Bullets [63](#)
burnout [500](#)

C

Camera
 mastering [133-136](#)
Camera gizmo [15](#)
CameraShake [86](#)
Canvas component, Title Menu
 pixel perfect [103](#)
 render mode [103](#)

sort order [103](#)
Canvas Scaler component [104](#)
Capsule Collider [244](#)
Capsule Collider 2D [431](#)
Cinemachine Cameras [435-438](#)
Class Inheritance [38](#)
clean-up process [215, 216](#)
coin
 creating [46, 47](#)
 spawning [52, 54](#)
collection detection [48](#)
 OnCollisionEnter() [48](#)
 OnCollisionExit() [48](#)
 OnCollisionStay() [48](#)
Combat [62](#)
 subsystems [62, 63](#)
Community tab [8](#)
compatibility [472](#)
Components [34](#)
 adding, to object [37, 38](#)
 examples [35-37](#)
 extending, in Unity Editor [42-44](#)
Concept Art [489, 490](#)
Corgi Engine [448](#)
Creativity [492, 493](#)
Creator Kit [458](#)
Critical Feedback [131, 132](#)
cube
 adding, to scene [24, 25](#)
_curFacing [84](#)
Custom Rolloff [278](#)

D

damage feedback [85-88](#)
Damage Types [260](#)
Death Effects [338](#)
Debug.Log() function [55](#)
debug messages [55, 56](#)
Depth of Field effect [316](#)
Design Docs [485-487](#)
_destroyOnCollision [77](#)
Dev Kit [474](#)
Difficulty Curve [354-356](#)
 managing [356](#)
Directional Light [15, 184](#)
Distance Fog [313, 314](#)
Dopesheet [201](#)
Doppler Level [277](#)
Double Key-Presses

handling [44](#)
Dropped Weapons [260](#)
Dungeon Crawler [450](#)
Dust Motes [325](#)

E

Early Access [501](#)
Easing [192](#)
Editor [12](#)
Elevator Pitch [486](#)
Endless Runner / Jumper subgenre [446](#)
enemy
 testing [235, 236](#)
Enemy objects [220](#)
EnemyObj_Spikes [75](#)
Environmental Storytelling [402](#)
Error List [66](#)
Explosion effect [146-150, 338-347](#)

F

feedback
 prioritizing [498, 499](#)
Fire Object [360](#)
First Person Shooters (FPS) [452-455](#)
Floating Spike [71](#)
Font Licensing [389](#)
for loops [54](#)
FPS Superhot [461](#)
Freeze Rotation [173](#)

G

game
 publishing [478, 479](#)
Game Design
 Risk and Reward [62](#)
Game Engine [2](#)
Game Flow [96](#)
 Cutscenes [97](#)
 dialog popup [97](#)
 game over screen [97](#)
 inventory screen [97](#)
 loading screen [98](#)
 main game HUD [97](#)
 options menu [98](#)
 pause menu [97](#)
 storefront and community screens [98](#)
 title menu [97](#)

versus, Gameplay Flow [96](#)
GameNameText [105](#)
GameObject [16](#)
Game Over text [124-126](#)
Gameplay Palette [185](#)
Game Session manager [63, 88-91](#)
Game View window [13](#)
gaming systems
 Linux [469](#)
 Mac [469](#)
 PC [469](#)
Genre [443](#)
GetComponent() function [50](#)
GetComponent() lookup function [40](#)
Gizmos [14](#)
Gold Master [503, 504](#)
Grid Snapping [301, 302](#)

H

Hand-Drawn Art Style [455](#)
Hard Lock
 versus, Soft Lock [88](#)
Heads Up Display (HUD) [116-121](#)
 _healthChange [77](#)
 _healthCur [64](#)
Health Manager [63](#)
 creating [63, 64](#)
 _healthMax [64](#)
HealthModifier component [76-80](#)
Health Modifiers [63](#)
Hierarchy Window [14](#)
Hooks the Player game
 first [10](#) minutes [395, 396](#)
HUD Component [121-124](#)
Hype [480](#)

I

Iconography [383-387](#)
Idle pose [198, 199](#)
Increment Snapping [302](#)
Inspector window [15, 16](#)
Instantiate() [54](#)
Intellisense [26](#)
Intensity Multiplier [313](#)
Introductory Stage [396](#)
 building [396-400](#)
 _invincibilityFramesCur [65](#)
 _invincibilityFramesMax [64](#)

Iron Triangle rule [494](#)
_isDead [65](#)
IsDead() function [67](#)
Is Trigger parameter [81](#)
IsValidTarget() [78](#)
Iteration [56](#)

J

Jump [45](#)
Juxtaposition [461, 462](#)

K

Keyframe Recording Mode [203](#)
Keyframes [201-206](#)
K.I.S.S. Philosophy [69-71](#)
KitBashing [308](#)
'Knockback' effect [254, 255](#)

L

Learn tab [7](#)
Left/Middle Anchor [107](#)
level design [136-138](#)
 first level, designing [141, 142](#)
 teaching through [357, 358](#)
 tips [142, 143](#)
LevelObj_Ground [25](#)
Level Prefabs
 building [303-308](#)
lighting way [358-360](#)
Linear progression [354](#)
Lit Environments [358](#)
Living Document [487](#)
Loadouts [260](#)
Looping [202](#)

M

Main Camera object [133](#)
marketing [480](#)
Marketing [502, 503](#)
mastery [370, 371](#)
Material [290](#)
matRedBrick [299](#)
matTransBlue [167](#)
Maya [308](#)
Melee attack
 animating [255-258](#)

basics [251-253](#)
mesh collider component [162](#)
Meshes [290](#)
Metroidvania [446](#)
Middle/Center Anchor [108](#)
milestone [130, 131, 500](#)
mobile games [470-474](#)
 console development [474, 475](#)
Mockups [490, 491](#)
momentum [500](#)
MonoBehaviour class [38](#)
morale [500](#)
Move Set [456](#)
Mudbox [308](#)
Music and SoundFX, importing
 creative commons [269](#)
 outsourcing [268](#)
 royalty free libraries [269](#)
 Unity Store [269](#)

N

Navigation Mesh [232](#)
 agents [232](#)
 areas [233](#)
 bake [233](#)
 generating [233](#)
 object [233](#)
NavMesh Agent [234, 235](#)
New Game + [373](#)

O

Object-Based Animation [432](#)
objects
 moving [20-22](#)
 placing, in scene [17-20](#)
 turning, into Prefab [50, 51](#)
Onboarding [357](#)
OnClick() list [114](#)
onDeath() function [66](#)
onPickedUp() function [50](#)
onPressStartGameBtn() function [114](#)
OnTriggerEnter() [49](#)
OnTriggerExit() [49](#)
OnTriggerStay() [49](#)
OnTriggerStay() function [170](#)
Orthographic View [136](#)

P

pacing [371](#), [372](#)
 overwhelming pacing [372](#)
 strong pacing [372](#)
 weak pacing [372](#)
paint over [492](#)
Paper Prototype [488](#)
 benefits [488](#), [489](#)
Parallax [438](#), [439](#)
ParallaxLayer [439](#)
Particle Effect Panel [326](#)
Particle Effects [323](#), [324](#)
PBR graphics system [311](#)
Peaks [355](#)
Permadeath [450](#)
Permanence [150](#)
Physics components [160](#)
 tweaking, through code [166-171](#)
Physics Joints [163-166](#)
PhysicsSandbox [160](#)
Pivot Point [295](#)
Platform [468](#), [469](#)
Platformer [446](#)
Play button [22-24](#)
player
 hunting down [226](#) [228](#)
 teaching [364](#)-[368](#)
 testing [368](#)-[370](#)
Player Controller component
 _movementAcceleration member [39](#)
 _movementVelocityMax member [39](#)
 writing [39](#)-[42](#)
PlayerController script [49](#), [249](#)
Player Expectations [462](#), [463](#)
Player Feedback [85](#)
Player object [220](#)
PlayerObj_Sphere object [68](#)
Player Progression [353](#)
playtesting
 setting up [496](#), [497](#)
Polish [405](#)
Post Processing [315](#)-[319](#)
Prefabs
 benefits [55](#)
 making [46](#)
 object, turning into [50](#), [51](#)
 spawning [46](#)
Pre-Production [193](#)
Previz [491](#)

Principles of Animation
Anticipation [191](#)
Appeal [193](#)
Arc [192](#)
Exaggeration [191](#)
Follow Through / Overlapping Action [192](#)
Secondary Action [192](#)
Slow In / Slow Out [191](#)
Squash and Stretch [190](#)
Staging [191](#)
Timing [192](#)
ProBuilder [291](#), [292](#)
Center Pivot [295](#), [296](#)
Extrusion hotkey [296](#)
Material Editor [294](#)
poly shape tool [293](#)
shape tool [292](#), [293](#)
smoothing [293](#), [294](#)
UV Editor [295](#)
Progression Tree [372](#)
Projectile weaponry
 basics [253](#), [254](#)
Project Management [130](#), [494](#), [495](#)
 milestones [130](#), [131](#)
proproduction [484](#), [485](#)
public members [42](#)
publishers
 teaming up [480](#)
publishing [478](#)
Pushable Block [171](#)-[174](#)
puzzle games [460](#)

R

Random.Range() [54](#)
Rect Transform component [107](#)
Reference Resolution [104](#)
Reset() function [67](#)
Resources Folder [50](#)
Rigidbody 2D [431](#)
Rigidbody parameters [163](#)
Roguelike [450](#)
Role Playing Games (RPGs) [457](#), [458](#)
Run Cycle [206](#)-[211](#)

S

SampleScene [131](#)
SceneManager [112](#)
Scene mode [13](#), [14](#)

Scene View window [13](#)
Scope [444](#)
 concerning [444, 445](#)
Scrubbing [204](#)
Searchbar [14](#)
Seesaw platform [174-177](#)
SerializeField attribute [42](#)
Shadows
 utilizing [183, 184](#)
SHMUPs (Shoot-Em-Ups) [447](#)
Sidescroller [446](#)
Skeletal Animation [432, 433](#)
Skybox [311-313](#)
Slicing [420](#)
smooth transition [405-410](#)
Soft Lock [88](#)
Sound Manager
 creating [275, 276](#)
Spatial Blend slider [276](#)
SpawnedSoundFX Prefab [280](#)
Spawn Zone [54](#)
Spike objects [71-76](#)
Springboard [178, 179](#)
Sprite [417](#)
 identifying, in project [418-420](#)
Sprite Batching [417](#)
Sprite Renderer component [419](#)
Sprite Sheets [420-422](#)
start() function [39](#)
static spikes [143](#)
 improving [144, 145](#)
Storyboarding [68, 69](#)
Strategy Game genre [458](#)
 4X strategy [459](#)
 RTS [458](#)
 Tower Defense subgenre [459](#)
Stress Testing [235](#)
Substance Painter [308](#)
Sun gizmo [15](#)
SwitchSceneOnCollision [403](#)

T

Text Heavy UI [383](#)
The Build [153-155](#)
The Lost Crypt
 studying [429, 430](#)
The Vertical Slice [496](#)
Third-Person games [455, 456](#)
Tilemap Renderer [428](#)

Tilemaps [423-429](#)
Tile Palette Toolbar [426](#)
Time.deltaTime [41](#)
Title Menu [101](#)
 Button component [113, 114](#)
 Canvas component [102](#)
 Rect Transform [102](#)
 scenes, adding to Build Settings [114-116](#)
 script, adding [111-113](#)
 UI elements, adding [104-106](#)
Toggle Effects [315](#)
tools [138](#)
 Move tool [139](#)
 Rect tool [140](#)
 Rotate tool [140](#)
 Scale tool [140](#)
 Transform tool [140](#)
 View tool [139](#)
Tooltip [43](#)
Top-Down adventure [449, 450](#)
 creating [451, 452](#)
TopDown Engine [451](#)
TorchfireBlock script [361](#)
Trail Renderer [259](#)
trials [370, 371](#)
triggers [48, 49](#)
TRL (Technical Requirements List) [475](#)
Tutorials [357](#)
Tweening [201, 202, 206](#)
Typography [387](#)
 Font importing process [388, 389](#)
 Multiple Languages [387](#)
 Never Use Defaults [388](#)
 Picking Readable Fonts [387](#)
 Showing Personality [387](#)
 Walls of Text [387](#)

U

Unity animation system [189](#)
Unity art tools
 setting up [288, 289](#)
Unity Asset store
 using [269-272](#)
Unity Asset Store [3](#)
Unity Audio System [266](#)
 platform considerations [285](#)
Unity Engine [1-4](#)
 Editor [12](#)
 extras [7-9](#)

installing [4-7](#)
project, creating [10, 11](#)
tutorials [7-9](#)
UnityEvents [225, 226](#)
Unity Hub [4](#)
Unity Particle Pack [347-350](#)
Unity Physics System [160](#)
Unity UI objects [98](#)
 Button [99](#)
 Canvas [99](#)
 font assets [100](#)
 image [99](#)
 Placement Anchor [100](#)
 stretching [100, 101](#)
 text [99](#)
Unity UI System [98](#)
Update() function [39, 44, 67, 84](#)
User Experience (UX) [382, 383](#)

V

Valleys [355](#)
VFXHandler component [147](#)
Victory
 achieving [400-405](#)
Victory Conditions [400, 401](#)
Vignette [317](#)
Visual Effects (VFX) [146](#)
Visual Scripting [28, 29](#)
Visual Studio
 using [25-28](#)
Volume Rolloff [278](#)
VR/AR revolution [477, 478](#)

W

Walking Simulator [453](#)
Weapon component [249-251](#)
Weapon Locator [247](#)
WeaponObj_Sword [245](#)
Weapon Particles [331-338](#)
weapons [63](#)
Weapon system
 sticks and stones [240, 241](#)
 Weapon, equipping [247-249](#)
 Weapon Sandbox [241-247](#)
Weapon Trails [258, 259](#)
WebGL [476](#)
White-Box Prototype [56, 57](#)
WindZone object [170](#)

WindZoneSpawner [179-182](#)

workspace

 keeping clean [314, 315](#)

World Map [373](#)

 building [374-378](#)

World Space [390-395](#)

Z

Z-Brush [308](#)

Z-Fighting issues [419](#)