



2ND EDITION

Mastering UI Development with Unity

Develop engaging and immersive
user interfaces with Unity



DR. ASHLEY GODBOLD

Mastering UI Development with Unity

Copyright © 2024 Packt Publishing

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 Packt Publishing 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.

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

Group Product Manager: Rohit Rajkumar

Publishing Product Manager: Kaustubh Manglurkar

Book Project Manager: Sonam Pandey

Senior Editor: Debolina Acharyya

Technical Editor: Reenish Kulshrestha

Copy Editor: Safis Editing

Proofreader: Safis Editing

Indexer: Rekha Nair

Production Designer: Vijay Kamble

DevRel Marketing Coordinators: Anamika Singh and Nivedita Pandey

First published: April 2018

Second edition: June 2024

Production reference: 1080524

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN 978-1-80323-539-4

www.packtpub.com

To my husband, Kyle, and daughter, Claire, who are supportive of everything I do. To my pets, Doggy D. Luffy, Nami Swan, Kyo, and Mona Persona 5, who cuddle with me while I write.

- Dr. Ashley Godbold

Contributors

About the author

Dr. Ashley Godbold is a programmer, artist, author, and professor. She holds a Bachelor of Science in mathematics, a master of science in mathematics, a bachelor of science in game art and design, and a doctorate of computer science in emerging media, where her dissertation research focused on educational video game design. She is an engineer for an AA game studio and runs her own small indie studio. She teaches college courses in game programming and development, information technology, computer science, mathematics, and data science. She spends her free time watching anime and playing video games with her husband, daughter, dog, and three cats.

I want to thank the wonderful people at Packt who helped me finish this second edition and showed an incredible amount of patience for me, despite my absolute inability to stick to a deadline – especially Debolina!

About the reviewers

Dmitrii Ivashchenko is a skilled Unity game developer with over 11 years of experience in mobile gaming and backend systems. At MY.GAMES, he leads a development team dedicated to crafting engaging games, contributing to the company's extensive portfolio that reaches over 1 billion users globally. An active member of the International Game Developers Association and the Academy of Interactive Arts and Sciences, Dmitrii shares his profound Unity knowledge through articles and conference presentations. He has also served as a judge for international awards and participated in open source projects. He aims to mentor and elevate the game development community, reflecting his passion for innovation and the pursuit of excellence in game creation.

Michael Ross, a self-taught UI/UX visionary, has applied his expertise across industries that include medical, automotive, VR, and gaming through his company, Neon Raven. Known for pioneering 3D CT imaging and automotive tools, Michael aims to use his diverse experience to share his knowledge broadly and support Packt Publishing's mission. Neon Raven, where game development and UI innovation converge, thrives under Michael's leadership, which is brought to life by the exceptional artistry of Nuno Duarte, whose boundless creativity and dedication to perfection has left an indelible mark on Neon Raven's success. Together, Michael and Nuno navigate the ever-evolving realm of UI development, united by a passion for innovation and a commitment to excellence.

Table of Contents

Preface	xv
---------	----

Part 1: Designing User Interfaces

1

Designing User Interfaces		3	
Technical requirements	3	Resolution and aspect ratio	7
Defining UI and GUI	3	Changing the aspect ratio and resolution	
The four game interface types	4	of the game view	8
Laying out the UI elements	6	Building for a single resolution	14
		Summary	16

2

Designing Mobile User Interfaces		17	
Technical requirements	17	Full screen/screen portion taps	27
Setting the resolution, aspect ratio, and orientation	18	The thumb zone	28
Setting the resolution in the Game view	18	Other mobile inputs	29
The Device Simulator	19	Device-specific resources	29
Building for a specific orientation	22	Summary	30
Recommended button sizes	25		

3

Designing VR, MR, and AR UI	31
What are XR, VR, MR, and AR?	31
Designing UI for VR	33
Visual UI placement and considerations	34
Interactable UI placement and considerations	35
Designing UI for MR	36
Designing UI for AR	37
Summary	38
Further reading	38

4

Universal Design and Accessibility for UI	39
What are universal design and accessible design?	39
Universal design principles	40
Equitable use	40
Flexibility in use	41
Simple and intuitive use	42
Perceptible information	43
Tolerance of error	43
Low physical effort	44
Size and space for approach and use	45
Accessibility design	45
Vision	46
Hearing and speech	47
Mobility	47
Cognitive and emotional	48
Additional resources	48
Summary	49

5

User Interface and Input Systems in Unity	51
The three UI systems	51
Unity UI (or uGUI)	52
IMGUI	52
UI Toolkit	53
Choosing between the UI systems	53
The Input Manager	54
The new Input System	54
Choosing between the Input System and the new Input System	54
Summary	55
The two input systems	53

Part 2: Unity UI Basics

6

Canvases, Panels, and Basic Layouts	59		
Technical requirements	60	Rect Transform component	77
UI Canvas	60	Anchor and Pivot Points	79
Rect Transform component	63	Canvas Group component	83
Canvas component	64	Introducing UI Text and Image	84
Canvas Scalar component	68	Examples	85
Graphic Raycaster component	72	Laying out a basic HUD	88
Canvas Renderer component	74	Placing a 2D game background image	99
UI Panel	75	Setting up a basic pop-up menu	103
Rect Transform	76	Summary	106
Rect Tool	76		

7

Exploring Automatic Layouts	107		
Technical requirements	108	Fitters	126
Types of automatic layout groups	108	Content Size Fitter	126
Horizontal Layout Group	110	Aspect Ratio Fitter	127
Vertical Layout Group	115	Examples	129
Grid Layout Group	116	Laying out a HUD selection menu	130
Layout Element	121	Laying out a grid inventory	138
Ignore Layout	122	Summary	148
The Width and Height properties	123		

8

The Event System and Programming for UI	149		
Technical requirements	149	UnityEngine.UI namespace	150
Accessing UI elements in code	150	UI variable types	150

The Event System	152	Event Trigger	163
Event System Manager	153	Event types	164
Input Manager	154	Adding an action to the event	166
Input functions for buttons and key presses	158	Event inputs	168
GetButton	158	Raycasters	170
GetAxis	159	Graphic Raycaster	170
GetKey	159	Other Raycasters	170
GetMouseButton	160	Examples	171
Input Modules	160	Showing and hiding pop-up menus with keypress	171
Standalone Input Module	161	Pausing the game	176
BaseInputModule/PointerInputModule	162	Dragging and dropping inventory items	178
Input for multi-touch	162	Pan and zoom with mouse and multi-touch input	188
Input for accelerometer and gyroscope	162	Summary	194

Part 3: The Interactable Unity UI Components

9

The UI Button Component		197	
Technical requirements	198	Examples	208
UI Button	198	Navigating through Buttons and using First Selected	209
The Button component	200	Loading scenes with Button presses	222
Transitions	200	Button Animation Transitions	224
Navigation	205		
Invisible button zones	207	Summary	229

10

UIText and TextMeshPro		231	
Technical requirements	232	The Paragraph properties	233
UIText GameObject	232	The Color and Material properties	235
The Text and Character properties	233	The Raycast and Maskable properties	236

Text-TextMeshPro	236	Font color	255
Text Input properties	238	Font size	255
Main Settings	239	Using style sheets	256
Extra Settings	242	Translating text	258
TextMeshPro Project Settings	243	Examples	259
Working with fonts	245	Creating animated text	259
Importing new fonts	245	Translating the dialogue	267
Custom fonts	250	Custom font	273
Font assets	253	TextMeshPro - Warped Text with Gradient	281
Exploring the markup format	254	Summary	286
Font style	254		

11

UI Images and Effects **287**

Technical requirements	288	Horizontal and circular health/progress meters	295
UI Image component properties	288	Mute Buttons with sprite swap	302
Image Type	289	Adding press-and-hold/long-press functionality	308
UI effect components	292	Creating a floating eight-directional virtual analog stick	316
Shadow	292		
Outline	293		
Position As UV1	294	Summary	325
Examples	295		

12

Using Masks, Scrollbars, and Scroll Views **327**

Technical requirements	327	Implementing UI Scroll View	334
Using masks	328	Scroll Rect component	337
The Mask component	328	Examples	342
Rect Mask 2D component	329	Making a scroll view from a pre-existing menu	342
Implementing UI Scrollbars	330		
The Scrollbar component	331	Summary	348

13

Other Interactable UI Components		349	
Technical requirements	350	The Dropdown component	362
Using UI Toggle	350	UI Input Field	365
Toggle component	350	Input Field component	366
Toggle Group component	354	Input Field - TextMeshPro	377
UI Slider	355	TextMeshPro - Input Field component	377
Slider component	355	Examples	380
UI Dropdown and Dropdown – TextMeshPro	358	Creating a dropdown menu with images	380
Dropdown Template	359	Summary	390

Part 4: Unity UI Advanced Topics

14

Animating UI Elements		393	
Technical requirements	394	Setting Animation Parameters in scripts	407
Animation Clips	394	Animator Behaviours	409
Animation Events	398	Animating pop-up windows to fade in and out	411
Animator Controller	399	Animating a complex loot box	428
The Animator of Transition Animations	405	Summary	456
Animator layers	406		

15

Particles in the UI		457	
Technical requirements	457	Creating a Particle System that displays in the UI	459
Particles in the UI	457	Summary	468
Examples	459		

16

Utilizing World Space UI 469

Technical requirements	469	Examples	477
When to use World Space UI	469	2D World Space status indicators	478
Appropriately scaling text in the Canvas	473	3D hovering health bars	481
Other considerations when working in World Space	477	Summary	487

17

Optimizing Unity UI 489

Optimization basics	490	Using multiple Canvases and Canvas Hierarchies	495
Frame Rate	490	Minimizing the use of Layout Groups	495
GPU and CPU	490	Hiding objects appropriately	496
Tools for determining performance	491	Appropriately time object pooling enabling and disabling	496
Statistics window	491	Reducing Raycast computations	497
Unity Profiler	492	Summary	498
Unity Frame Debugger	494	Further reading	498
Basic Unity UI Optimization Strategies	495		

Part 5: Other UI and Input Systems

18

Getting Started with UI Toolkit 501

Technical requirements	502	Visual Elements and UI Hierarchy	509
Overview of UI Toolkit	502	Creating UI with the UI Builder	511
Installing the UI Toolkit package	503	Using the UI Document component	513
Parts of the UI Toolkit system	506	Making The UI interactable with C# The UIElements namespaces	513

Getting a reference to UI Documents variables	514	Using the UI Toolkit to make an Editor virtual pet	517
Managing Visual Element events	516	Resources	563
Accessing Visual Element properties	516	Summary	564
Examples	517		

19

Working with IMGUI	565		
Technical requirements	566	Examples	571
IMGUI overview	566	Using IMGUI to show framerate in-game	571
IMGUI Controls	567	Using IMGUI to make an Inspector	
IMGUI in the Inspector	571	button that imports data	573
		Summary	578

20

The New Input System	579		
Technical requirements	580	Creating basic character controller Actions	591
Installing the Input System	580	Creating a basic character controller	
Polling vs subscribing	581	with the PlayerInput Component	595
Input System elements	583	Creating a basic character controller	
Connecting Actions to code	586	by referencing Actions in your code	600
		Summary	602

Index	603
--------------	------------

Other Books You May Enjoy	614
----------------------------------	------------

Preface

There are a multitude of built-in UI elements that can be incorporated into a game built in Unity. This book will help you master Unity's UI system by describing, in-depth, the various UI objects, functionalities, and properties and providing step-by-step examples of their implementation.

Who this book is for

This book is intended for game developers who have worked with Unity and are looking to improve their knowledge of the UI systems provided within Unity. Individuals looking for in-depth explanations of specific UI elements, as well as individuals looking for step-by-step directions explaining how to implement UI items that appear in multiple game genres, will also find this book helpful. A basic understanding of Unity and C# programming is needed.

What this book covers

Chapter 1, Designing User Interfaces, covers basic information related to designing user interfaces. The distinction between a graphic user interface and user interface is defined. It discusses the four different types of game interfaces, how to create an aesthetically pleasing UI based on principles of design, and the concept of interface metaphors. Additionally, a detailed explanation of setting the aspect ratio and resolution of a Unity project is discussed.

Chapter 2, Designing Mobile User Interfaces, covers considerations that a UI designer must take into account when developing for mobile, both aesthetically and mechanically. Additionally, it discusses the resources available to developers to help them design mobile user interfaces.

Chapter 3, Designing VR, MR, and AR UI, covers the basic concepts of designing user interfaces for VR, MR, and AR applications. It looks at how interactions in these applications differ from other applications and discusses the best practices for designing them.

Chapter 4, Universal Design and Accessibility for UI, covers basic concepts related to designing user interfaces so that they can be used by the greatest number of people. This chapter will explore the topic of universal design and designing for accessibility while discussing steps that a UI designer can take to make sure their user interfaces are as barrier-free as possible.

Chapter 5, User Interface and Input Systems in Unity, reviews the various systems provided by Unity to work with UI. Unity provides three systems for designing user interfaces and two systems for controlling the inputs. This chapter explores the various systems, compares their benefits, and discusses when to use which one of them.

Chapter 6, Canvases, Panels, and Basic Layouts, explores the development of a user interface by appropriately laying out UI elements within a Canvas. Panels are used, and an introduction to Text and Images is also provided. The examples included in this chapter show you how to lay out a basic heads up display, create a permanent background image, and develop a basic pop-up menu.

Chapter 7, Exploring Automatic Layouts, discusses how to implement the various automatic layout components to streamline the UI building process. The examples included within this chapter utilize the automatic layout functionality to create a selection menu in the HUD and a gridded inventory.

Chapter 8, The Event System and Programming for UI, covers the event system and how it pertains to the UI. How to add Event Triggers to UI elements is discussed. It covers the keywords necessary to program for the UI system, how to access UI components via code, and how to write functions that can be accessed via Event Triggers.

Chapter 9, The UI Button Component, explores the various properties of buttons. The examples in this chapter walk through how to set up keyboard and controller navigation of buttons, how to load scenes when buttons are pressed, how to create animated button transitions, and how to make buttons swap their images.

Chapter 10, UI Text and Text-TextMeshPro, discusses the properties of text more thoroughly and demonstrates how to affect their properties via code. The examples at the end of the chapter show how to create a dialog box with text that animates as if it were being typed in, how to create a custom font, and how to create text that wraps with a gradient.

Chapter 11, UI Images and Effects, shows more ways in which UI images can be used and manipulated. Additionally, it demonstrates how to apply various effects to UI elements.

Chapter 12, Using Masks, Scrollbars, and Scroll Views, covers how to create scrollable windows with masks so that your UI can hold more items than are immediately in view.

Chapter 13, Other Interactable UI Components, covers a myriad of other UI inputs. Examples of how to use the various inputs and how to create a dropdown menu with images are covered at the end of the chapter.

Chapter 14, Animating UI Elements, is all about animating the UI. The examples in this chapter show how to animate menus to fade in and out and how to create a treasure box animation using the Unity State Machine.

Chapter 15, Particles in the UI, expands upon the animation example of the previous chapter and provides further ways in which you can zhuzh up your UI using particle effects.

Chapter 16, Utilizing World Space UI, showcases how to create UI elements that exist within the game scene as opposed to on the “screen” in front of all in-game items. The examples cover how to create an interactive UI for a 2D scene and interactive, hover health bars for a 3D scene.

Chapter 17, Optimizing Unity UI, covers basic concepts of creating optimized user interfaces. It defines key terms, provides an overview of tools included within Unity that can help you determine how performant your game is, and covers various optimization strategies for working with Unity’s UI system.

Chapter 18, Getting Started with UI Toolkit, covers the new Unity UI Toolkit and explains how to use it to create basic layouts. It covers the key concepts of using this different UI system, while also demonstrating how to use it to create both Editor and runtime UI.

Chapter 19, Working with IMGUI, discusses how to use the IMGUI system to build user interfaces. After exploring the basic concepts of IMGUI, the chapter covers how to use the system to create developer tools that appear in the Editor and at runtime.

Chapter 20, The New Input System, provides an introduction to using the new Input System for easy input setup. It covers the publisher-subscriber architectural pattern while introducing the basic concepts and principles of the Input System. Additionally, it covers how to update a project that uses the Input Manager to one that uses the new Input System, as well as how to connect your code to the Input System in two different ways.

To get the most out of this book

This book assumes you have a good understanding of navigating within and working with the Unity Editor. Additionally, it assumes you have a basic understanding of programming in C# for Unity.

Software/hardware covered in the book	Operating system requirements
Unity 2020 or higher	Windows, macOS, or Linux

In addition to having Unity installed, you will also need a code editor (IDE). While no preference is established in this book, examples of supported IDEs are Visual Studio and JetBrains Rider.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “Currently, there is a bit of a problem with the `AnimationComplete` trigger.”

A block of code is set as follows:

```
[System.Serializable]
public class Translation {
    public string languageKey;
    public string translatedString;
    public Font font;
    public FontStyle fontStyle;
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Vector2[] newPositions = new Vector2[] { Input.GetTouch(0).position,
Input.GetTouch(1).position};
```

Bold: Indicates a new term, an important word, or words that you see on screen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “When **Permanent** is selected for the **Visibility** property, the respective scrollbar will remain visible, even if it is not needed, if its corresponding movement is allowed.”

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

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 copyright@packt.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 authors.packtpub.com.

Share Your Thoughts

Once you've read *Mastering UI Development with Unity*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your e-book purchase not compatible with the device of your choice?

Don't worry!, Now with every Packt book, you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the following link:



<https://packt.link/free-ebook/9781803235394>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

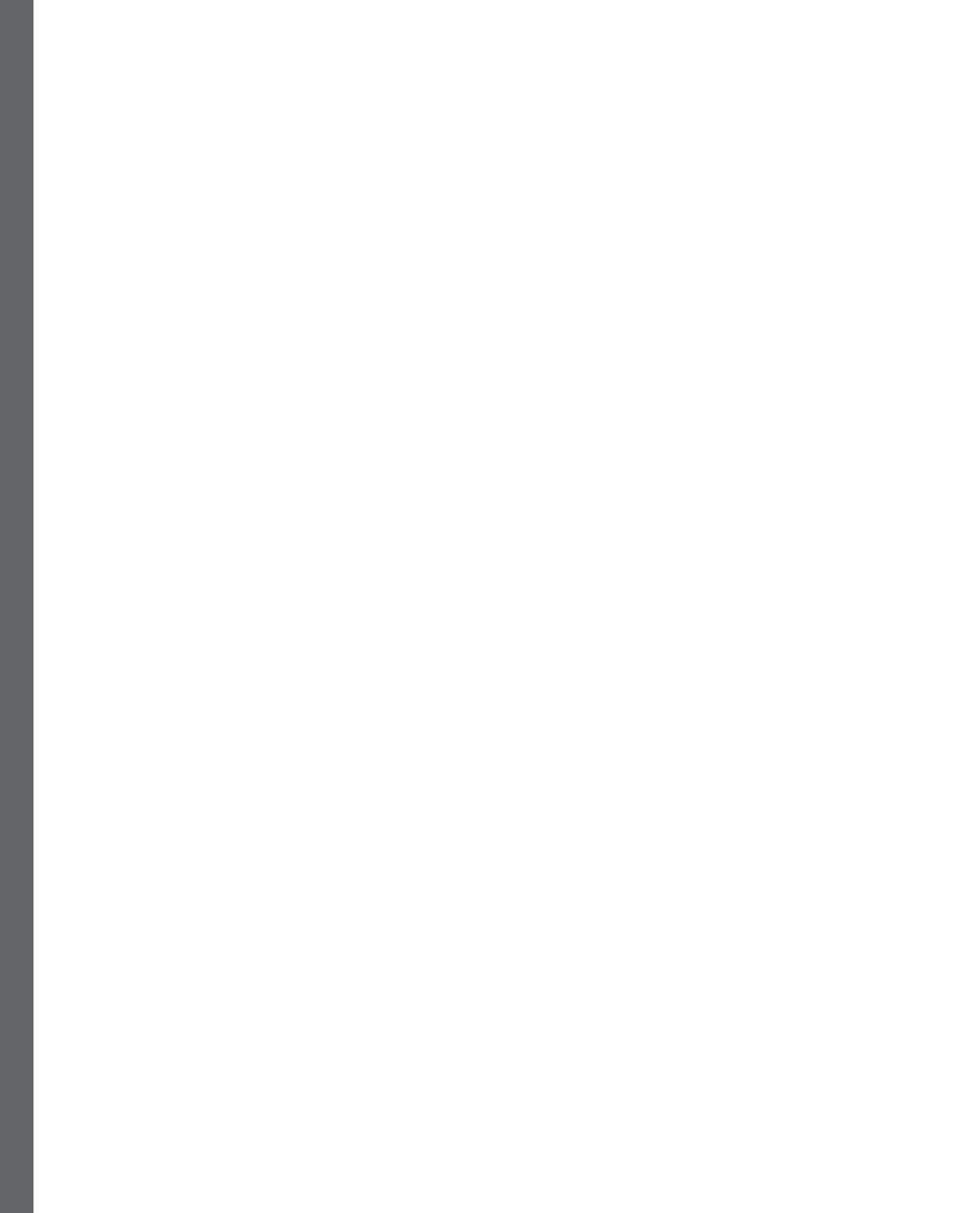
Part 1:

Designing User Interfaces

In this part, you will get a general overview of design principles to consider while designing user interfaces for various platforms. How to design UI for the special cases of mobile games and XR experiences is explored. Additionally, how to design UI for all platforms with consideration for universal design and accessibility is covered. Lastly, the various systems within Unity that facilitate the development of UI are compared and discussed.

This part has the following chapters:

- *Chapter 1, Designing User Interfaces*
- *Chapter 2, Designing Mobile User Interfaces*
- *Chapter 3, Designing VR, MR, and AR UI*
- *Chapter 4, Universal Design and Accessibility for UI*
- *Chapter 5, User Interface and Input Systems in Unity*



1

Designing User Interfaces

When working with UI, it is important to understand a few design basics. This chapter will cover the foundation of designing UI and a few key concepts to start you off in the right direction.

In this chapter, we will discuss the following topics:

- Defining UI and GUI
- Describing the four types of interfaces
- Laying out your user interfaces
- Discerning and setting resolution and aspect ratio

This book is not about the art of designing UI. It is a technical text that discusses the implementation of UI functionality. However, I do want to discuss some basic design principles of UI design. I don't expect you to be an amazing UI designer after reading this chapter. I do hope that you get some basic understanding of layout and design principles from this chapter, though, so that maybe your artist friends won't make too much fun of you.

Technical requirements

For this chapter, you will need the following:

Unity 2020.3.26f1 or later

Defining UI and GUI

So, what exactly do UI and GUI stand for, and what's the difference? **UI** stands for **user interface** and **GUI** (pronounced “gooey”) stands for **graphical user interface**. To *interface* means to *interact with*, so the UI is the set of devices that let a player interact with a game. The mouse, keyboard, game controller, touch screen, and so on are all part of the UI. The GUI is the subset of the UI represented by graphics. So, onscreen buttons, dropdown menus, and icons are all part of a game's GUI. As the

GUI is a subset of the UI, many people (myself included) tend to just refer to the GUI as the UI. Unity also refers to all the GUI items they provide templates for as the UI.

This book will focus primarily on GUI design, but it will discuss some non-graphical aspects of UI controls, such as accessing data from the mouse, screen tap, keyboard, or controller. This chapter specifically will look at some basic design considerations for different interface types.

The four game interface types

When you say “game UI,” most people think of the **heads-up display (HUD)** that appears in front of all the in-game items. However, there are actually four different types of game interfaces: non-diegetic, diegetic, meta, and spatial.

Fagerholt and Lorentzon first described these four different interface types in the 2009 paper *Beyond the HUD: User Interfaces for Increased Player Immersion in FPS Games: Master of Science Thesis*. Since then, the terminology has been widely used throughout the field of UI game design. You can find the original publication at <http://publications.lib.chalmers.se/records/fulltext/111921.pdf>.

The distinction between the four is determined by a cross of the following two dimensions:

- **Diegesis:** Is it part of the story?
- **Spatiality:** Is it in the game’s environment?

The following diagram demonstrates the cross relationship between the two questions and how they define the four types of interfaces:

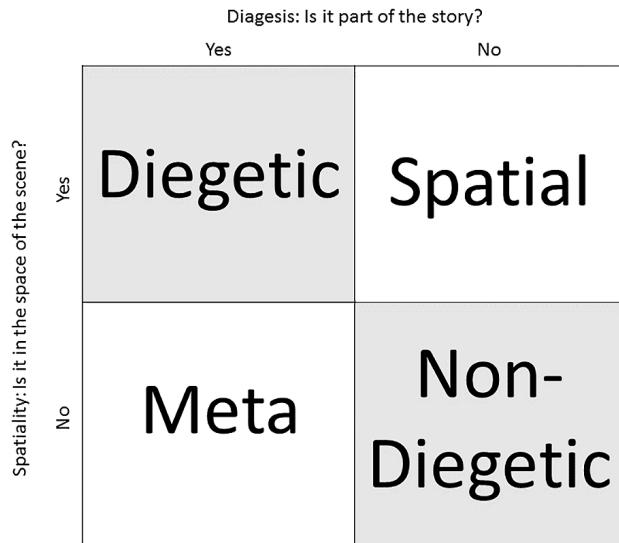


Figure 1.1: Four types of interfaces

A game's HUD falls into the **non-diegetic** category. This information exists purely for the player to view and the characters within the game are not aware of its presence. It exists on the *fourth* wall of the game view and appears to be on the screen in front of everything. The examples of this type of UI are endless, as nearly every game has some non-diegetic UI elements.

Alternatively, a **diegetic** interface is one that exists within the game world and the characters within the game are aware of its presence. Common examples of this include characters looking at inventory or maps. The most widely referred-to example of diegetic UI is the inventory and health display within *Deadspace*. The inventory displays on a holographic display window that pops up in front of the playable character, and he interacts with it as you select his weaponry. His health is also indicated by a meter on his back. The inventory of *Alone in the Dark* (2008) is displayed in a diegetic way as well. While there are some UI elements that only the player can see, the main character views inventory within their jacket pockets and interacts with the items. *Uncharted Lost Legacy* and *Far Cry 2* both use maps that the characters physically hold in the scene and interact with. *Fallout 3* and *Fallout 4* use a diegetic interface to display the inventory and map on the character's Pip-Boy, which is permanently attached to their arm. Games also use this type of display when characters are in a vehicle or suit, where various displays appear on the shield, window, or cockpit.

Meta interfaces are interfaces that the characters in the game are aware of, but they are not physically displayed within the scene. Common examples of this are speed displays for racing games. *Forza 7* actually uses a combination of meta and diegetic displays for the speedometer. A meta speed indicator is persistently on the lower-right corner of the screen for the player to see. Since the character is constantly aware of how fast they are driving, they would be aware of this speed indicator, therefore making it a meta interface. There is also a diegetic speedometer in the car's dash that is displayed when playing in first-person view. Another common usage of this type of display is a cell phone that appears on the screen but is implied the playable character is interacting with. *Persona 5*, *Catherine*, and *Grand Theft Auto 5* all use this interface type for cell phone interactions.

The last type of interface, **spatial**, exists in the scene, but the characters within the game are not aware of it. Interfaces that exist in the scene but that the characters are not aware of are incredibly common. This is commonly used to let the player know where in the scene interactable items are, what the in-game character is doing, or information about characters and items in the scene. For example, in *Legend of Zelda: Breath of the Wild*, arrows appear over the heads of enemies, indicating who Link will attack. Link is not actually aware of these arrow icons; they are there for the player to know who he is focusing on. *Xenoblade Chronicles 2* uses a spatial interface to indicate where the player can dig by displaying a shovel icon over the diggable areas.

Laying out the UI elements

When laying out the UI for your game, I strongly recommend checking other games of the same genre and seeing how they implemented their UI. Play the game and see whether it feels good to you.

If you are unsure of how to lay out your game's UI, I recommend dividing the game's screen into a *guttered grid*, like the one shown in the following diagram, and placing items within the non-guttered areas:

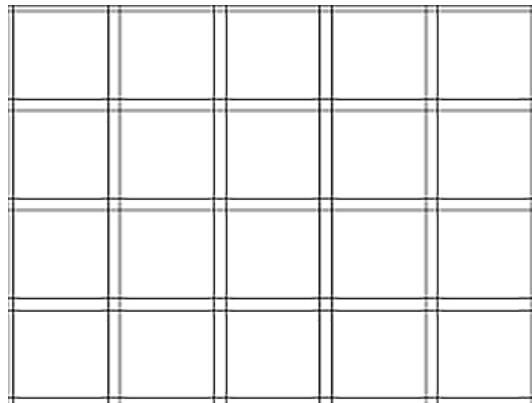


Figure 1.2: A guttered grid

You can use as many grids as you want, but laying out the items with reference to the grid will help ensure that the UI is arranged in a balanced way.

In most cases, the HUD items should remain at the outer edges of the grid. Any UI that displays in the center grids will restrict the player view. So, this area is good for pop-up windows that pause the gameplay.

The device your game will be played on is important when determining the layout. If your game is designed for a mobile device and has a lot of buttons the player will interact with, the buttons are generally best suited for the bottom or side portions of the screen. This is due to the way players hold their phones and the top-center part of the screen is the most difficult area to reach with their thumb. Additionally, reaching for this area will cause them to block the majority of the game view with their hand. We will discuss designing UI for mobile more thoroughly in *Chapter 2*.

You'll note that when you play computer games, they tend to have much smaller and more cluttered UI than mobile and console games. This is due to visibility and interaction. Clicking on small objects with a mouse is significantly easier than tapping them with a finger or selecting them with the D-pad. Also, the screen resolution is much bigger, which allows for more space to be taken up by the UI.

When trying to determine the size and relative location of UI items, you can reference **Fitts' Law**. Fitts' Law can mathematically calculate how long it will take a user to navigate to a UI item based on its size and distance away from the user's starting position. I won't go over the math here (despite the math teacher in me desperately wanting to), but the lessons that can be garnered from Fitts' Law are as follows:

- Don't make interactable UI small and far apart
- Make the most important interactable items the largest and near each other

Next, we'll look at resolution and aspect ratio.

Resolution and aspect ratio

A game's **resolution** is the pixel dimension of the screen on which it plays. For example, a game could run at 1,024x768. This means that the game is 1,024 pixels wide and 768 pixels tall. The **aspect ratio** of a game is the ratio of the width and height (expressed as width:height). This aspect ratio is determined by dividing the resolution width by the resolution height and then simplifying the fraction. So, for example, if your game has a resolution of 1024x768, the aspect ratio would be as follows:

$$1024px/768px=4/3$$

Here, the fraction 4/3 is the aspect ratio 4:3.

The following table provides a list of common aspect ratios and related resolutions:

Aspect Ratio	Resolution			
3:2	720x480	1280x854	1440x960	2880x1920
	1152x768			
4:3	640x480	1024x768	1440x1080	1920x1440
	800x600	1280x960	1600x1200	2048x1536
	960x720	1400x1050	1856x1392	
5:3	1280x768	3000x1800		
5:4	1280x1024	2560x2048	5120x4096	
16:9	1024x576	1280x720	1600x900	2560x1440
	1152x648	1366x768	1920x1080	3840x2160
16:10	640x400	1440x900	1920x1200	3840x2400
	1280x800	1680x1050	2560x1600	

Figure 1.3: Common aspect ratios and resolutions

When designing your UI, the resolution and aspect ratio will play an important role in how your UI will look. Knowing the resolution and aspect ratio of your target device will be an important first step in designing your UI for two reasons:

- It will determine the layout of your UI
- The way you build the UI within Unity will be determined by how many resolutions and aspect ratios you plan to support

If you build to a single resolution/aspect ratio, the UI will be much easier to build as you won't have to make sure all the elements maintain their relative position at multiple aspect ratios. However, if you build a game that will run at multiple resolutions/aspect ratios (for example, a mobile project or a web game that scales within a window), you want your UI to scale and move appropriately. You'll also want to be able to easily change the resolution during testing so that you can make sure the UI is positioned appropriately as its display window morphs.

Even if you will allow your resolution and aspect ratio to vary, you should still decide on a default resolution. This default resolution represents the resolution of your ideal design. This will be the resolution that your initial design and UI layout are based on, so if the resolution or aspect ratio varies, the UI will try to maintain the same design as best it can.

Note

Since all televisions sold today have a 16:9 aspect ratio, any UI you make for a console game should be developed with a 16:9 aspect ratio in mind.

Changing the aspect ratio and resolution of the game view

You can easily switch between different resolutions and aspect ratios in the **Game** tab. This will allow you to see how your UI scales at the different resolutions and aspect ratios:

1. If you navigate to your **Game** tab, you will see the words **Free Aspect**. Clicking on **Free Aspect** will reveal a menu that shows various aspect ratios and resolutions:

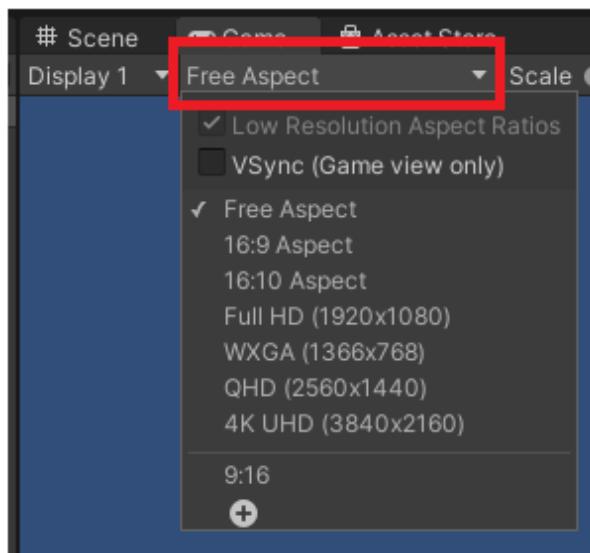


Figure 1.4: Selecting Free Aspect mode from the Game view

The items displayed in this list are the most common aspect ratios and resolutions for the build target you currently have selected. In the preceding screenshot, my build target was **PC, Mac & Linux Standalone**, so the most common monitor settings are displayed. If I were to change my build target to iOS, I would see a list of popular iPhone and iPad screen dimensions.

Free Aspect means that the game's aspect ratio will scale relative to the window of the **Game** view. So, by moving the frame around on the **Game** window, you will change the aspect ratio.

2. You can easily see the effects of **Free Aspect** on your game's aspect ratio, by setting your Editor's layout to one that shows both the **Screen** and **Game** tabs open simultaneously. For example, setting **Layout** to **2 by 3** will do this. Select the **Layout** dropdown in the top-right corner of the Unity Editor to change the layout.

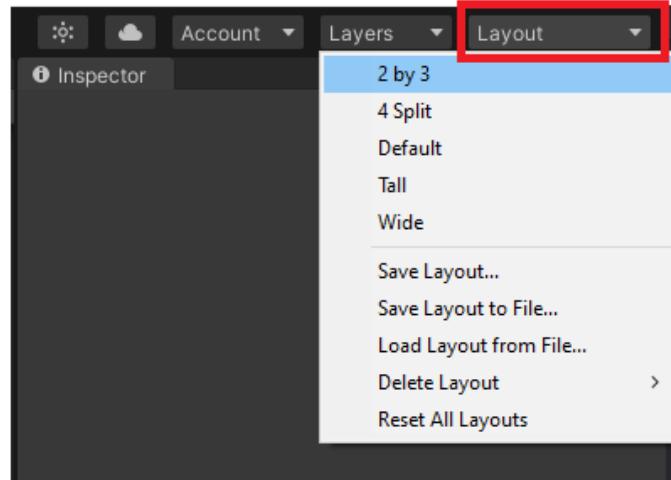


Figure 1.5: Changing the Editor Layout

Now the **Game** and **Scene** tabs will both be visible on the left-hand side of your screen.

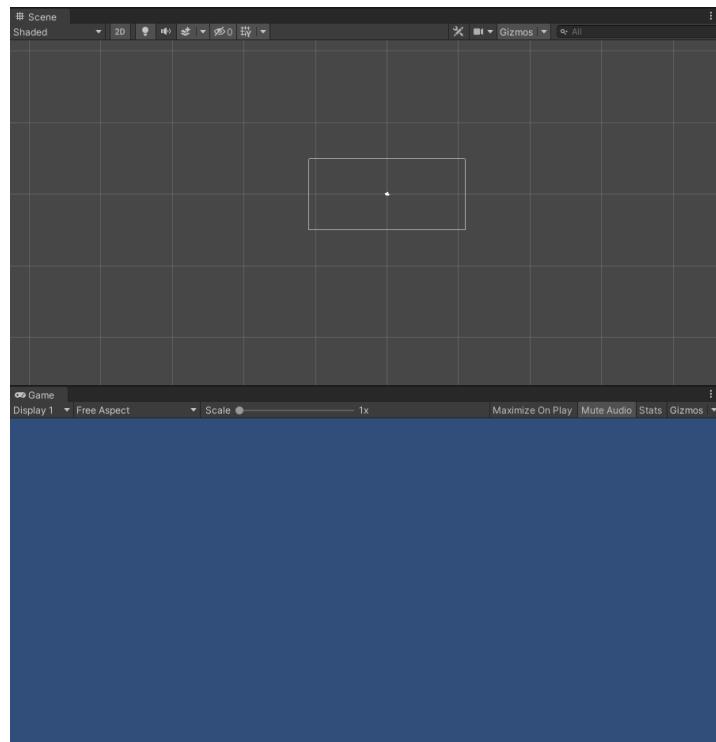


Figure 1.6: Results of the 2 by 3 layout

- Now, reduce the size of the **Game** tab so that it is a very small thin rectangle. You will see that the main camera in the Scene view is now also displaying as a very small thin rectangle:

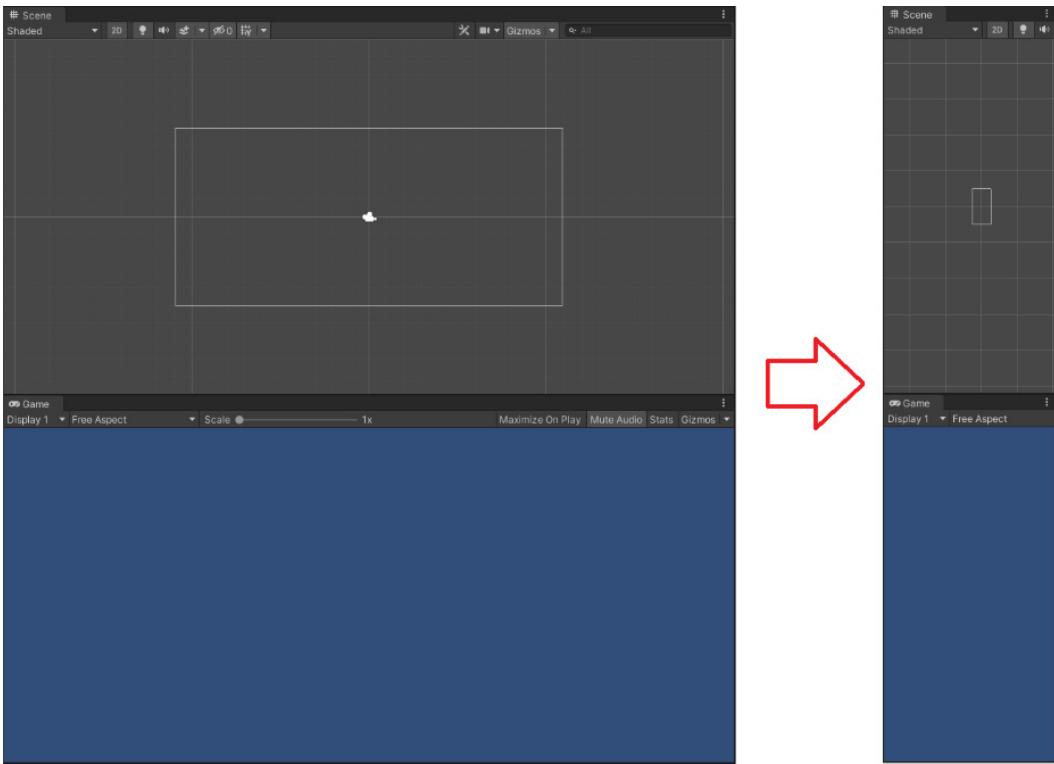


Figure 1.7: Results of resizing the Game view in Free Aspect mode

- You can select one of the aspect ratios in the dropdown and see that, as you rescale the game window, the blue area representing the actual game will maintain the ratio you selected and black bars will fill in any extra spacing. The camera will also maintain that ratio.
- Full HD (1920x1080)** will attempt to emulate the 1,920x1,080 resolution. It's pretty likely that the window you have set for the **Game** tab is not big enough to support 1,920x1,080 pixels; if so, it will be scaled as indicated in the following screenshot:

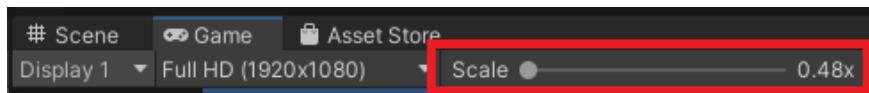


Figure 1.8: Game view scale

6. If the resolution or aspect ratio you want to use is not available in the resolution dropdown menu, you can add your own item to this menu by selecting the plus sign at the bottom of the dropdown. If you want to create a set resolution item, set **Type** to **Fixed Resolution**. If you want to create a set aspect ratio item, set **Type** to **Aspect Ratio**.

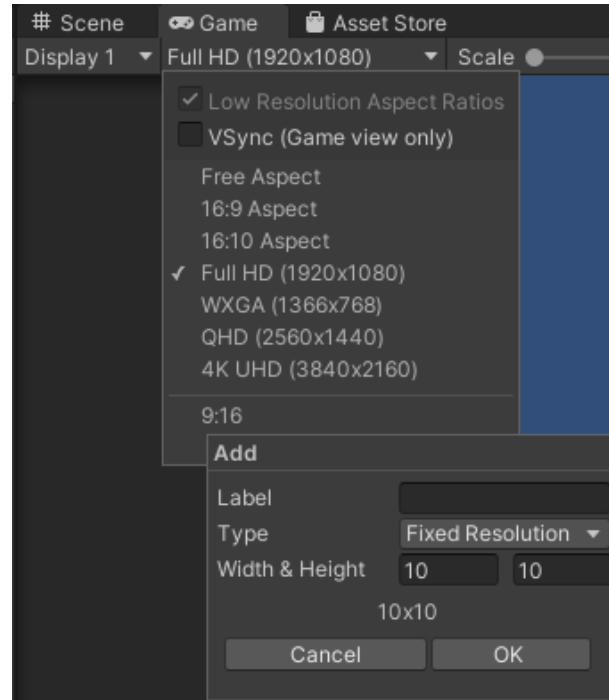


Figure 1.9: Adding a new resolution or aspect ratio preset

For example, if you wanted to make a game that was reminiscent of an old Game Boy game, you could add a 160x144 pixels preset:

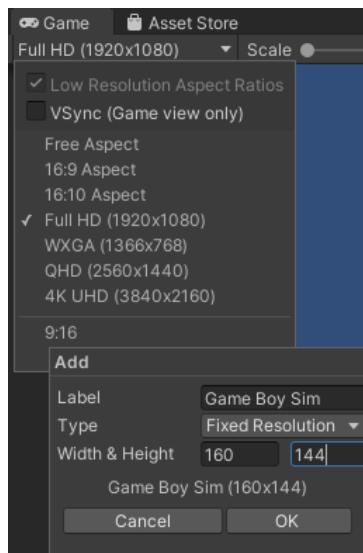


Figure 1.10: Creating a fixed resolution preset

- Once you hit **OK**, the new preset will item will be displayed at the bottom of the list. When you select it, the camera and visible area of the **Game** tab will maintain the aspect ratio created by a 160x144 resolution:

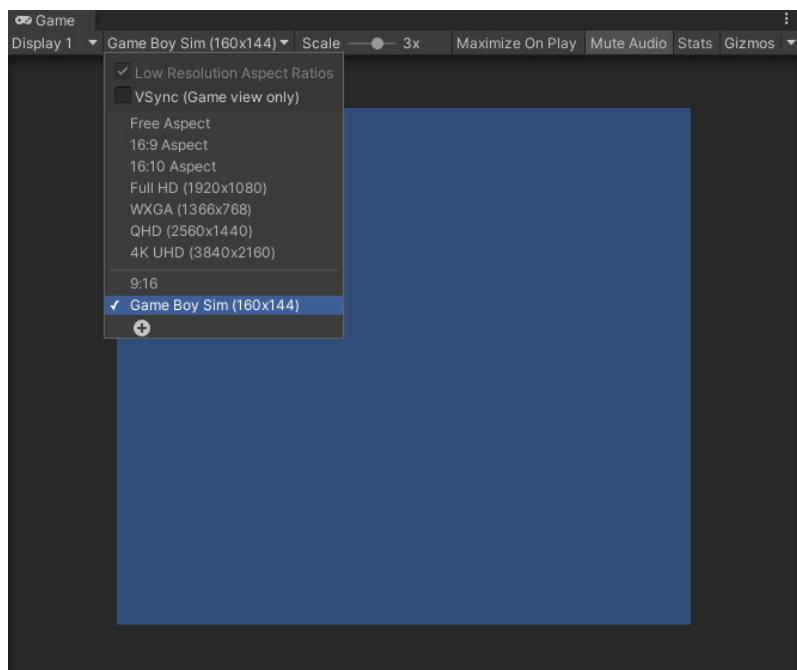


Figure 1.11: Selecting a custom preset

Building for a single resolution

If you are creating a game that you plan to build on the **PC, Mac, & Linux Standalone** target platform, you can force the resolution to always be the same. To do so, go to **Edit | Project Settings | Player**. Your Inspector should now display the following:

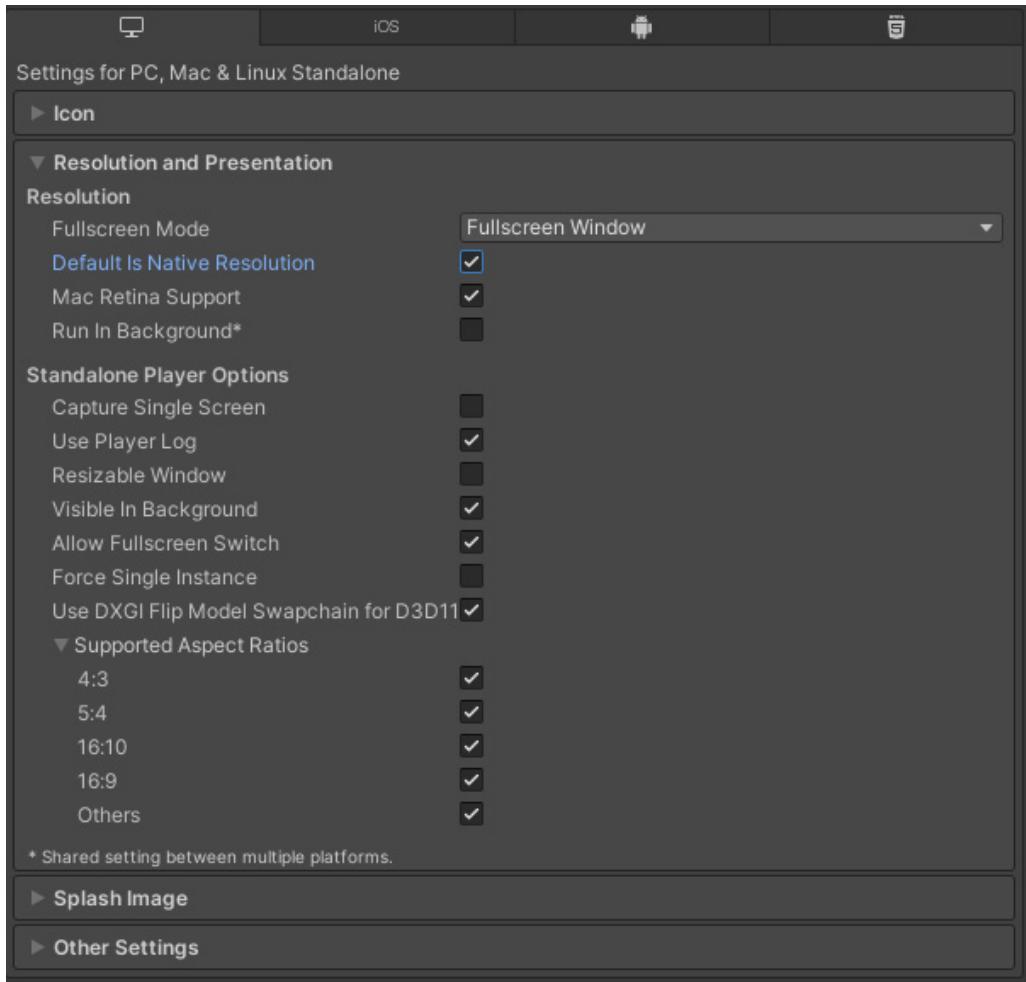


Figure 1.12: PC, Mac & Linux Standalone Player resolution settings

You may have more or fewer platforms displayed here; it depends on the modules you have installed with Unity.

To force a specific resolution on a **PC, Mac, & Linux Standalone** game, deselect **Default is Native Resolution**. The option to input **Default Screen Width** and **Default Screen Height** will be made available to you and you can enter the desired resolution values. Then, when you build your game, it will play at the size you specified.

The following screenshot shows the settings for forcing a PC game to play in a window with Game Boy Color dimensions:

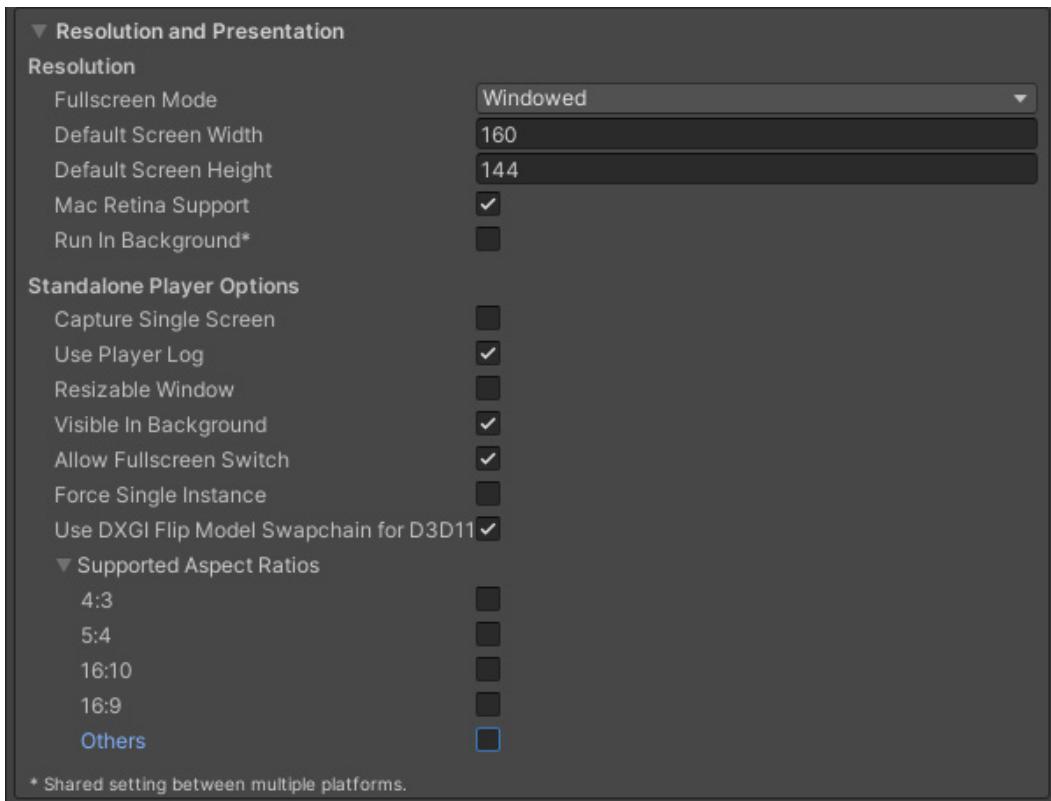


Figure 1.13: Setting a specific PC, Mac, & Linux Standalone Player resolution

You can also force a specific resolution with a **WebGL** build. There are fewer options to worry about, but the general concept is the same. The following screenshot shows the settings for forcing your game to display at **160x140** in the **Player Settings** for **WebGL**:

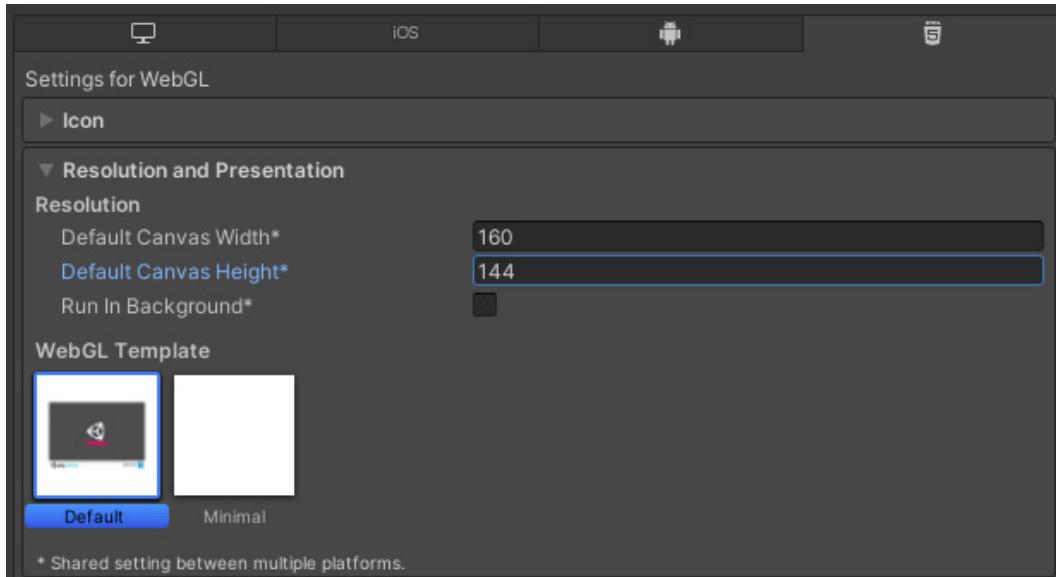


Figure 1.14: Setting a specific WebGL resolution

In *Chapter 2*, we will discuss how to set the resolution properties for mobile games that have varying resolutions that you cannot pre-define.

Summary

This chapter discussed some basic design principles and terminology related to UI. You should now be able to distinguish between GUI and UI and define the four types of interfaces: diegetic, spatial, meta, and non-diegetic. Additionally, you should understand some basic rules of laying out UI and how to work in different resolutions and aspect ratios.

The next chapter will expand upon these design principles and look at some important considerations for designing UI for mobile games.

2

Designing Mobile User Interfaces

One of the most challenging aspects of designing for mobile interfaces is the sheer amount of possible aspect ratios of mobile devices. Mobile devices include phones and tablets. Phones tend to be much longer than tablet devices, and a UI that fits perfectly on a phone may overlap or look squished when put on a tablet. On top of awkward resolutions, there are also weird quirks that will affect your UI, such as the notch in the iPhone and the folding screens of the various Samsung Galaxy devices.

Mobile devices also have a different set of inputs and interactions. For example, on a mobile device, you can touch the screen at multiple locations simultaneously, but on a PC, you can only click with your mouse in one place at a time. Mobile devices require on-screen keyboards and other peripherals to perform the same actions that you might perform on a console or PC game.

In this chapter, I will discuss the design considerations around the various quirks of mobile devices and cover the following topics:

- How to simulate your game at various mobile resolutions and build at a specific orientation
- The recommended button sizes for mobile games
- Utilizing invisible buttons to create tap areas
- Laying out interactions based on the thumb zone
- How multi-touch input plays a part in mobile UI
- When to use the accelerometer and gyroscope

Technical requirements

For this chapter, you will need Unity 2020.3.26f1 or later.

Note

When describing mobile interfaces, I will mostly focus on iOS and Android operating system phones and tablets. However, occasionally I will reference Microsoft, as they do create a series of tablet devices that, despite running Windows operating systems, do have touch capabilities.

Setting the resolution, aspect ratio, and orientation

When you design a mobile UI, you'll want to make sure it makes sense and is visible at various resolution sizes and aspect ratios. You also may want to allow for different screen orientations. In *Chapter 6*, I will discuss how you can develop user interfaces that scale to multiple resolutions and layouts, mechanically. But for now, let's just review how resolution, aspect ratio, and orientation can affect your design and how to view your game with the various screen settings.

Setting the resolution in the Game view

Recall in *Chapter 1*, we looked at how you can change the resolution and aspect ratio of your **Game** view. When you change your game's build settings to iOS or Android, a new list of presets will be provided to you. For example, in the following image, you can see the list of possibilities when the **Build** is set to iOS:

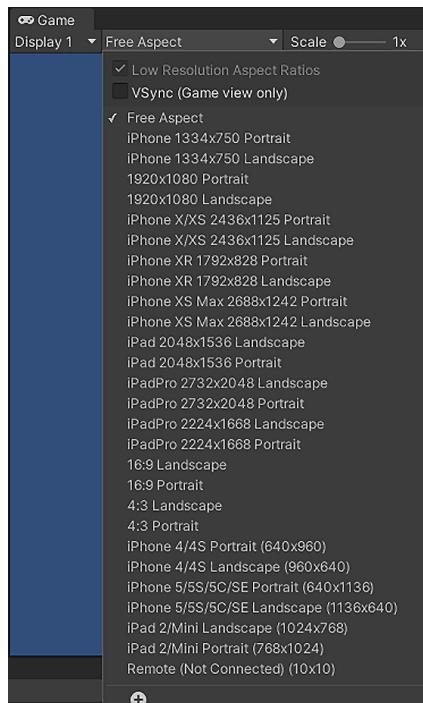


Figure 2.1: iOS resolutions in Game view

It will not show all the possible iOS resolutions, but many of the common newer ones. You can find a more complete list at <https://www.ios-resolution.com/>. Similarly, not all Android resolutions will be listed when switching to an Android build, especially considering there are significantly more Android resolutions. However, you can find more information about Android screen resolutions at https://developer.android.com/guide/practices/screens_support.html#testing.

The Device Simulator

Sometimes, setting the **Game** view's aspect isn't sufficient to fully see how your UI will appear on a device. For example, the controversial notch introduced in the iPhone X doesn't display in the **Game** view and can throw a huge wrench in your most carefully designed UIs. However, you can see this notch and how it overlaps with your UI using the **Device Simulator**.

To enable the Device Simulator, complete the following steps:

1. Download the Device Simulator Package from **Package Manager**.

To do so, go to **Window | Package Manager** and search for **Device Simulator** in the list. It's highly likely you won't see it in the list initially. If you do not see it, make sure that the packages being displayed are those from **Unity Registry**, as shown in the following screenshot:

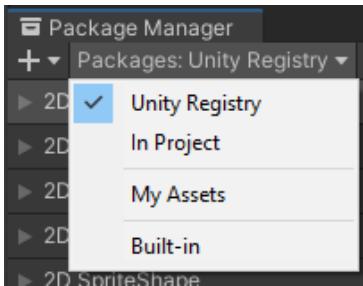


Figure 2.2: Unity Registry packages

2. If you still do not see it in the list, you have to enable preview packages. Click on the settings cog in the top-right corner, select **Advanced Project Settings**, and then select **Enable Preview Packages**, as shown in *Figure 2.3*:

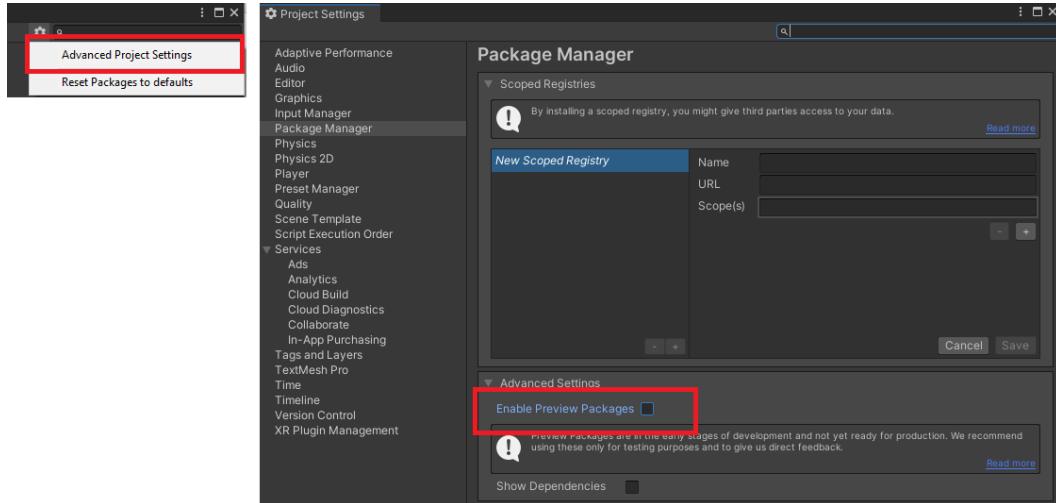


Figure 2.3: Enable Preview Packages

3. Once you locate **Device Simulator** in the list, install it:



Figure 2.4: Install Device Simulator

- Now that the **Device Simulator** is downloaded, you can change your **Game** view to reflect a simulator by selecting the **Game** dropdown and then selecting **Simulator**, as shown in the following screenshot:

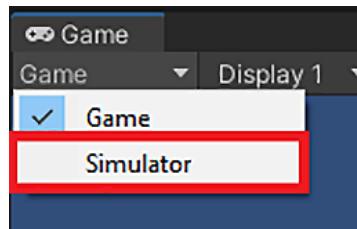


Figure 2.5: Select Simulator view

- Now you can select multiple devices from the dropdown to simulate. For example, I can select an iPad 5th generation or an iPhone X, where the iPhone X shows the dreaded notch:

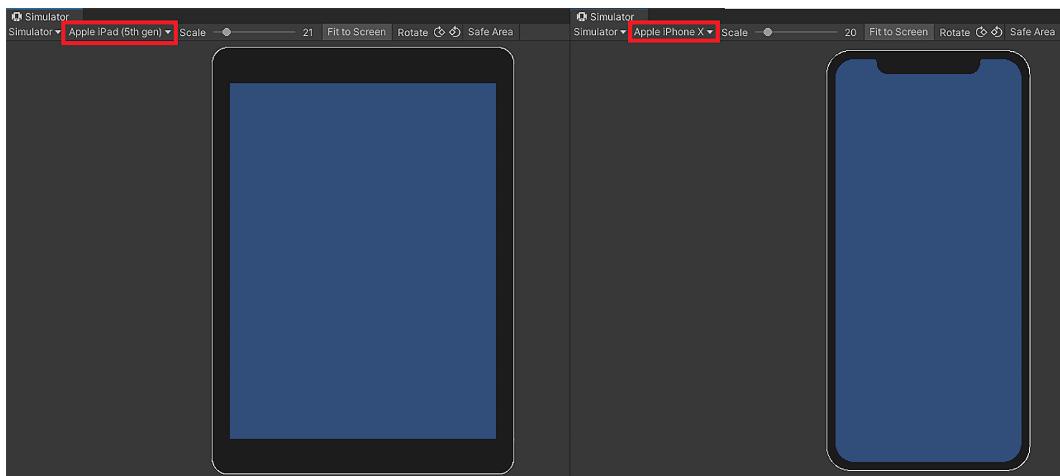


Figure 2.6: Different devices on Simulator

6. Additionally, you can select **Safe Area** to see the best places to lay out your UI:

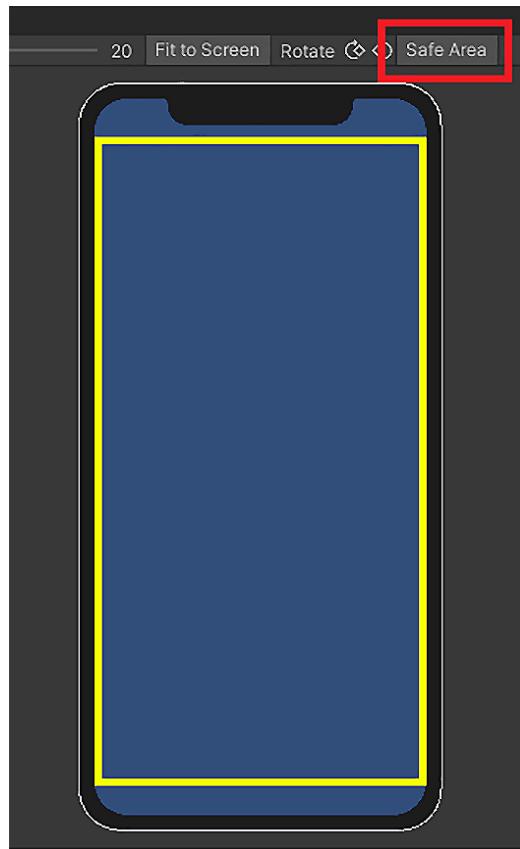


Figure 2.7: Safe Area

Building for a specific orientation

You may have noticed in *Figure 2.1* that there are both landscape and portrait options for each resolution. This allows you to view your game depending on which orientation you want to support while developing.

When building for mobile devices, you can't specify resolution and aspect ratio and will instead have to support all resolutions and aspect ratios. However, you can choose between screen orientations on mobile devices. There are two different orientations: **Landscape** and **Portrait**.

Games built so that they are wider than they are tall are said to have landscape resolution. Games built taller than they are wide are said to have portrait resolution. For example, a **16:9** aspect ratio would be a landscape resolution, and a **9:16** aspect ratio would be a portrait resolution, as illustrated:



Figure 2.8: Landscape versus portrait Orientation

So, while you can't choose the exact aspect ratio your mobile game will build to, you can choose the orientation, which forces the aspect ratio to be either wider or taller. You can set the orientation by navigating to **Edit | Project Settings | Player Settings** and selecting the mobile device. If you are building for both iOS and Android, you will not have to set these properties for both. As you can see from the following screenshot, the asterisk next to the property of **Default Orientation** states that the settings are shared between multiple platforms:

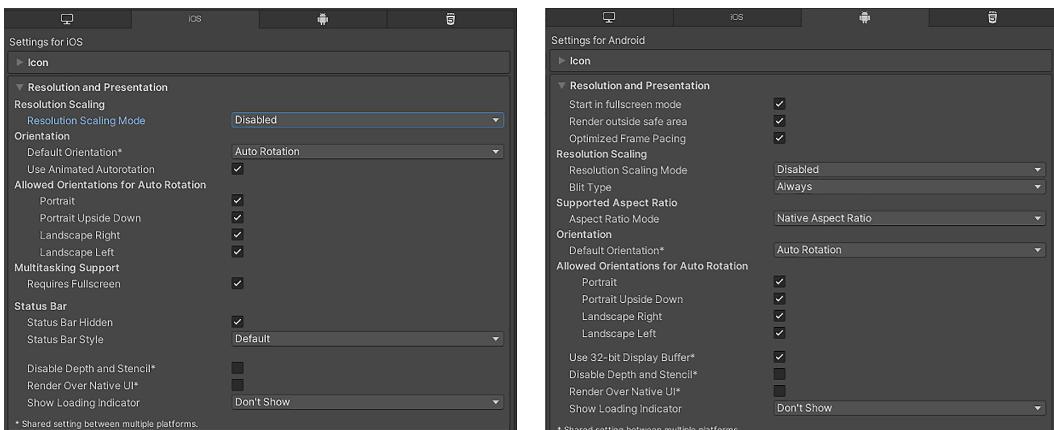


Figure 2.9: Resolution and Orientation Settings for mobile

You can set the **Default Orientation** to either **Auto Rotation** or one of the other rotations, as shown:

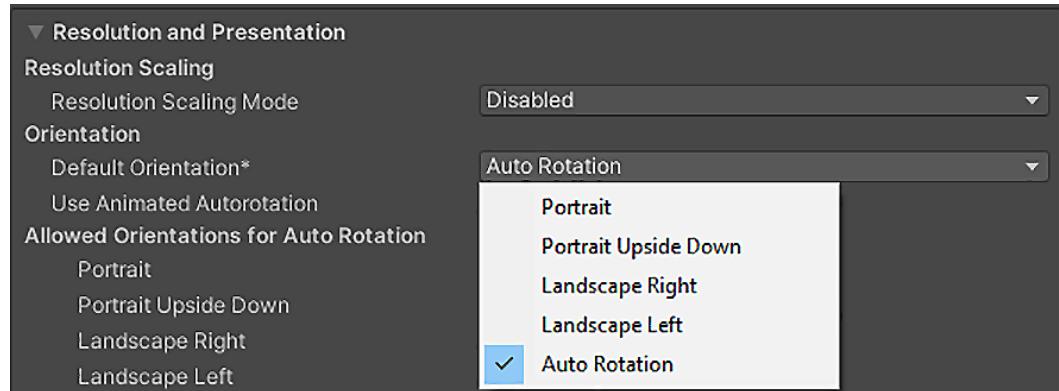


Figure 2.10: Orientation Options for mobile

Unity defines the following orientations as the following rotations:

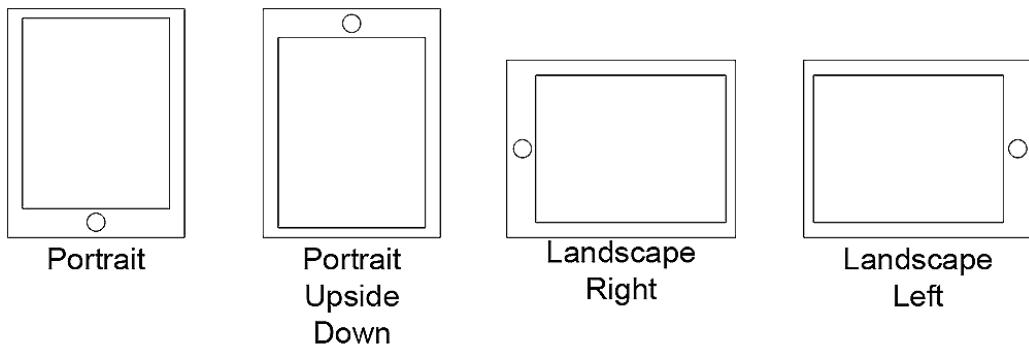


Figure 2.11: Mobile orientation rotations

When you select a rotation other than **Auto Rotation** as the **Default Orientation**, the game will only play at that orientation on the device. If you select **Auto Rotation**, you will have the option to select between multiple orientations:

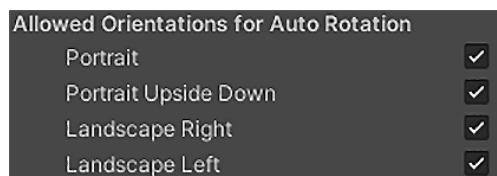


Figure 2.12: Auto Rotation options

In most cases, it is best to choose only the **Landscape** orientations or only the **Portrait** orientations, but not all four. Generally, allowing all four orientations will cause issues with the game's UI's ability to scale appropriately.

Players tend to prefer to be able to rotate their games (especially if they're like me and like to play games in bed while their phone is charging, thus being forced to face a specific direction), so unless you have a good reason to stop rotation, it's a good idea to enable both **Portrait** and **Portrait Upside Down** or **Landscape Right** and **Landscape Left**.

Recommended button sizes

When creating a mobile game, pretty much all of your interactions are controlled by button and screen taps. Buttons that are a reasonable size on a PC or console game may be too small for a mobile game. Therefore, you'll want to make sure to design your buttons so that they are still visible on the smaller screen and large enough to be touched by a finger.

Apple, Google, and Microsoft all have specific recommendations for the size of a button's *touchable area* when designing for their devices. Apple recommends that buttons be 44 points x 44 points. Google recommends 48 dp x 48 dp with 8 dp spacing between two buttons. Microsoft recommends 9 mm x 9 mm with 2 mm padding between two buttons.

You can find information about designing touch hit areas for each mobile platform at the following locations:

- Apple: <https://developer.apple.com/design/human-interface-guidelines/layout>
- Google: <https://material.io/design/usability/accessibility.html#layout-and-typography>
- Microsoft: <https://docs.microsoft.com/en-us/windows/uwp/design/input/guidelines-for-targeting>

Annoyingly, all of these recommendations are in different units of measurement. So, what do these numbers even mean in terms of design? How do you make sure your buttons are 9 mm x 9 mm or 44 points x 44 points? And why are they talking about these measurements in different units? It's almost like they are all competitors and don't want to work nicely together! To answer these questions, let's first look at what the various units of measurement represent:

- A **point (pt)** is used to measure what represents a physical measurement on a screen. 1 point is 1/72 of an international inch or 0.3528 mm. It is primarily used in typography and print media. When working with a program such as **Illustrator**, creating an object in points and then exporting your image at 72 ppi makes pixels and points the same size. Points and pixels are not the same, except when exporting at 72 ppi.

- **Density-independent pixels (dp)**, pronounced “dips”, is a unit of measurement that was created to maintain consistently sized items on screens with different **dpi (dots per inch)** values. A density-independent pixel measures the size of 1 pixel on a 160 dpi screen. Using this conversion is like saying it would appear at this size on a 160 dpi screen, and it should appear as the exact same physical size on any other screen. You can read more about density-independent pixels at <https://developer.android.com/training/multiscreen/screendensities>.
- When **millimeters (mm)** are used to describe a button size, the size is a physical representation of the button on the screen. So, if you were to take a ruler and hold it up to the screen, it would be consistent with this unit of measurement.

OK, so they all represent some physical unit of measurement on the screen. That makes things a little easier. Let's convert all of these values to millimeters so we can compare them in a unit of measurement that is a bit easier to conceptualize. I'm also going to convert them to points, since you can use points in a program, such as Illustrator, to create your button art.

If you need to convert any of these units of measurement and you're not too keen on the idea of doing math, googling *convert points to mm* will bring up a nice conversion calculator for you.

You can also use the following converter tool – it is really handy for bouncing between all of the different units of measurement: <http://angrytools.com/android/pixelcalc/>

In the following chart, I rounded to the nearest integer the measurement for points, and to the nearest tenth for millimeters, to make things easier. We can use this image as a way to compare the different sizes (the image has been scaled and the sizes may not translate to their real-world measurements):

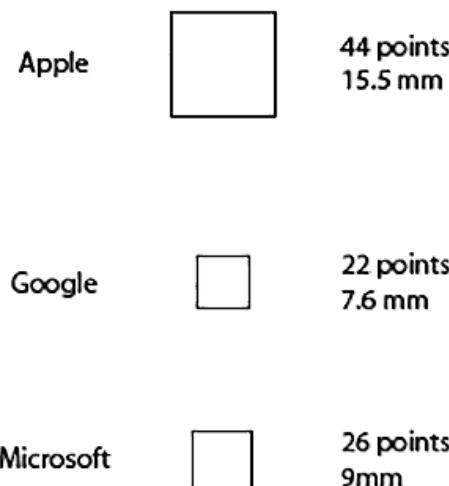


Figure 2.13: Recommended minimum tappable button sizes

So, which size should you use? It's up to you. You don't have to use their recommendations, but I personally go with the Apple recommendation since it is the largest and therefore meets the recommendations of the other two. Additionally, the larger the button is, the more people will be able to easily touch it. In *Chapter 4*, we will discuss designing our UI so that the maximum amount of people will be able to interact with it.

Another consideration is whether your game will be played with thumbs or with fingers. If the game will be played with thumbs, you'll want bigger buttons because thumbs are bigger! The numbers described previously are minimum recommendations, so they would be used for a finger tap, not a thumb tap.

So, how do you ensure that your buttons are always the size you want in your game? The **Canvas Scaler** component! In *Chapter 6*, we will discuss how ensuring a button of a specified size, regardless of resolution, can be achieved by setting the **Canvas Scaler** component's **UI Scale Mode** to **Constant Physical Size**. You have the option to have your Canvas's measurement units be in millimeters or points (as well as a few other units).

My recommendation when designing for mobile devices is to have multiple devices on hand to test at various resolutions. Play the game and see how it feels to you. Ask people with smaller and larger hands than yours to play. Even after following the minimum guidelines specified by the various mobile platforms, you may still find your buttons are too small for what you need.

Google and Microsoft also specify visible sizes that they recommend, so you can have a smaller button image as long as the button's hit area is the recommended size. If you want a button that is smaller visually but has a larger hit area, instead of attaching the button component to the tiny piece of art, attach it to a larger parent hit area and change the target image of the button to the tiny art.

Full screen/screen portion taps

Many mobile games have a single input whereby you can tap anywhere on the screen to make an action happen. For example, endless runners tend to allow the player to tap or press and hold anywhere on the screen to jump. To achieve this, you only have to add an invisible button that covers the whole screen. If you have another UI that receives inputs, it needs to be in front of the full-screen button so that the button does not block the inputs to the other UI items.

Some games require that you tap on specific regions of the screen to perform specific actions. For example, I created a game called Sequence Seekers for my doctoral dissertation. This game included a down-the-mountain mode in which the player had to tap the left or right-hand side of the screen to move left or right in the game. I achieved this by adding invisible buttons that covered the two halves of the screen, as shown here:

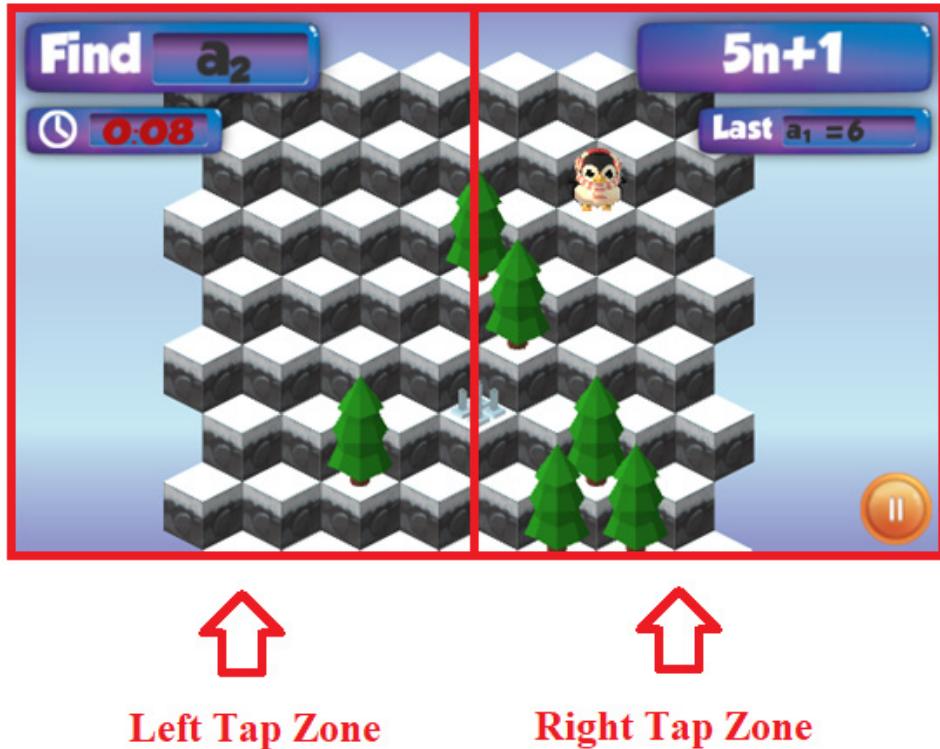


Figure 2.14: Using invisible buttons to create tab zones

In *Chapter 9* and *Chapter 11*, we'll discuss how to implement such buttons as well as how to implement floating D-Pads and joysticks.

The thumb zone

When designing a mobile game, it's important to consider how the player will hold the device. You don't want to put your UI in areas that will be difficult for the player to reach. Players tend to prefer to hold and play with one hand. Not all games allow for this, but if possible, you want to allow your players to do so. How do you know whether your UI is in an area that's reachable by the thumb? Put the UI in the thumb zone! Essentially, the thumb zone is the area of the phone that is comfortable for the player to reach when holding the phone with one hand. You can find the thumb zone on your particular phone by holding the phone and easily moving your thumb around without having to move your hand.

The following blog post offers a really great explanation of the thumb zone, along with a handy (no pun intended) template for finding the thumb zone on various devices: <https://www.scotthurff.com/posts/how-to-design-for-thumbs-in-the-era-of-huge-screens/>

The phones referenced in the link are a bit on the old side, but it is still one of the best resources related to the thumb zone available on the internet.

As a lefty, I implore you to consider making the game as easy to play with the left hand as it is for the right when designing with the thumb zone in mind.

Other mobile inputs

When designing for mobile, it's important to remember that the input works a little differently than it does with a computer or console game. Most of the input on a mobile is controlled by the touchscreen, accelerometer, or gyroscope. This opens up a different set of design choices for you when creating mobile games.

Touchscreen devices can generally access multiple touches. You can use multi-touch for different types of interactions, but the most common usage of multi-touch allows the player to pinch-to-zoom. In *Chapter 8*, we will discuss how to use multi-touch input to create pan and pinch-to-zoom functionality in a game.

Most mobile devices have a built-in accelerometer and many also have a gyroscope. Without getting too technical in describing how they actually work, the difference between the accelerometer and the gyroscope is what they measure. The accelerometer measures acceleration within the 3D coordinate system and the gyroscope measures rotation. We will review examples of how to use the accelerometer and gyroscope in *Chapter 8*.

Device-specific resources

If you are making a UI for a mobile device, you may want to use the device-specific UI elements to maintain a consistent style. You can find various art assets and templates for designing UI for each mobile platform at the following locations:

- Apple: <https://developer.apple.com/design/resources/>
- Android: <https://developer.android.com/design/index.html>
- Windows: <https://developer.microsoft.com/en-us/windows/apps/design>

Summary

Creating a UI for mobile devices isn't too different from creating a UI for a console or computer, but it is different in that you can accept more than one screen input and can also access information about the device's accelerometer and gyroscope. Additionally, resolution plays an important part in your game's development, since mobile devices have a huge range of resolutions and aspect ratios.

In the next chapter, we'll discuss the design considerations for developing UI for XR applications, including VR, MR, and AR.

3

Designing VR, MR, and AR UI

The study of **user interface (UI)** design for **virtual reality (VR)**, **mixed reality (MR)**, and **augmented reality (AR)** is expansive and ever-growing. It is an emergent technology, and best practices are not yet fully defined; however, there are a lot of researchers and developers working diligently to determine what those best practices are. In this chapter, I'll summarize some of the generally agreed-upon best practices for designing a UI for **extended reality (XR)** experiences.

In this chapter, I will discuss the following:

- Distinguishing between XR, VR, MR, and AR applications
- Design considerations and best practices for developing UI for VR
- Design considerations and best practices for developing UI for MR
- Design considerations and best practices for developing UI for AR

What are XR, VR, MR, and AR?

Before we can begin discussing best practices for VR, MR, and AR, we should clarify the definition of XR, VR, MR, and AR.

You may notice, if you look ahead on this chapter, that there are no sections titled *Designing UI for XR*. That is because XR encompasses VR, MR, and AR! It is an umbrella term. In fact, a more concise title for this chapter could have been *Designing XR UI!* XR includes VR, MR, and AR technologies:

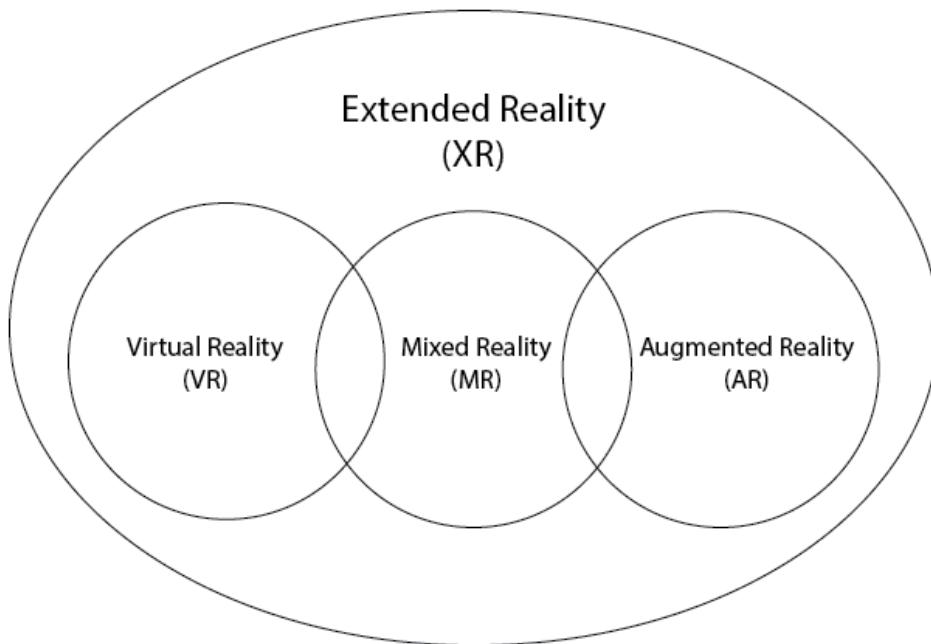


Figure 3.1: Representation of XR technologies

VR experiences are perhaps the easiest to describe. They exist fully in the virtual world. The entire physical world is blocked from view, and the only thing you can see is the virtual space. This type of space can often be so immersive that the brain struggles to distinguish it from reality. Devices that facilitate VR fit over the user's eyes to completely block out visuals in the real world. Popular devices today that support VR are the Meta (formally Oculus) Quest series, the Sony PlayStation VR series, and the HTC Vive series. Some popular examples of VR games include *Half-Life: Alyx* and *Beat Saber*.

MR overlays virtual items in the real world and allows you to interact with them. The user can see the real world and interact with both real-world items and virtual items. MR devices are also worn on the user's face in the form of glasses and headsets. They either do not fully block out the real world from the user's view or they allow video passthrough to allow the user to see the real world through lenses. Popular devices today that facilitate MR exercises are the Meta Quest 2 (and above), the Microsoft HoloLens series, and Magic Leap. A popular example of an MR experience is *I Expect You To Die: Home Sweet Home*.

AR is similar to MR in that it combines real and virtual worlds; however, it differs in that the virtual items do not interact with the real world. In AR, virtual items are simply overlaid on the real world and don't appear within the same space as the real world. Also, it does not enable the user to interact with virtual items in a way that feels like it is happening in their world. AR interactions usually happen on a screen, while MR interactions happen within a physical space. The distinction is subtle, but you can consider AR as interacting with virtual items that appear on a screen while MR is interacting with

items that appear as if they are in your space. There are significantly more AR devices available on the market. Smartphones and tablets are the most popular facilitators of AR; however, some smart glasses also do it. Additionally, there are AR screens, kiosks, and installations that can be found all over the world. Popular examples of AR games include *Pokémon GO* and *Ingress*.

Note

There is some nuance to the ideas of XR and their distinctions that I have not fully delved into in this section. If you would like to learn more, I recommend researching the virtuality continuum: <https://www.interaction-design.org/literature/topics/virtuality-continuum>.

Now that we understand the difference between VR, MR, and AR, let's look at some best practices for designing UI for these experiences.

Designing UI for VR

Normally, when you think of UI, you think of a **heads-up display (HUD)**—UI that is on the screen in front of all gameplay. But in VR, there is no screen in the same sense as there is in games played on a console. The player feels as if they are fully immersed in the world, and putting something on the screen they are experiencing a VR game through (the lenses) would result in a fuzzy, unviewable blur since it would be right on top of their eyes. Due to this, VR UI tends to be placed within the world the player will be immersed in.

Note

In *Chapter 16*, we will discuss steps around creating UI with a physical representation within the world.

While there may be some exceptions, there are three locations where a UI in a VR game is usually placed:

- Embedded within the world in a static position. The player must approach it by moving their avatar toward its virtual location.
- Embedded within the world a set distance in front of the player's face. The UI moves with the player and is always in the same relative location regardless of where the player turns their head. Or the player's avatar does not move in space and the UI also does not move.
- Attached to the player's hands—the UI appears as some peripheral device attached to the player's avatar's virtual hands/arms.

Which of the three placements you choose depends on your game's design but can also be affected by if and how you want the player to be able to interact with the UI. In essence, you want to make sure the player can see the UI and interact with it (if necessary).

Let's start our VR UI design exploration with considerations for designing visual UI.

Visual UI placement and considerations

As I've said before, there is not an explicit set of rules for designing VR gameplay experiences as the technology is still considered emergent and researchers and developers are still learning what those best practices are.

When creating a UI that will be a specific distance from the player at all times, it is important to put it in a place that the player can see. Just because a player can see an item doesn't mean it is comfortable to see it. For example, while a player can see items in their peripheral range, this would not be a comfortable place to put the UI. The following diagram shows the general angle for a person's **field of view (FOV)**, where the gray areas are maximum viewing angles:

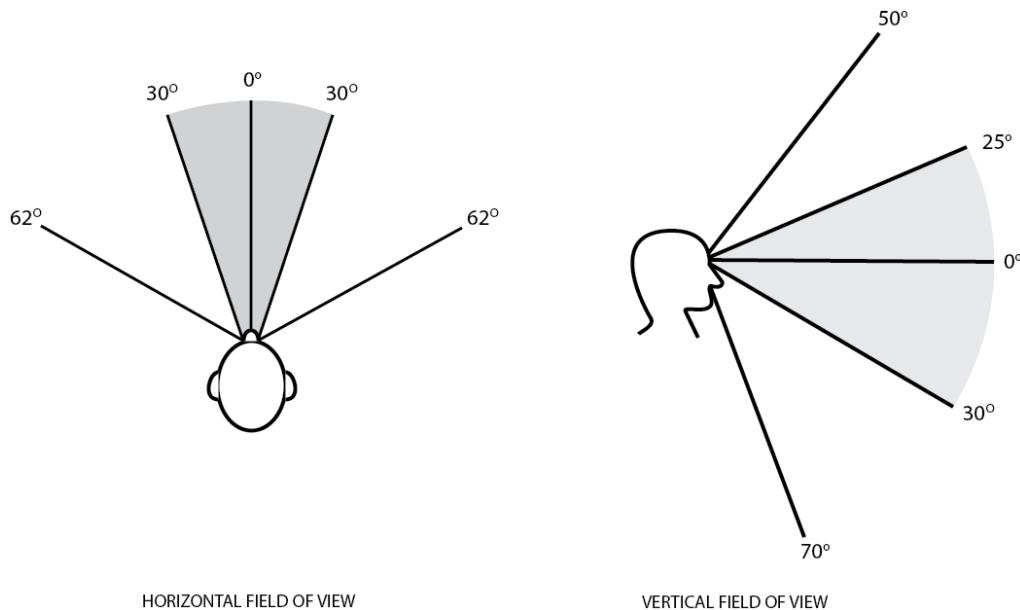


Figure 3.2: Generalization of the human field of view

Note

The angles described in the previous diagram are general guidelines and not set in stone. They will change depending on the user's eyesight, whether or not you want to require them to rotate their eyes (and not their head), and the device you are using.

Keep in mind that when a player moves their head, the camera moves with it. So, if your UI is placed in a position that encourages them to rotate their head and is anchored to their head, the UI will move with their head. So, try to place any anchored UI in the area described in *Figure 3.2*.

On top of the angle relative to the player's eye, you must also consider the distance from the player's eye. If it's too close or too far, it will cause eye strain. Try to keep any visual UI, especially text that must be read, within 1.3 to 3 in-game meters from the player's eyes.

If the player is in a stationary position, creating a static curved UI a set distance away from them always helps the player comfortably move their head around to view the UI.

The last consideration for visual UI I want to discuss is text size. While the recommendation may vary depending on your source, the one I see consistently cited (no pun intended) is that UI should appear a multiple of 2.32 cm tall for every meter away the UI displays. So, if it displays 1 meter away, it should appear 2.32 cm tall. If it displays 2 meters away, it should appear to be 4.64 cm tall. Personally, I have very bad eyesight, and I tend to make my UI text a bit bigger than this so I can easily view it. My recommendation is to have multiple people with varying eye strengths play your game and tell you if the UI is comfortable or not.

Now that we've reviewed considerations for designing visual UI in VR space, let's review some considerations for interactable UI.

Interactable UI placement and considerations

Interactable UI is achieved in a few different ways in VR, and it depends on whether the experience uses a controller or hand tracking.

When using a controller, here are common ways in which UI can be interacted with:

- The player simply presses a button to perform an interaction—for example, pressing a menu button to make a menu button appear and disappear.
- The player selects a UI item with a ray. The player points toward the UI item, a ray appears from their hand, and they then press a button to interact with the item.
- The player's avatar virtually approaches a UI item, and the player interacts with it. This includes pressing buttons in virtual space with a virtual hand or grabbing virtual items.

When using hand tracking, here are common ways in which UI can be interacted with:

- The player performs some gesture, and a UI interaction is completed. For example, making a thumbs-up gesture could cause a menu to appear or disappear.
- The player points with their finger at an item, and a ray appears from their finger. They then complete some other gesture to confirm the selection.
- The player's avatar virtually approaches a UI item, and the player interacts with it. This includes pressing buttons in virtual space with a virtual hand or grabbing virtual items.

When interacting with UI via a ray, it is not necessary for the player to physically be able to reach it. But if you want to create UI that the player's avatar must virtually interact with, you will need to make sure it is within reach of the player both horizontally and vertically. Factors for reachability include the following:

- Whether the game is played in a stationary mode or if the player is expected to move around
- Whether the player's avatar can move around or not
- The player's height
- If players are sitting or standing
- The player's general mobility and disabilities

A good rule of thumb is to try to make sure players can reach your items by placing them between 0.5 and 0.75 meters from their stationary position. Just as you want to test your visual UI with people with various levels of sight, you will want to test your UI with people of various heights and abilities.

With both controllers and hand tracking, you want to make sure that whatever interaction you choose is not tiring for the player. Requiring them to perform multiple gestures or hold their hands up for a significant amount of time can cause fatigue and strain that you don't want to inflict on your player.

That concludes the major considerations for designing UI for VR that I wish to discuss. However, there is a lot more research and information out there for you to explore! For more information on designing UI for VR, I recommend checking out the resources provided at the end of the chapter.

Now that we've looked at designing UI for VR, let's look at designing UI for MR.

Designing UI for MR

Designing UI for MR is extremely similar to designing UI for VR, so most of what I discussed in the previous section translates. However, designing UI for MR does have a few extra caveats.

The main distinction between designing UI for VR and MR is MR incorporates the player's physical space and items around them. When playing VR, people generally are assumed to have a wide-open space around them—so that when they flail about recklessly, they don't hit anything and hurt themselves.

The opposite tends to be true with MR experiences—players are generally encouraged to be around furniture, walls, and other items in their real world.

The UI being placed virtually not only has to be accessible via the player, but it also cannot intersect with things that exist within their world. You can't have the player pressing buttons that are inside their desk or viewing screens that are behind their walls. So, when designing UI for an MR experience, you must consider the player's space and not just the player's body when placing your UI.

Ways around this include allowing the player to pick where their UI displays, displaying it on top of surfaces that the technology detects (how it is detected is highly dependent on the device the player is using), or attaching it to the player.

MR is an even more emergent space than VR, especially in the video game field. MR has mostly been used for industrial and medical purposes up to this point and is usually performed in a highly controlled space. MR for a gamer in their home with infinite possibilities of furniture and object placement is still in its infancy and until recently was only achievable in Unity via the Microsoft HoloLens. However, the Meta Quest 2 has added the ability to create MR games in Unity, and Meta Quest 3 plans to have more curated and polished MR support. Additionally, Magic Leap has partnered with Unity to allow for MR development. So, you will likely see more best practices and technology around MR UI in the coming years.

Note

For information about developing with Unity and the HoloLens, see <https://learn.microsoft.com/en-us/windows/mixed-reality/develop/unity/unity-development-overview?tabs=arr%2CD365%2Ch12>.

For information about MR capabilities planned for Quest 3, see <https://developer.oculus.com/blog/build-the-next-generation-of-vr-mr-with-meta-quest-3/>.

For information about developing with MagicLeap, see <https://m11-developer.magicleap.com/en-us/learn/guides/unity-overview>.

Sadly, since MR is such a new and emergent space within the Unity engine, I cannot say much more about designing UI for it. However, AR has been around for quite some time, so let's explore it now!

Designing UI for AR

Remember, that what distinguishes AR from MR is AR tends to be on a screen that overlays the real world, while MR appears more immersed and inside the world. So, interactive items for AR experiences will be displayed on the screen, not within the world.

Designing UI for AR is highly dependent on the device that is being used to augment reality. For the sake of clarity, I am going to focus my discussion on AR games developed for cell phones rather than trying to speak universally and include things such as installments and kiosks.

When designing UI for AR app on a mobile or touchscreen, you will want to follow the rules outlined in *Chapter 2*. Best practice is to place most non-augmented digital items (that is, the HUD UI) on the outer edges of the screen. Then, augmented digital items will appear in the middle of the screen. This will cause items that are augmenting the world to be the focus.

Summary

In this chapter, we discussed some basic design considerations for developing XR experiences. Since the ways in which we experience VR, MR, and AR are all different, we reviewed considerations you should keep in mind when designing for each of these types of experiences.

XR is still an emergent space, so there are not yet many widely accepted standards. This is particularly true with MR, which just recently began having consumer-available MR devices. However, the information provided in this chapter should help you get started with XR UI design and get you started thinking about how the XR space differs from screen space.

In the next chapter, we will discuss the concepts of universal design and accessibility and how you can make your user interfaces more inclusive and user-friendly.

Further reading

Further resources for learning about designing UI for XR can be found here:

- <https://medium.com/@oneStaci/https-medium-com-ux-vr-18-guidelines-51ef667c2c49>
- <https://www.youtube.com/watch?v=u6FPoOJ4AuM>
- <https://developer.qualcomm.com/blog/xr-user-interfaces-and-perception-technologies>
- <https://aixr.org/insights/4-challenges-facing-ux-design/>
- <https://uxplanet.org/ux-101-for-virtual-and-mixed-reality-part-1-physicality-3fed072f371>

4

Universal Design and Accessibility for UI

When designing your interface, you want to consider all the different types of players who may interact with your game. You want to consider their size, their age, their current situation, their ambient location, their output device, their input device, their cognition, their preferences, their mobility levels, and more. You want your interface to be usable by as many types of people as possible.

It's important to consider these factors at the beginning of the design process as designing your UI initially with these things in mind is significantly easier than incorporating them later when your UI is already designed, built, and implemented.

In this chapter, we will discuss the following topics:

- What are universal design and accessible design?
- Universal design principles
- Accessibility design

What are universal design and accessible design?

Ronald Mace coined the term **universal design** and defined it as follows:

“The design of products and environments to be usable by all people, to the greatest extent possible, without the need for adaptation or specialized design.”

Universal design is the concept of designing products that are usable by all people regardless of age, size, preference, ability, disability, condition, or situation. It does not focus on designing for a specific group of people but instead focuses on designing products that are useable by the widest range of people. While initially focused on architecture, the concepts of universal design have been expanded to include the design for all types of products, including video games. **Accessible design** is a design

that specifically focuses on usability by those with impairments and disabilities and is a subset of universal design.

For a design to be effective, purposeful decisions must be made to ensure that the design is both functional and useful to all types of people. When designing a user interface, you should consider how to make it usable and accessible to all people from the beginning of the design process. It is much easier to design a universally usable and accessible UI if you start with these considerations in mind than it is to retrofit features on top of pre-existing interfaces.

In the following sections, I will discuss considerations for designing user interfaces using the principles of universal design as well as the special considerations you should take to make your interface accessible.

Since this book is on the development of UI within Unity, which is primarily a video game engine, the majority of the examples and use cases I will describe will be related to video games.

Universal design principles

In 1997, a working group at North Carolina State University developed the 7 Principles of Universal Design. These principles are meant to help guide the design of products so that they can be universally usable. These principles are formally defined in terms of different types of designs with an emphasis on architecture. I will paraphrase each principle in terms of its application to digital user interfaces and then provide examples of how each principle can be used in video game interface design. If you'd like to view the list of principles and their specific guidelines, you can visit <https://universaldesign.ie/what-is-universal-design/the-7-principles/>.

Many of these principles overlap and some of the examples I will provide could apply to multiple principles.

Equitable use

The equitable use principle states that a design should be equally usable and appealing to all. If an identical experience is not possible for all, they should have an equivalent experience. An interface should be designed to appeal to all users and should not stigmatize or segregate users. This is the first principle because it ultimately drives all the other principles.

For example, high-contrasting colors should be used in interfaces. High-contrasting interfaces are not only appealing to all users but they help mobile users sitting in direct sunlight see your interface and they avoid stigmatizing users with visual impairments such as low vision and color blindness.

Remember that the goal of universal design is not explicitly about designing for accessibility (that is, designing for groups with disabilities). Utilizing something like a high-contrasting color scheme is beneficial to all, not just those with visual impairments. It is also not a setting that explicitly needs to be turned on, such as turning on a color blind mode, which would be designing for accessibility. I'll discuss designing for accessibility later in this chapter.

If you're making a PC game, you don't want to hide information so that it is only accessible by a mouse. Allowing information to be navigated to via the d-pad on a controller or tabbed to by a keyboard will allow users who are using different input devices (due to personal preference or vision impairments) to access the same information as a mouse user.

Flexibility in use

The flexibility in use principle states that people with a wide range of preferences and abilities should be accommodated by the design and people should be provided choice in how they use the design. The key to this principle is providing users with choices when it comes to interacting with your game.

If you're making a mobile game, you can provide left and right-handed modes that swap which side of the screen the buttons are on, as I did in my game Barkeology, which is shown in the following figure. This allows players to easily interact with the interface without blocking the gameplay with their dominant hand.

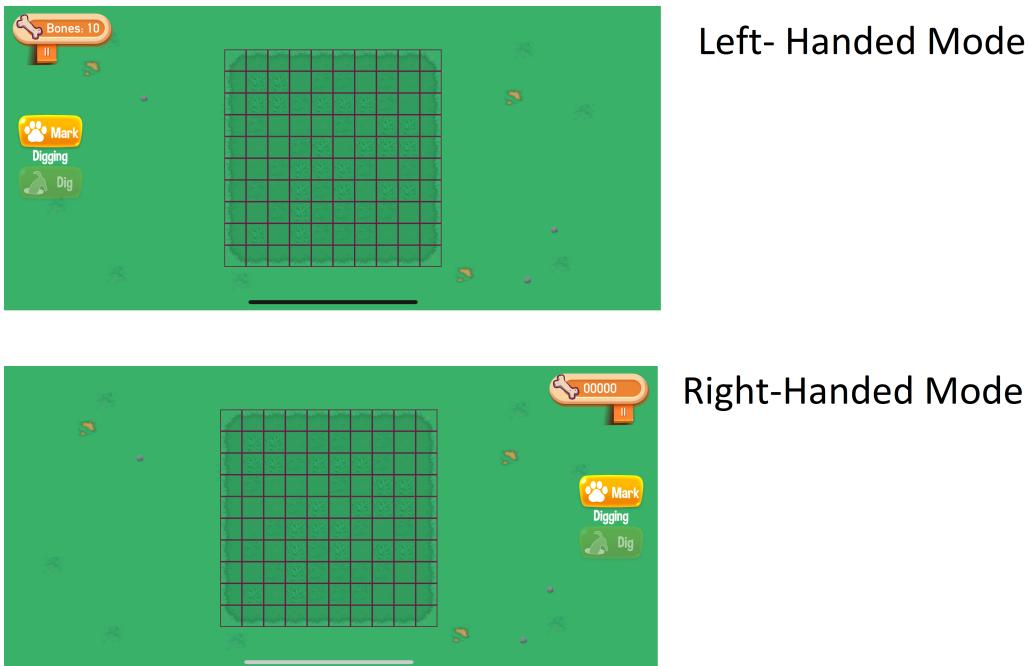


Figure 4.1: Left-handed mode versus right-handed mode in Barkeology for iOS

Fantasy Life Online lets you choose between three locations for the placement of the “fury button” – a button that appears whenever a special attack is available to the player. This allows the player to pick a location that is most comfortable to them based on their playstyle and the way they hold their phone.

For PC games, you can allow players to choose different input devices. For example, you can provide the option to play with a keyboard and mouse or a controller. You can allow players to map the keyboard or controller however they wish or give them predefined schemes to choose from. Allowing players to choose controller button schemes is also applicable to consoles.

If your game has text, allow users to change the size or the speed at which it is presented to them.

There are many more examples of ways in which you can allow flexibility in your interfaces. When designing your interfaces, just be sure to consider the different preferences your players may have so that you can lay out your interface and map your inputs accordingly.

Simple and intuitive use

The simple and intuitive use principle states that a design should be easy to understand for people, regardless of their past knowledge, experience, skill, or language level.

Don't make your interfaces overly complicated. If you have to explain it, it probably needs to be redesigned. Players who have never played a video game before should be able to understand your interface just as easily as a veteran.

Place the most important information in the most visible locations and make them the easiest to find. Don't hide common functionality behind multiple button clicks and menus. If your menu system is nested and particular menus are accessed more frequently than others, consider mapping them to hotkeys or buttons.

Provide feedback to the user to let them know when they are interacting with your interface. If you have on-screen buttons meant to be interacted with by a tap or mouse click, have them provide feedback to the user to let them know when they have highlighted them or clicked on them.

Provide prompts to the user that signal to the player which parts of your interface are interactable. You can accomplish this by designing buttons that look physically pressable or by drawing attention to items with animations or color schemes.

It's important to remember that your users will have various reading levels and may speak different languages. Reduce your dependency on text by using more **icon metaphors** when possible. Metaphors are symbols whose meanings are widely recognized. For example, most people will recognize the meaning of the buttons shown in the following figure as play, pause, menu, settings, close/cancel, confirm, mute, and save:



Figure 4.2: Examples of interface metaphors

Accompany the icon metaphors with text so that users have multiple ways of perceiving what specific icons mean.

It's not always possible to completely remove all text from your game's UI and replace it with icons and imagery. So, translating your game into multiple languages could increase your UI's universal perceptibility. In *Chapter 11*, I will discuss things you can do while building your UI to make the translation process go more smoothly, as well as cover an example of creating a UI translation system.

Perceptible information

The perceptible information principle states that information should be conveyed to users in a perceptible way regardless of the environment they are in or their sensory abilities. When considering this principle, you want to think of alternate ways in which information can be conveyed and how you can clearly convey it in multiple scenarios. Remember that the interface is the lens through which the player perceives and interacts with your game. So, they must be able to understand it.

The colors of the UI should stand out compared to the background, but it should also not stand out so much that it causes eye strain. There is no specific color scheme you have to use for your game, but as a general rule, split complementary color schemes are the best for reducing eye strain while also producing enough contrast to make items distinguishable.

High-contrast text will make it easier to see in different lighting. Make sure the text is appropriately sized so that it's visible and allow the text to be resized to the user's preference. Also, choose fonts that are clear and easy to read.

Voiceover isn't just for dialogue! You can provide voiceover for your menus as well. Far Cry 6, for example, has an automated voice read out the various menu options on the start screen to allow those with visual impairments to still perceive the items on the screen. This option is turned on by default and does not require users to interact with a menu to access it.

If you're making a console game, consider the fact that your players will have different types of TVs with different resolutions and brightness settings. Upon starting a PlayStation or Xbox game, you may have seen a prompt to adjust the brightness until an image was visible or maybe saw a prompt to move the corners of a box until they reached the edge of your screen. These prompts are safeguards to ensure that the game appears correctly to players regardless of their TV.

Tolerance of error

The tolerance of error principle states that adverse consequences of interacting with the design should be minimized. We've all accidentally saved over or deleted a save file at one point in our life without meaning to because we hit the wrong button or misread the directions. The tolerance of error principle is meant to minimize these types of occurrences.

A well-designed interface doesn't necessarily have to be convenient or easy to use. People love to click or tap quickly through things. Sometimes, we want to make that more difficult for them. The concept of **designing for inconvenience** involves making interfaces less convenient or more difficult to interact with. This may seem like a counterintuitive design, but if the interface that lets you delete that last save file you accidentally deleted were less convenient, perhaps your save file would still be with us.

Warning popups that ask users to double confirm a deletion, switching the location of affirmation buttons so that users don't click quickly through things, adding timers between clicks, and requiring press and hold are all ways that we can make our interfaces slightly less convenient for the user to interact with when the consequences of interacting with it may be detrimental.

Low physical effort

The low physical effort principle states that the design should be comfortable to use and fatigue and discomfort should be minimized. One of the guidelines for this principle states to minimize repetitive actions. This may seem impossible to do in video games, which, let's face it, tend to require a lot of repetitive actions, but organizing your interfaces in ways that reduce clicks and mouse drags can improve the quality of your UI.

One way to reduce physical effort with your interface is to group similar actions on the screen. Don't make users bounce back and forth on the screen. Don't require users to jump to multiple menus when it would be easier for them to do something all in a single view menu. You can reduce mouse usage/drag by assigning actions to hotkeys or creating shortcuts. Also, allowing users to navigate through menus with arrow keys or controller buttons rather than just the mouse may be more comfortable for some users.

When designing UI in VR, the amount of physical effort needed to interact with your interface jumps exponentially compared to a game on a traditional 2D screen. Grouping similar items and allowing users to navigate menus with the controller rather than pointing will make interacting with your interface much more comfortable.

The rumble feature can be a great way to provide feedback for your interface and gameplay, but it can also be considered uncomfortable to some players and has been purportedly linked to hand-arm vibration syndrome. If you include it, be sure to allow users to turn it off.

One frustrating interaction on a console game is entering information into a form. If you have players enter long text strings with an on-screen keyboard, having to navigate to each letter gets very tedious, very fast. Consider the possibility of allowing players to enter these long strings on their phone or computer and send the data to the game, instead of on the console.

Size and space for approach and use

The size and space for approach and use principle states that interfaces should allow users to interact with them regardless of their mobility, size, posture, or position. This principle ties closely to the previous principle of low physical effort in that you want them to not only be able to physically interact with your interface but also do so comfortably.

When determining the keyboard layout for your game, make sure that the player does not have to stretch their hands in ways that are either impossible or uncomfortable for those with smaller hands. If you have two buttons on a controller that are frequently used together, you want to make sure that the button combination is doable. For example, you do not want to ask them to hold down both the X button and the B button on the Xbox controller at the same time.

Allowing for multiple types of input devices is key to this principle as it will allow people to use input devices designed for their specific size and mobility. For example, with a PC game, some people may find interacting with a controller easier than a keyboard.

Additionally, allowing for the sensitivity of your inputs to be adjusted can help those with different mobility issues. For example, I have problems with hand pain and tremors. When I can adjust the sensitivity on controllers and mice, this makes it easier for me to interact with the game in a way that doesn't affect gameplay and can reduce the pain I experience doing so.

When designing a VR interface, don't put UI items too high or too far away from the player's reach. This could make it impossible for smaller players, players sitting down, or players with mobility issues to interact with your UI.

In *Chapter 2*, we discussed the thumb zone. This is a location of the screen that can be easily reached by the player's thumb while they're holding their phone and where most of your interactable UI should be placed. We also want to make sure that buttons are as big as possible on mobile screens. You don't want the buttons to be so small and so close together that those with larger hands would not be able to interact with them.

Now that we've reviewed the various principles of universal design, let's look at how we could design specifically for those with impairments and disabilities.

Accessibility design

Recall that universal design involves designing UI that is universally usable for all people. Accessibility design, on the other hand, involves designing with specific impairments and disabilities in mind. In this section, I will discuss a few very specific types of impairments and disabilities and how you can design your UI in such a way that it is accessible to individuals with these impairments and disabilities. A few of these examples will overlap with the ones that I discussed in the universal design section.

Vision

When designing your interfaces, you should consider different types of visual impairments and disabilities, including (but not limited to) color blindness, low vision, and blindness.

Essential information should never be conveyed by color alone and you should always include another way of conveying that information. For example, let's say a game has a number that is red when negative and green when positive. That number could also have negative and positive symbols to indicate the sign. If the number represents some change, maybe it can fly up when positive and fly down with negative. This will make sure those who have color blindness will still be able to see these important pieces of information.

Whenever possible, you should avoid color combinations that will be indistinguishable for those who are color blind; if it's not possible, use other indicators to make items distinct. For example, if you have a match three game that uses colors to indicate matchable pieces, also include symbols on the pieces to make them easier to distinguish.

The following websites offer very good information on designing accessible UI for color blindness:

- <https://medium.com/theuxblog/how-to-design-for-color-blindness-a6f083b08e12>
- <https://www.smashingmagazine.com/2016/06/improving-color-accessibility-for-color-blind-users/>

As stated earlier in this chapter, you should make sure that your text and other UI elements have very high contrast. This makes it easier for those with color blindness and those with low vision to be able to clearly perceive your UI. The following website is an excellent resource for being able to check the contrast between two colors: <https://webaim.org/resources/contrastchecker/>.

While the preceding website shows if colors meet a contrast ratio required by web regulations, the information translates well to video games.

If you have important information that only appears temporarily on the screen in the form of a popup, make sure it appears in the player's line of sight. Placing it outside of the line of sight could be difficult for those with low peripheral vision to see. Better yet, allow the player to choose where on the screen the important information will pop up.

Remember, according to the principles of universal design, choice is very important. Allow players to change settings for as many of the visual elements as possible. If your game uses a crosshair or cursor, allow the player to change the appearance of the crosshair or cursor. Allow your players to change the size of the interface or text. Allow players to change the font of the interface to one that has higher kerning or spacing or is less frilly.

It's also important to indicate information in ways that are not purely visible. For example, suppose your UI turns red to indicate your player has low health and you have a red health meter on the screen. You can also include a beeping sound and a vibration of the controller to indicate low health. This will help those with both low vision and color blindness be able to perceive the low health status.

If you have input boxes, you can use speech as a way to input text, rather than just a visual keyboard for input.

Remember, you can also use a description track to describe all of the user interface elements of your game, thus allowing those with low vision and blindness to still be able to perceive what is on the screen.

Hearing and speech

There are multiple ways in which you can make your interface accessible to those with hearing and speech impairments in disabilities. Essentially, you do not want any important information to be conveyed through sound alone and you do not want to require speech for any of your inputs. Additionally, if your UI makes any type of noise, keep overlapping noises to a minimum.

In the low-health example I described previously, there were multiple ways other than just the beeping sound in which the player was made aware of the low health. If your UI includes speech input, make sure that it is not the only form of input and allows users to input information through other means.

While it might not seem like an interface design, including subtitles for all spoken dialogue, would need to be designed as a UI element. Your UI for subtitles will need to be clear and readable. Usually, you should put them in some sort of box so that they easily contrast against the background. You also want to make sure that they sync up with the spoken speech in the game.

Mobility

When designing your interfaces, you want to make sure they are accessible for people with all levels of mobility. The best way to make your game accessible to mobility levels is to include as many options as possible for input devices, input mappings, and configurations. Allowing the player to choose for themselves how they interact with your interface might ensure that it fits their particular mobility needs.

You want your controls to be as simple as possible and require as few buttons as possible. If your controls are particularly complicated, provide alternatives that simplify the control scheme.

Additionally, don't require the player to use an input for a small part of the game that is different than the rest of the game. For example, if most of the game is controlled by a keyboard, don't require the start screen to be interacted with by a mouse. Allow the player to use the same input device for all aspects of the game.

Cognitive and emotional

It's important to consider the cognitive and emotional states of your players as well as their language skills when designing your user interface.

Don't assume that your player can read the language in which you have created your UI or that they can read quickly. If your game contains text, use other contextual elements and images to convey the text's intent. If you have text that displays in your UI and then goes away, allow players the ability to choose when it goes away rather than having it do so at its own speed.

Use your UI to remind players of important gameplay elements and important information about how to interact with your interface. Remind your players of what the controls are by prominently displaying them in your UI. Use your UI to clearly indicate interactive elements and objectives.

Don't place flickering images or repetitive patterns in your UI as this could be triggering to individuals prone to seizures.

While this is probably more related to your gameplay than your user interface, I would like to point out that content warnings in your games with particularly triggering content can be helpful to prepare your user for things that may cause distress. For example, many games now come with a content warning if they have upsetting content (such as Doki Doki Literature Club).

Additional resources

There is so much more I could say about the universal design and accessibility design aspects of user interfaces, but alas, this is not a book about designing UI it is about the development of UI. I hope this chapter at least gave you an insight into how important it is to consider these elements and how important it is to consider them at the beginning of your user interface development.

If you'd like to learn more about accessibility design, I highly recommend that you review the information provided at the following website: <https://gameaccessibilityguidelines.com/full-list/>. It provides an extensive list of best practices related to accessibility for video games. Additionally, it provides multiple examples of best practices that you can use for reference.

You can also visit <https://caniplaythat.com/>, a website that provides information about accessibility in the gaming industry.

Summary

In this chapter, we discussed some key considerations for designing your UI so that it will be as universally perceptible as possible. We discussed the principles of universal design and reviewed some considerations you can implement in your designs to improve their accessibility.

In the next chapter, we will leave the design discussion behind and start looking at how to implement UI. We'll review the different interface systems available in Unity so that you can start putting some of this design knowledge to use!

5

User Interface and Input Systems in Unity

Now that we've discussed various design considerations for developing user interfaces, we can start discussing how to implement them within Unity. Unity provides various systems for creating UI. It has systems in place that allow you to create a UI that will be displayed in your game, or UI that will be displayed only in the Editor. Additionally, it provides multiple systems for receiving input from the player.

At the time of writing, two of these systems are still in active development and do not come packaged in Unity by default. This book will primarily focus on development with the systems that are complete, but since Unity does intend to make these systems standard features at some point, I would be remiss not to discuss them just because they are still in **preview**.

In this chapter, I will discuss the following topics:

- Identifying UI Toolkit, Unity UI, and IMGUI
- Choosing between the three UI systems
- Identifying the Input Manager and the Input System
- Choosing between the Input Manager and the Input System

Let's start by looking at the three UI systems within Unity.

The three UI systems

Unity has three systems that can be used to build UI. Which you choose will depend on where your UI will be displayed, what you are trying to accomplish, whether you're working on a pre-existing project, and how comfortable you are with coding.

The UI you build can either be in-game or in-Editor. In-game UI is the UI that can be accessed by your players. In-Editor UI is UI that displays within the Unity Editor and assists with development.

If you want to build UI for your game or application, you can choose between the Unity UI (uGUI) system or the UI Toolkit. If you want to build UI that appears in your Unity Editor, you can use either the UI Toolkit or IMGUI. The following Venn diagram summarizes the uses of the different UI systems:

Where is the UI?

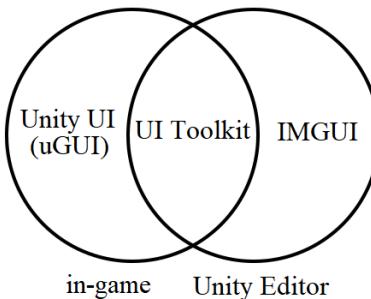


Figure 5.1: A comparison of in-game and in-Editor UI

Looking at *Figure 5.1*, you may be thinking, “Well, UI Toolkit works for everything! I’m just going to learn that and be done with my UI learning journey! That was an easy choice!” Well, unfortunately, it’s not that simple. Let’s look at the different systems a little more in-depth so that you can decide which is right for you.

Unity UI (or uGUI)

The **Unity UI** system, also known as **uGUI**, is the system that is built into Unity *out of the box* and doesn’t require any additional downloads. It is GameObject and Component-based and includes multiple types of UI elements to choose from. When it comes to developing UI for a game or application, this is the most robust and stable option. Since this is the only system for building in-game UI that is not in preview mode and is included within Unity, the majority of this text will focus on how to develop UI using this system.

IMGUI

The **IMGUI**, or **Immediate Mode GUI**, system is a code-based GUI system used to make interfaces within the editor. Its primary function is to assist programmers with development, and it is not recommended for the development of in-game UI due to performance issues. Due to the fact that this system is not intended to be used in-game and this book will primarily concern itself with UI development for games, I won’t spend a significant amount of time covering it, but I will discuss some of its basic functionality and usage in *Chapter 19*.

UI Toolkit

UI Toolkit is a new UI system that is in active development. It is not included within the engine by default and must be downloaded with the Package Manager. Additionally, it is a preview package, which means you have to elect to even see it in the list of available packages provided by Unity. Unity does plan on replacing both uGUI and IMGUI with UI Toolkit eventually. UI Toolkit is being designed with traditional web-development concepts and is structured entirely differently than the GameObject-based uGUI. In *Chapter 18*, I will cover how to download the UI Toolkit package and how to work with it.

Choosing between the UI systems

Which system you choose to use is going to depend on a few things. As discussed earlier, whether you are making UI for the Editor or a game will determine which system you choose. If you're working on UI for a game, you can use Unity UI or UI Toolkit. If you're working on UI for the Editor, you can use IMGUI or UI Toolkit.

UI Toolkit is a new system that is not fully implemented, so if you are working on a pre-existing game with UI already in place, you probably won't be using UI Toolkit. It also may not have all the features you are looking to work with. Because UI Toolkit is in development, it is not guaranteed to be stable, and updating it mid-development may adversely affect your project.

Your comfort with coding could also drive your decision. In general, the coding required to use Unity UI is less intensive than IMGUI and may be more familiar to you than UI Toolkit, since it is GameObject-based. However, if you are familiar with web-based UI creation, UI Toolkit may seem really familiar to you.

If you are considering using the UI Toolkit, I recommend reviewing the examples in this book, as well as the following Unity documentation page, before you make your decision on which system to use: <https://docs.unity3d.com/Manual/UI-system-compare.html>.

Now that we've reviewed the three UI systems provided by Unity, we can review the two input systems.

The two input systems

As defined in *Chapter 1*, **UI** stands for **user interface** and encompasses all mechanisms by which the user and the game convey information to each other. When discussing the three UI systems, we talked about three ways in which the game communicates with the user – specifically through the use of GUI on an output device (i.e., a screen). However, if the user wants to communicate with the game, the user will have to have some means through which they can provide input. The game will then need to process that input.

There are two core ways in which Unity can handle input. The **Input Manager** or the **Input System**. Just as there are varying factors that would determine which UI system you may use, there are also varying factors that can help you determine which input system to use. Both systems allow you to easily process multiple types of input as if they are the same thing. For example, each system will let you process a keyboard space bar and an Xbox controller A button as if they are the same type of input. How they do that will be discussed more thoroughly in later chapters, but for now, let's just look at the general differences between the two.

The Input Manager

The Input Manager is the input system that comes with Unity by default and does not require any additional package downloads. Without having to configure any settings, you can easily accept input from things such as a keyboard, mouse, joystick, or touchscreen. It achieves this by providing pre-defined *input axes* that essentially specify keywords and buttons that bind to them. We will review how this functions more thoroughly in *Chapter 8*.

The new Input System

The Input System (colloquially referred to as *the new Input System*) is a package that is currently in development and is, as Unity states in its documentation, *intended to be more powerful, flexible, and configurable* than the Input Manager: <https://docs.unity3d.com/Packages/com.unity.inputsystem@1.3/manual/index.html>.

If you have been working on a project that is using the Input Manager, it is possible to convert your project to one that uses the new Input System. We will discuss how to implement the Input System in *Chapter 20*.

Choosing between the Input System and the new Input System

Let me start by saying there's not necessarily a right answer on which system you choose. You can theoretically process any type of input you wish with either system. Which you choose will primarily be based on preference, how complicated your project's set of inputs is, and whether you are developing for multiple platforms.

If you are not planning on allowing your player to remap controls (as discussed in *Chapter 4*) or are not planning on cross-platform development, you are probably fine with using the Input Manager and don't need to go through the process of downloading the Input System. However, if you want to have ultra-configurable control schemes, accept inputs from a variety of devices, and accept complicated input actions, you will likely find it easier to write the code that processes these inputs using the Input System than you would using the Input Manager.

Since the Input Manager is used in so many projects that are currently in development, I would be doing you a disservice to completely omit it just because it is not the *new hot*. Additionally, the new Input System is still new and still in active development, so it is subject to drastic changes with each update. However, it does make some things significantly easier to build and is gaining traction in popularity among developers. Therefore, I will not choose just one of these systems to focus on in this book.

Summary

Unity provides multiple ways in which you can display information to your user through the use of three UI systems. Which you choose depends on your needs and whether you are developing UI for a game or the Editor. This book will primarily focus on uGUI, since it is the most stable UI version used for in-game development and is provided within Unity, without additional downloads. However, how you can use IMGUI to develop Editor UI and the UI Toolkit to use both Editor UI and in-game UI will be discussed in the later chapters of this book.

Unity also provides multiple ways in which you can process inputs from a user. While the new Input System is still in development and does not come with Unity by default, I will make sure to give you enough information to use it in your projects.

In the next chapter, we will start developing UI using the uGUI system, by exploring UI Canvases, Panels, and layouts.

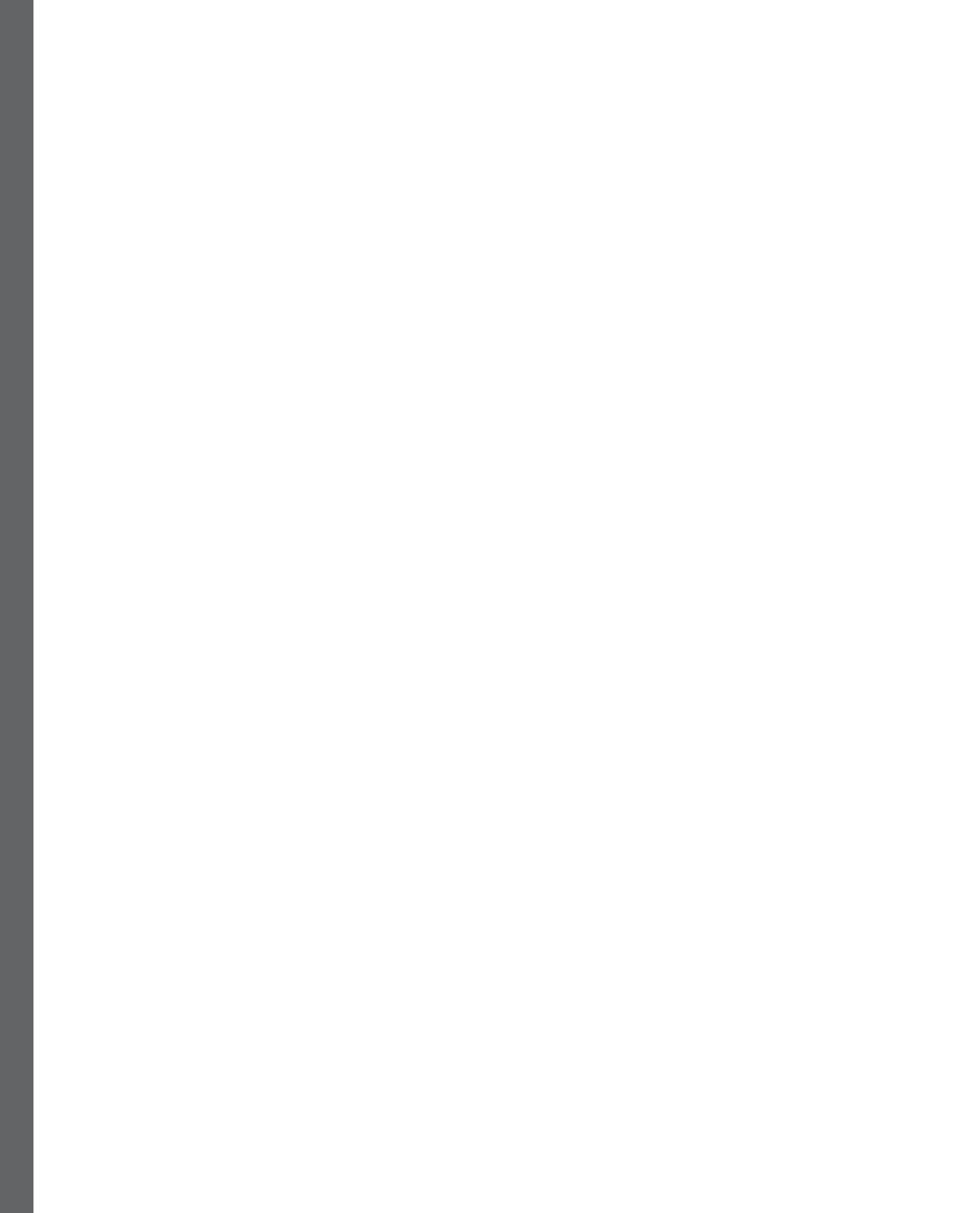
Part 2:

Unity UI Basics

In this part, you will learn the basics of working with the **Unity UI (uGUI)** system. You'll learn how to lay out UIs with the use of Canvases and Panels and their Rect Transform component. You'll also look at how you can use Unity's various Automatic Layout components to help you design UI more easily. Lastly, you'll look at how you can write code for the uGUI System and program interactions for your UI.

This part has the following chapters:

- *Chapter 6, Canvases, Panels, and Basic Layouts*
- *Chapter 7, Exploring Automatic Layouts*
- *Chapter 8, The Event System and Programming for UI*



6

Canvases, Panels, and Basic Layouts

As discussed in the previous chapter, the majority of this text will focus on the Unity UI system, uGUI. Canvases are the core of all UI made with Unity UI. Every single uGUI element must be contained within a Canvas for it to render within a scene. It works similarly to a canvas on which an artist paints, but instead of painting on them, we lay out UI elements on them. So, we'll start our exploration of the various UI elements provided in the uGUI system with Canvases.

Canvases serve the purpose of not only holding all the UI elements within them but also determining how the elements will render and how they will scale. It's important to start focusing on setting up a UI that will scale at multiple resolutions and aspect ratios early on, as trying to do so later will cause a lot of headaches and extra work. Therefore, we will also discuss how to make sure our UI scales appropriately.

In this chapter, we will discuss the following topics:

- Creating UI Canvases and setting their properties
- Creating UI Panels and setting their properties
- Using the Rect Tool and the Rect Transform component
- Properly setting anchor and pivot points
- How to create and lay out a basic HUD
- How to create a background image
- How to set up a basic pop-up menu

Technical requirements

You can find the relevant codes and asset files of this chapter here:

<https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2006>

UI Canvas

Every UI element you create must be a child of a **UI Canvas**. To see a list of all UI elements you can create within Unity, select + | UI from the **Hierarchy** window, as shown in the following screenshot:

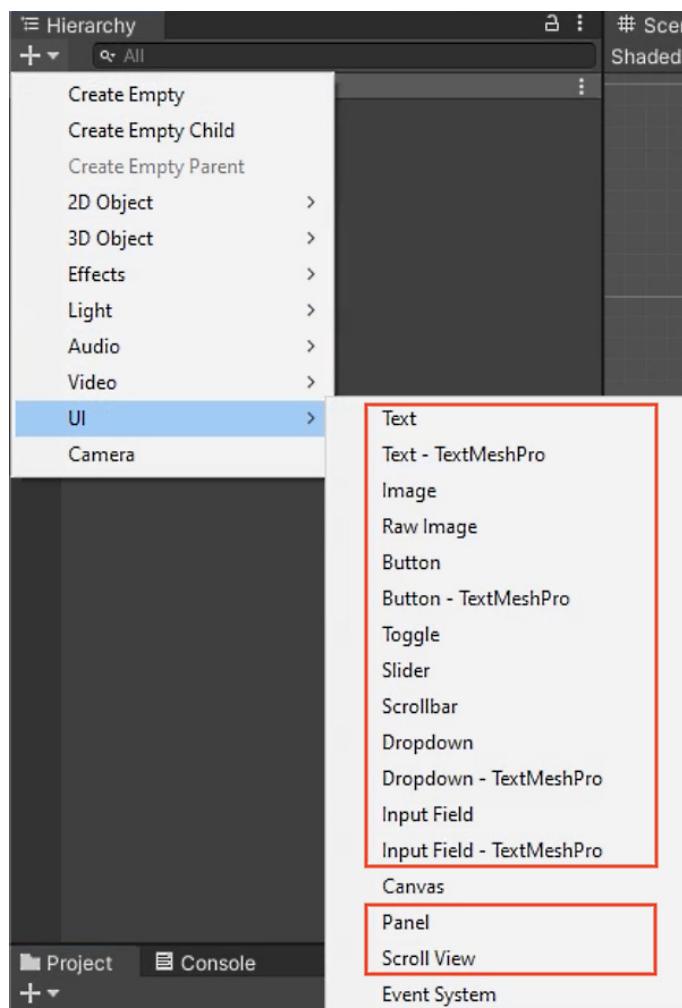


Figure 6.1: The renderable UI elements within the Unity UI (uGUI) system

Every one of the UI items highlighted in the preceding screenshot is a renderable UI item and must be contained within a Canvas to render. If you try to add any of those UI elements to a scene that does not contain a Canvas, a Canvas will automatically be added to the scene, and the item you attempted to create will be made a child of the newly added Canvas. To demonstrate this, try adding a new **UI Text** element to an empty scene. You can do so by selecting **+ | UI | Text**.

This will cause three new items to appear in the Hierarchy list: **Canvas**, **Text**, and **EventSystem**, where the **Text** is a child of the **Canvas**.

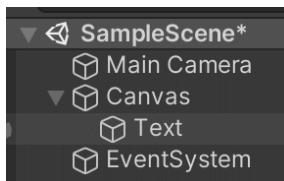


Figure 6.2: The result of adding a UI Text element to the scene

Now that you have a **Canvas** in your scene, any new UI elements you add to the scene will automatically be added to this **Canvas**.

Note

If you try to take a renderable UI element out of a **Canvas**, it will not be drawn to the scene.

You can also create an empty **Canvas** by selecting **+ | UI | Canvas**. When you create a new **Canvas** in a scene, if an **EventSystem** **GameObject** does not already exist within the scene, one will automatically be created for you (as you saw in the preceding screenshot). We'll discuss the **EventSystem** **GameObject** further in *Chapter 8*, but for now, all you really need to know is the **EventSystem** allows you to interact with the UI items.

Note

You can have more than one **Canvas** in your scene, each with its own children.

When you create a Canvas, it will appear as a large rectangle within your scene. It will be significantly larger than that rectangle representing the camera's view:

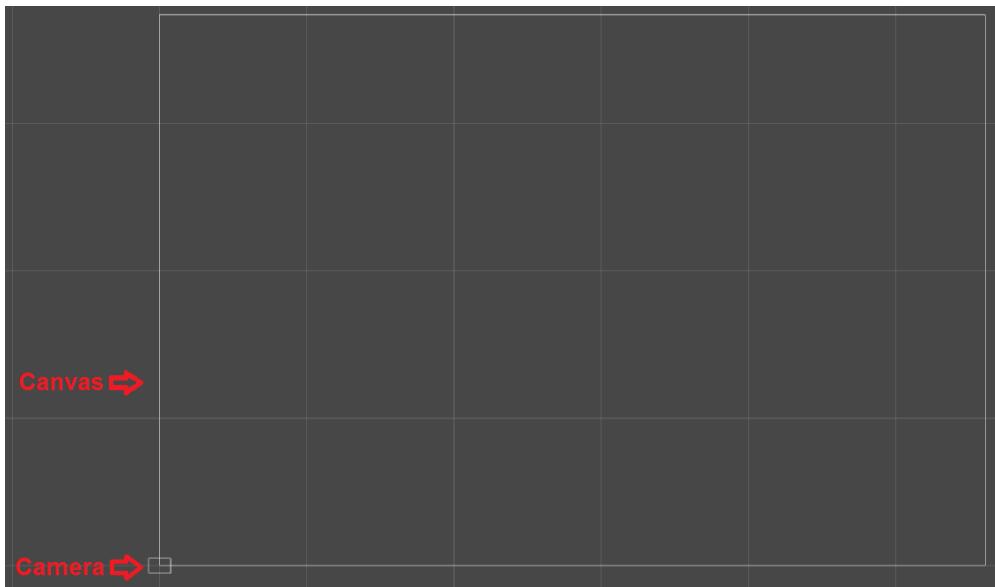


Figure 6.3: The renderable UI elements within the Unity UI (uGUI) system

The Canvas is larger than the Camera because the Canvas component has a scaling mode on it. The scaling mode by default equates to one pixel within the UI to one Unity unit, so it's a lot bigger. A nice consequence of this large size is that it is really easy to see your UI items as a somewhat separate entity, and this keeps it from cluttering up your camera view.

Every newly created Canvas automatically comes with four components: **Rect Transform**, **Canvas**, **Canvas Scaler**, and **Graphic Raycaster**, as shown in the following screenshot:

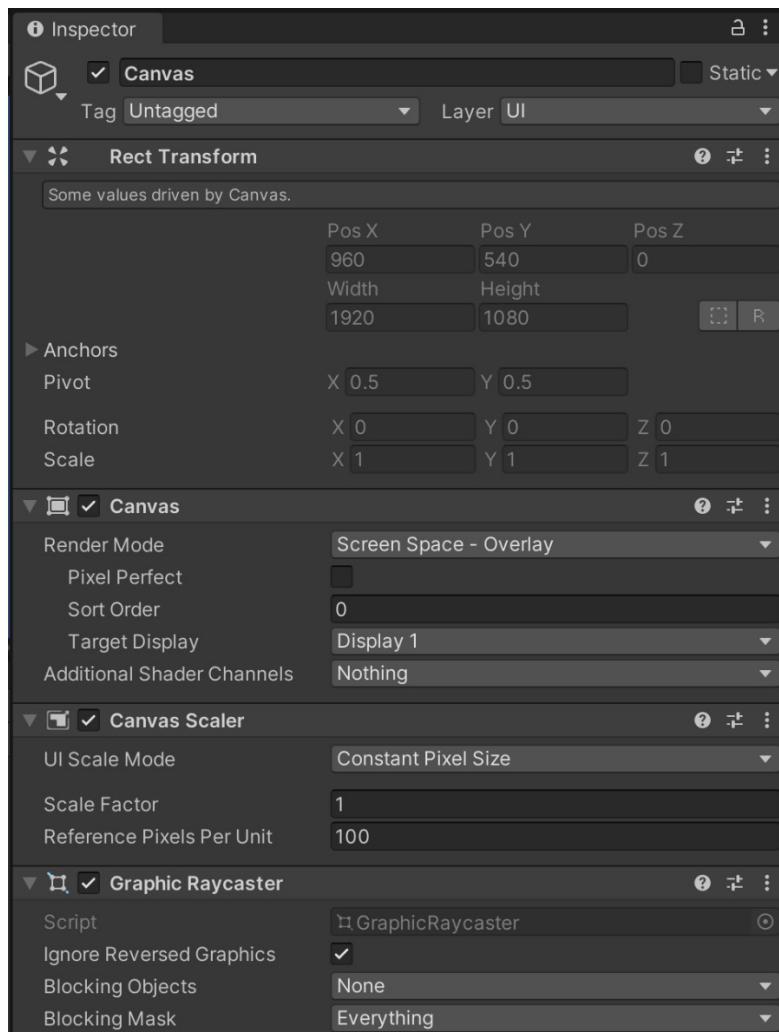


Figure 6.4: The components of a Canvas GameObject

Let's explore each of these components.

Rect Transform component

Every Unity UI GameObject has a **Rect Transform** component as its first component. This component is very similar to the **Transform** component on non-UI GameObjects in that it allows you to place the object within the scene.

You'll note that when you first place a Canvas in the scene, you can't adjust the values within the **Rect Transform**, and there is a message stating, "**Some values driven by Canvas**", as shown in the

preceding screenshot. This message means you cannot control the position of the Canvas because the properties selected in the **Canvas** component determine them.

When a Canvas component has its **Render Mode** set to **Screen Space-Overlay** or **Screen Space-Camera**, the adjustment of the **Rect Transform** component's values is disabled. In these two modes, the values are determined by the resolution of the game display because the Canvas fills up the full-screen area. When the Canvas' **Render Mode** is set to **World Space**, you can adjust the values as you see fit, as this component will then determine its location within the scene. We will discuss how to use the **Rect Transform** Component more thoroughly later in this chapter, but for now, let's review the Canvas component and the various render modes more thoroughly.

Canvas component

The **Canvas** component allows you to select the Canvas **Render Mode** from a dropdown. There are three render modes: **Screen Space-Overlay**, **Screen Space-Camera**, and **World Space**. The different render modes determine where in the scene the UI elements will be drawn and how the **Rect Transform** component will be used.

When developing your UI, the first thing you should always do is appropriately set your Canvas' render mode based on your needs. If you have multiple Canvases in your scene, they can each have a different render mode. Changing the render mode will change the properties available on the Canvas component. Let's review the purpose of each of the render modes and the properties that come with them.

Screen Space-Overlay

Screen Space-Overlay is the default render mode. If you are asked to think of a video game UI, it's highly likely that you will envision one rendered in **Screen Space-Overlay**. This render mode overlays all the UI elements within the Canvas in front of everything in the scene as if it is drawn on top of the screen. So, UI items like **heads-up-displays** (HUDs) and pop-up windows that appear on the same plane as the screen will be contained within a **Screen Space-Overlay Canvas**.

Remember, when a Canvas is using the **Screen Space-Overlay** render mode, you cannot adjust the **Rect Transform** component of the Canvas. This is because the canvas will automatically resize based on the size of the screen (not the camera).

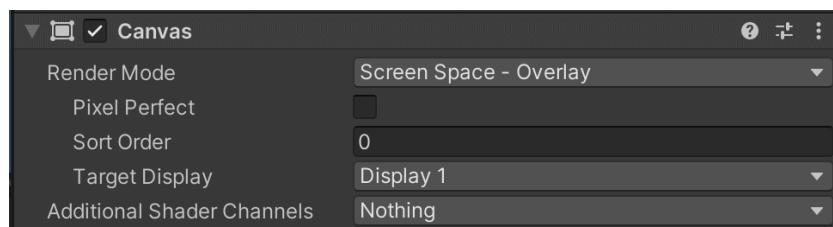


Figure 6.5: Screen Space – Overlay Render Mode properties

When you have **Screen Space-Overlay** selected, the following properties become available:

- **Pixel Perfect:** Selecting this check box will cause the elements rendered in the Canvas to line up with pixels. It can make the UI elements appear sharper and less blurry. This can cause performance issues, so only use it if absolutely necessary.
- **Sort Order:** This option will determine the order in which all the **Screen Space-Overlay** Canvases in your scene will render. You can think of it as a stacking order. The higher the number, the *closer* the items within the Canvas will appear to the person viewing the scene. In other words, higher sort order numbered canvases will appear *on top* of lower sort order numbered canvases.
- **Target Display:** If you are creating a PC, Mac, Linux Standalone game, you can have different cameras display on up to eight different monitors. You can also have different UI for each monitor. This is where you will tell the Canvas which of the displays it will render on.
- **Additional Shader Channels:** Shaders are essentially algorithms that describe the color of a GameObject based on light and material. Each Canvas automatically includes the following shader channels: Position, Color, and Uv0. However, this property lets you add additional channels. Shaders are a pretty complicated topic, so we won't spend a lot of time discussing them in this text. In **Screen Space-Overlay** mode, not all of the channels available in the dropdown menu work, and you'll see a message in the component describing which are not working.

Next, let's look at **Screen Space Camera**.

Screen Space-Camera

Screen Space-Camera performs similarly to **Screen Space-Overlay**, but it renders all UI elements as if they are a specific distance away from the camera. As you can see from the following screenshot, if there is no **Render Camera** selected, this render mode works exactly like the **Screen Space-Overlay** mode (as indicated by the warning message):

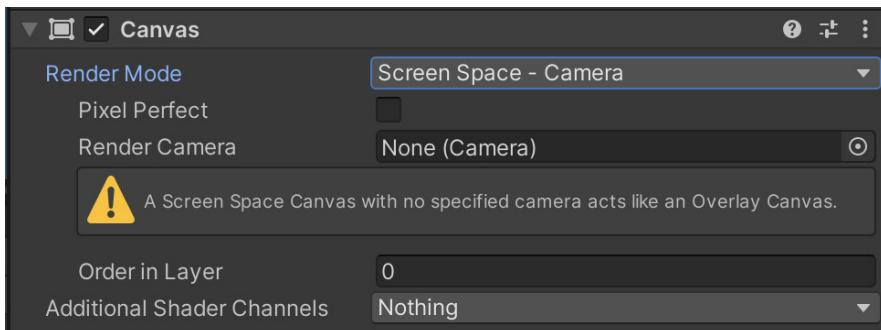


Figure 6.6: Warning message for Screen Space – Camera Render Mode

You can assign any camera in your scene to the **Render Camera** in the **Screen Space-Camera** render mode. This is the camera to which the canvas will draw. Once you add a camera to the **Render Camera** slot, the warning message will disappear, and new options will be made available to you, as shown in the following screenshot:

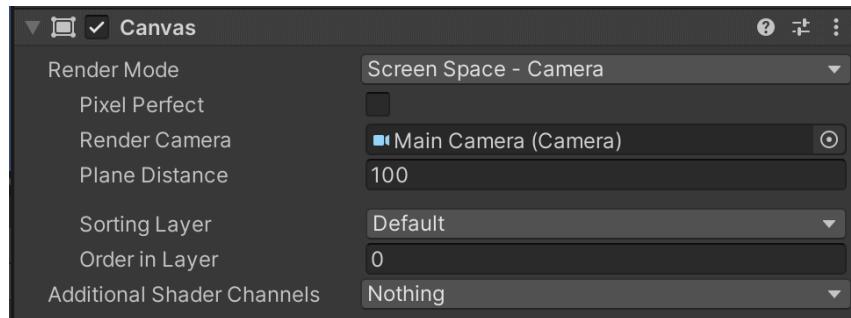


Figure 6.7: Screen Space – Camera Render Mode properties

When you have **Screen Space-Camera** selected, the following properties become available:

- **Pixel Perfect:** This is the same option as with **Screen Space-Overlay**.
- **Render Camera:** As stated earlier, this is the camera to which the canvas will draw.
- **Plane Distance:** This property tells the Canvas how far away from the camera it should display.
- **Sorting Layer:** This property allows you to choose which **Sprite Sorting Layer** to display the Canvas with.
- **Order in Layer:** This property determines the order in the **Sprite Sorting Layer** (chosen earlier) that the Canvas will display. This order works similar to the way **Sort Order** works, with higher numbers appearing on top of lower numbers.
- **Additional Shader Channels:** This property works as it does in **Screen Space-Overlay**.

This rendering mode is helpful if you want a Canvas to render from a different perspective than that of your main camera. It is also helpful for creating a static background that will consistently scale with the camera in a 2D game. Since you can use **Sprite Sorting Layer** with this rendering mode, you can make sure the Canvas containing the background always renders behind all other objects in the scene.

Remember, when a Canvas is using the **Screen Space-Camera** render mode, you cannot adjust the **Rect Transform** component of the Canvas. This is because the canvas will automatically resize based on the size of the camera (not the screen).

World Space

The last rendering mode is **World Space**. This mode allows you to render UI elements as if they are physically positioned within the world.

In **Screen Space-Overlay** and **Screen Space-Camera**, you cannot adjust the properties of the **Rect Transform** component. The positions of UI elements within Canvases with those two rendering modes do not translate to world space coordinates and are instead relative to the screen and camera. However, when a Canvas is in **World Space** render mode, the values of the **Rect Transform** can be adjusted because the coordinates of the UI elements are based on actual positions within the scene. These Canvases do not have to face a specified camera as the other two Canvas types do.

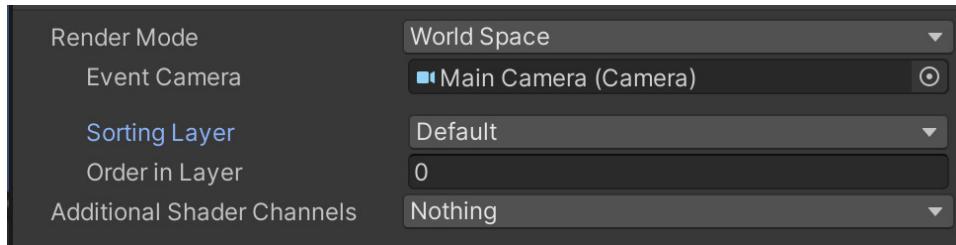


Figure 6.8: Render Mode – World Space properties

This mode requests an **Event Camera**, rather than a **Rendering Camera** as **Screen Space-Camera** mode requested. An **Event Camera** is different than a **Rendering Camera**. Since this Canvas is in the **World Space**, it will be rendered with the **Main Camera**, just as all the other objects that exist within the scene. The **Event Camera** is the camera that will receive events from the **EventSystem**. So, if items on this Canvas require interactions, you have to include an **Event Camera**. If the player won't be interacting with the items on the Canvas, you can leave this blank.

The reason you need to specify an **Event Camera** is ray casting. When the user clicks on or touches the screen, a ray (one-directional line) is cast infinitely forward from the point of click (or touch) into the scene. The direction it points is determined by the direction the camera is facing. Most of the time, you will set this as your **Main Camera** because that is the direction in which the player will expect the events to occur.

When you have **World Space** selected, the following properties become available:

- **Event Camera:** As stated earlier, the camera assigned to this slot determines which camera will receive the events of the Canvas
- **Sorting Layer:** This property is the same as in **Screen Space-Camera**
- **Order in Layer:** This property is the same as in **Screen Space-Camera**
- **Additional Shader Channels:** This property works as it does in **Screen Space-Overlay**

Next, let us look at the **Canvas Scalar** component.

Canvas Scalar component

The **Canvas Scalar** component determines how the items within the canvas will scale. It also determines the pixel density of the items within the UI Canvas.

In *Chapter 1*, we discussed how to build your game at a single resolution or a single aspect ratio. However, most of the time, you will not have the luxury of choosing the resolution or aspect ratio of your game. You'll note that I only mentioned specifying the aspect ratio and resolution for the PC, Mac, and Linux Standalone builds and the WebGL builds. When you build to something that will play on a handheld screen or a TV screen, you cannot guarantee how big that screen will be.

Due to the fact that you cannot guarantee the resolution or aspect ratio of your game, the need for your UI to adjust to various resolutions and scaling is very important; that's why this **Canvas Scalar** component exists.

The **Canvas Scalar** component has four **UI Scale Modes**:

- **Constant Pixel Size**
- **Scale With Screen Size**
- **Constant Physical Size**
- **World**

The first three **UI Scale Modes** are available when the Canvas **Render Mode** is set to **Screen Space-Overlay** or **Screen Space-Camera**. The fourth **UI Scale Mode** is automatically assigned when the Canvas **Render Mode** is set to **World Space** (it cannot be changed when set).

Constant Pixel Size

When a Canvas has the **UI Scale Mode** set to **Constant Pixel Size**, every item in the UI will maintain its original pixel size regardless of the size of the screen. You'll note that this is the default setting, so by default UI does not scale; you must turn the setting on for it to scale by altering screen resolutions.

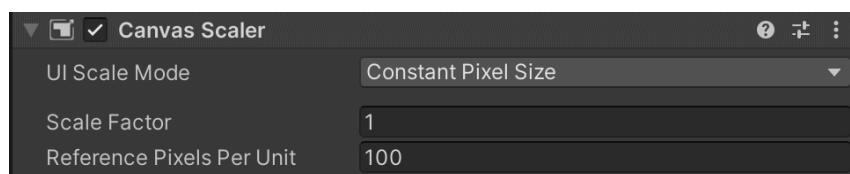


Figure 6.9: Constant Pixel Size UI Scale Mode properties

When you change the **UI Scale Mode** to **Constant Pixel Size**, you will see the following properties appear within the inspector:

- **Scale Factor:** This setting creates a scale multiple for all objects within the UI. For example, if you set this number to 2, everything within the UI will then double in size. If you set this number to 0.5, all items will have their size halved.
- **Reference Pixels Per Unit:** This setting determines how many pixels take up a single in-game unit. When this number is set to 100, that means two objects that are one game unit apart from each other will be 100 pixels apart. Put another way, if two objects are at the same y-coordinate, but one object has an x-coordinate of 1, and the other has an x-coordinate of 2, they are exactly 100 pixels apart. This property works the same way in all other modes, so I will not discuss it in the next sections.

Scale With Screen Size

When you have the **Canvas Scalar** component set to **Scale with Screen Size**, UI elements on the Canvas will scale based on a **Reference Resolution**. If the screen is larger or smaller than the **Reference Resolution** value, the items on the Canvas will then scale up or down accordingly. In *Chapter 1*, I told you that you should decide on a default resolution that represents the ideal resolution of your UI design. This default resolution will be the **Reference Resolution**.

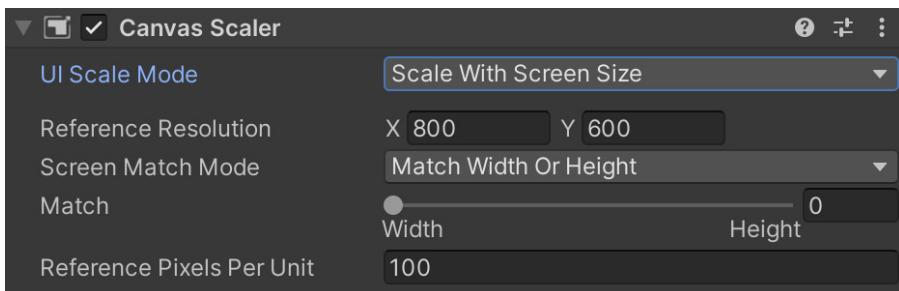


Figure 6.10: Scale With Screen Size UI Scale Mode properties

If the aspect ratio of your screen matches the **Reference Resolution** value, then things will scale up and down without any problems. If it does not match, then you need to use the **Canvas Scalar** component to define how items will scale if the aspect ratio changes.

This can be done by using the **Screen Match Mode** settings. In the following list are the three different **Screen Match Modes** that determine how the Canvas will scale if the game's aspect ratio does not match the **Reference Resolution** aspect ratio:

- **Match Width Or Height:** This will scale the UI with respect to the reference height or the reference width. It can also scale based on a combination of both.

- **Expand:** If the screen is smaller than the **Reference Resolution**, the canvas will be expanded to match that of the **Reference Resolution**.
- **Shrink:** If the screen gets larger than the **Reference Resolution**, the canvas will be reduced to match that of the **Reference Resolution**.

The **Expand** and **Shrink Screen Match Modes** do not have any further properties to edit; they just “do their own thing.” However, the **Match Width Or Height Screen Match Mode** does have a **Match** property. This property is a sliding scale that can be adjusted between 0 (**Width**) and 1 (**Height**).



Figure 6.11: Scale With Screen Size UI Scale Mode properties

When the value of **Match** is set to 0, the **Canvas Scaler** will force the Canvas to always have the same width specified by the **Reference Resolution**. This will maintain the relative scales and positions of objects along the width of the Canvas. So, objects will not get further away from or closer to each other in the horizontal direction. However, it will completely ignore the height. So, objects can get further from or closer to each other in the vertical direction.

Setting the **Match** value to 1 will accomplish the same thing but will maintain the positions and scales of the objects along the height, not the width.

Setting the **Match** value to 0.5 will compare the game's width and height to that of **Reference Resolution**, and it will try to maintain the distances between objects in both the horizontal and vertical directions.

The **Match** value can be any number between 0 and 1. If the number is closer to 1, scaling will favor the height, and if it is closer to 0, it will favor the width.

None of these **Match** settings will be perfect for all games at all aspect ratios and resolutions. The settings you choose will depend on how you want the UI to scale. If you want the relative vertical positions to be maintained, use **Height** (1). If you want the relative horizontal positions to be maintained, use **Width** (0). It really just depends on which spacing you care the most about.

I recommend using the following settings based on the orientation of your game:

Orientation	Match Value
Portrait	0 (Width)
Landscape	1 (Height)
Varies	0.5 (Width and Height)

Table 6.1: Orientation and Match Value

I chose these settings based on whichever of the two numbers on the **Reference Resolution** is the smallest. In **Portrait** mode, the width will be the smallest, so I find it important to maintain the relative position of the items in the width. This is a personal preference and just a recommendation, and it will not necessarily make sense for all games. However, I have found it to be a good rule of thumb for most games.

It is best to avoid making games that will vary between portrait and landscape mode unless you have minimal UI or are very comfortable with creating scalable UI.

Constant Physical Size

When a Canvas has the **UI Scale Mode** set to **Constant Physical Size**, every item in the UI will maintain its original physical size, regardless of the size of the screen. Physical size references the size that will appear to the user if they were to take out a ruler and measure it on their screen. Much like **Constant Pixel Size**, items on Canvases with this **UI Scale Mode** setting will not scale.

If you had a UI item that you wanted to always be a specific width and height, you'd put it on a Canvas that has this **UI Scale Mode**. For example, if you wanted a button to always be 2 inches wide and 1 inch tall, you'd use this mode. This is particularly helpful in mobile games, allowing you to make sure buttons are sized large enough for human fingers based on the recommendations we discussed in *Chapter 4*.

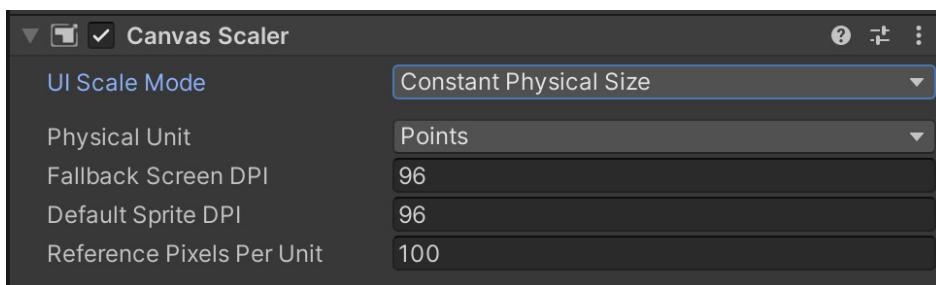


Figure 6.12: Scale with Constant Physical Size UI Scale Mode properties

When you change the **UI Scale Mode** to **Constant Physical Size**, you will see the following properties appear within the inspector:

- **Physical Unit:** The unit of measure. You can select from **Centimeters**, **Millimeters**, **Inches**, **Points**, and **Picas**.
- **Fallback Screen DPI:** If the DPI is unknown, this is the assumed DPI.
- **Default Sprite DPI:** The DPI of all sprites with Pixels Per Unit that are equal to the **Reference Pixels Per Unit** value.

World

World is the only **UI Scale Mode** available for Canvases set to **World Space**. You'll see from the following screenshot that the mode cannot be changed:



Figure 6.13: Scale with World UI Scale Mode properties

When you change the **UI Scale Mode** to **World**, you will see the following property appear within the inspector:

- **Dynamic Pixels Per Unit:** This is the Pixels Per Unit setting for all dynamic UI items (such as text).

Graphic Raycaster component

The **Graphic Raycaster** component allows you to check to see whether objects on the Canvas have been hit by a user input using the Event System. As discussed when looking at the **World Space** Canvas **Render Mode**, when a user touches the screen, a ray is cast forward from the point on the screen at which the player touches. The **Graphic Raycaster** checks these rays and sees if they hit something on the Canvas.

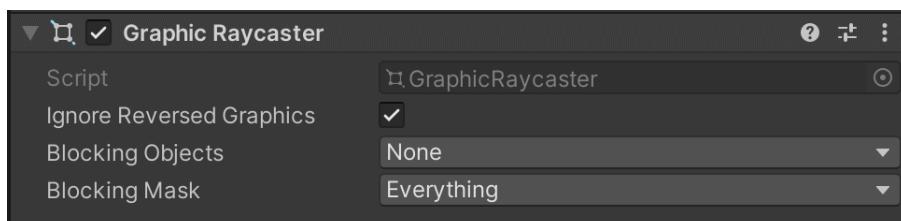


Figure 6.14: The Graphic Raycaster component

You can adjust the following properties on the **Graphic Raycaster** component:

- **Ignore Reversed Graphics:** If a UI element is facing away from the player, having this selected will stop the hit from registering. If it is not selected, hits will register on back-facing UI objects.
- **Blocking Objects:** This setting specifies which types of items in front of it will block it from being hit. So, if you select **Two D**, any **Two D** object in front of the items on this Canvas will stop the items from being interacted with. However, 3D objects will not stop the interaction. The possible options are shown here:

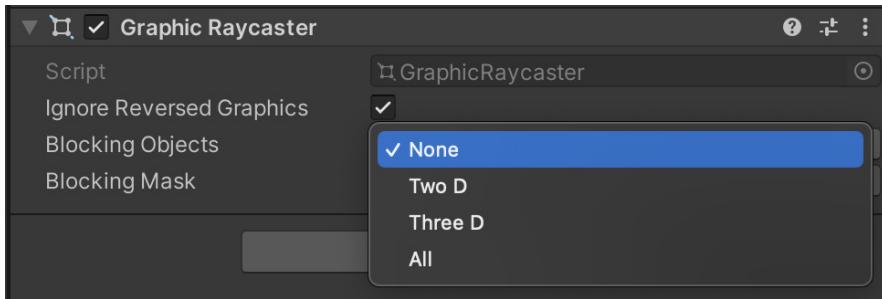


Figure 6.15: Graphics Raycaster Blocking Objects options

- **Blocking Mask:** Selecting items on this property works similarly to the **Blocking Objects** property. This allows you to select items based on their Rendering Layer, so you can get a little more specific. The possible options are shown as follows:

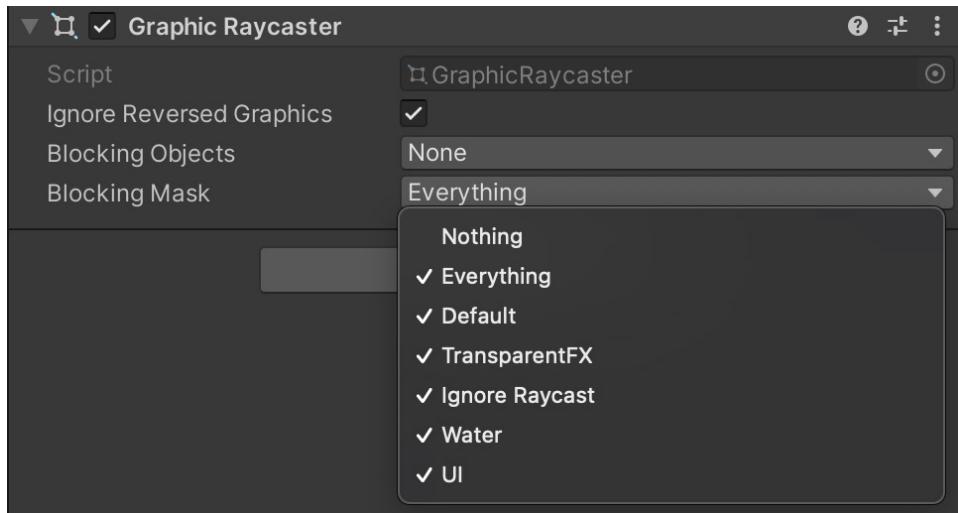


Figure 6.16: Graphics Raycaster Blocking Mask options

We will discuss Raycasting and the Event System more thoroughly in *Chapter 8*.

Canvas Renderer component

The **Canvas Renderer** component is not on a Canvas GameObject but on all renderable UI objects.

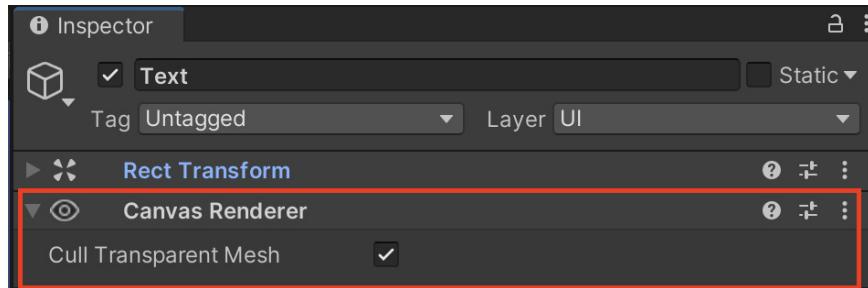


Figure 6.17: The Canvas Renderer component

For a UI element to render, it must have a **Canvas Renderer** component on it. All the renderable UI elements that you create via the + | UI menu will automatically have this component attached to them. If you try to remove this component, you will see a warning similar to the following:

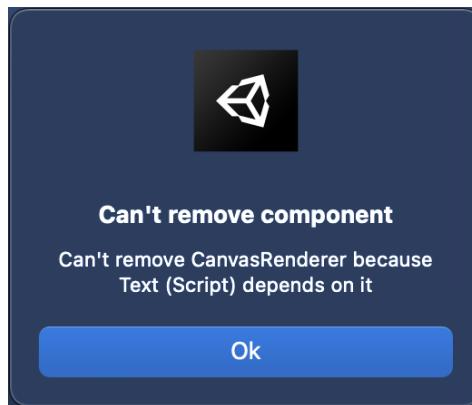


Figure 6.18: Warning message when you try to remove a Canvas Renderer component

In the preceding screenshot, I tried to remove the Canvas Renderer component from a Text UI object. As you can see, it will not let me remove the Canvas Renderer component because the Text component relies on it. If I return and remove the Text component, I would then be able to remove the Canvas Renderer component.

The only property on the **Canvas Renderer** component is the **Cull Transparent Mesh** toggle. “To cull” means to not draw. So, this property states that the renderer will not draw any geometry that has its vertex color alpha at or close to 0.

UI Panel

The main function of **UI Panels** is to hold other UI elements. You can create a Panel by selecting + | **UI | Panel**. It's important to note that there is no Panel component. Panels are really just GameObjects that have **Rect Transform**, **Canvas Renderer**, and **Image** components. So, really, a UI Panel is just a UI Image with a few properties predefined for it.

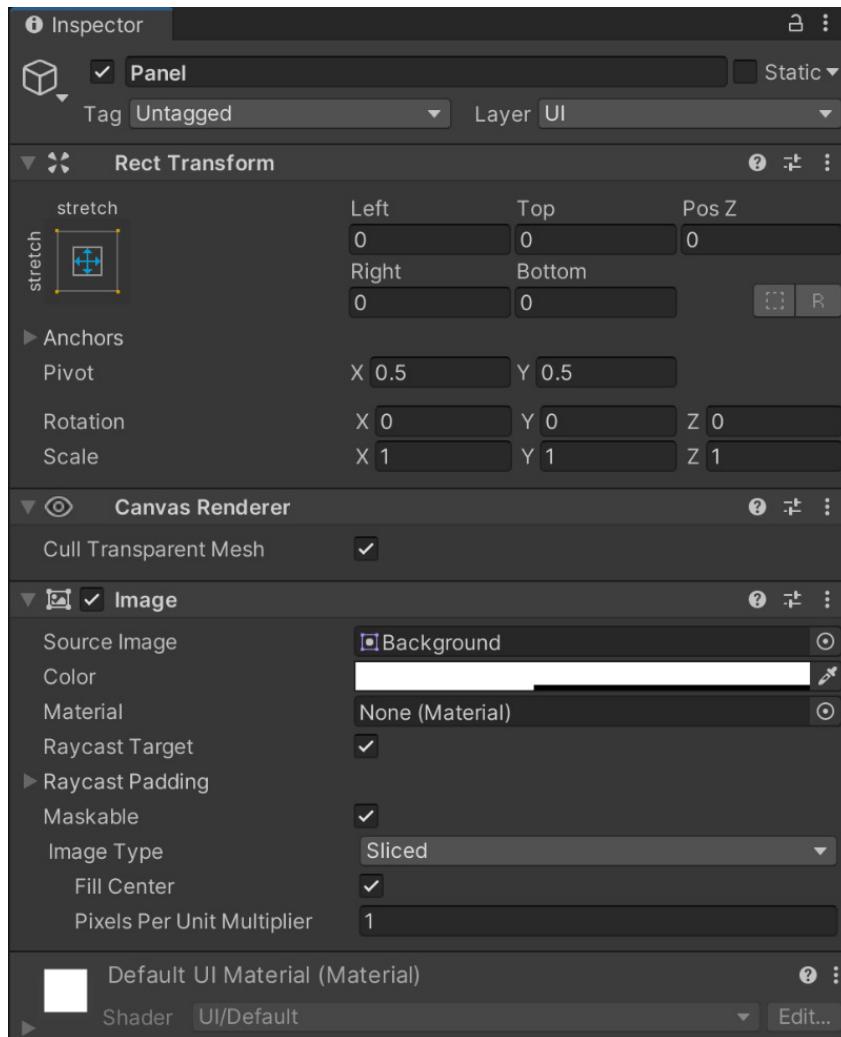


Figure 6.19: The components on a Panel GameObject

By default, Panels start with the **Background** Image (which is just a grey rounded rectangle) as the **Source Image** with medium opacity. You can replace the **Source Image** with another Image or remove the image entirely.

Panels are very useful when you are trying to ensure that items scale and are appropriately positioned relative to each other. Items that are contained within the same Panel will scale relative to the Panel and maintain their relative position to each other in the process.

We will look at the **Image** component more thoroughly soon, but now that we are looking at an object that will allow us to edit its **Rect Transform** component, let's explore that component.

Rect Transform

Each UI element has a **Rect Transform** component. The **Rect Transform** component works very similarly to the **Transform** component and is used to determine the position of the object on which it is attached.

Rect Tool

Any of the Transform tools can be used to manipulate UI objects. However, the Rect Tool allows you to scale, move, and rotate any object by manipulating the rectangle that encompasses it. While this tool can be used with 3D objects, it is most useful for 2D and UI objects.



Figure 6.20: The Rect Tool

- To move an object with the Rect Tool, select the object and then click and drag inside the rectangle.
- To resize an object, hover over the edge or corner of an object. When the cursor changes to arrows, click and drag to resize the object. You can scale uniformly by holding down the *shift* button while dragging.
- To rotate, hover at the corner of the objects--slightly outside of the rectangle until the cursor displays a rotating circle at its corner. You can then rotate by clicking and dragging.

Positioning modes

When using the Rect Tool, it is important that you have the correct positioning modes selected. You can select **Center** or **Pivot** and **Global** or **Local**. The modes will toggle by clicking on the buttons:



Figure 6.21: The various positioning modes

- When in **Center** mode, the object will move based on its center point and rotate around its center point.

- When in **Pivot** mode, the object will rotate around its pivot point rather than its center point. You can also alter the position of the pivot point in this mode by hovering over the pivot point and clicking and dragging to move.
- When in **Global** mode, the Rect Transform's bounding box will be a non-rotated box that encompasses the entire object.
- When in **Local** mode, the Rect Transform's bounding box will be a rotated box that snugly fits the object.

The following illustration shows the bounding boxes of the Rect Transform for a Panel in **Global** and **Local** modes. The empty blue circle represents the object's pivot point:

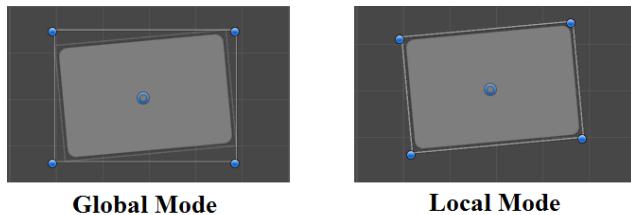


Figure 6.22: Global mode vs. Local mode

Next, let's look at the **Rect Transform** component.

Rect Transform component

As stated earlier, UI elements do not have the standard **Transform** component; they have the **Rect Transform** component. If you compare it to a standard **Transform** component, you will see that it has quite a few more properties:

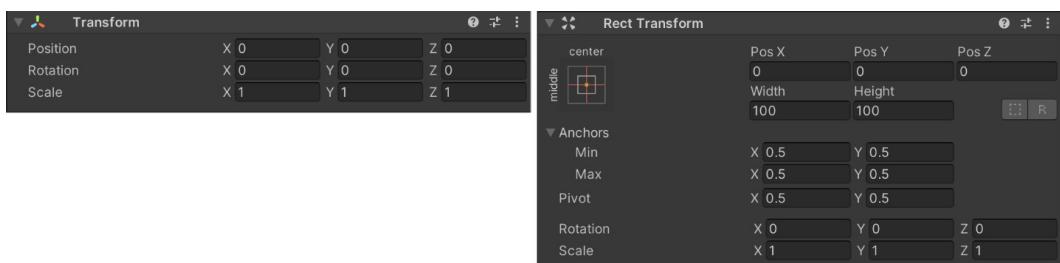


Figure 6.23: Transform vs. Rect Transform

You can use it to change the position, rotation, and scale, just as you can with **Transform**, but there are the added properties of **Anchor Presets** (represented by the square image in the top left corner), two values for determining dimension (**Anchor Min** and **Max** points), and the **Pivot** point. It's important to note that the **Scale** value is considered the local scale. If you rescale the object with the Rect Tool, even with the Local positioning mode, the values within **Scale** will remain at 1.

You may have noticed that the labels for the position and dimensional values are different in the preceding illustration than they are in the one provided in the *UI Panel Properties* section. This is because the labels that represent the position and dimension change depending on the Anchor Preset chosen. We'll discuss how to use these Anchor Presets momentarily, but let's look at some different examples of labels the position and dimensional values can hold.

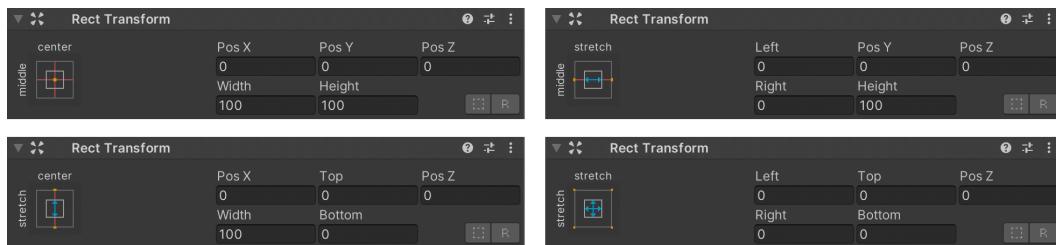


Figure 6.24: Variation in Rect Transform properties

If the **Rect Transform** has its Anchor Preset set to not include stretch, as with the top-left example in the preceding screenshot, the values for position are determined by **Pos X**, **Pos Y**, **Pos Z**, and the dimensions are determined by **Width** and **Height**.

If the **Rect Transform** has its Anchor Preset set to include stretch, as with the other three examples, the positions perpendicular to the stretch and the dimensions parallel to the stretch are labeled with **Left**, **Right**, **Top**, and **Bottom**. These values represent the offset from the border of the parent's **Rect Transform**.

The **Anchor** point of an object determines the point from which all the relative positions are measured from. The **Pivot** point determines the point from which the scaling and rotating modifiers happen. It will rotate around this point and scale toward this point. We will look at Anchors and Pivot more thoroughly in the Anchors and Pivot section.

Rect Transform edit modes

There are two different edit modes available to you within the **Rect Transform** component—Blueprint mode and Raw Edit mode—as represented by the following icons, respectively:



Figure 6.25: Rect Transform edit modes

The Blueprint mode will ignore any local rotation or scaling applied to it and will display the Rect Transform bounding box as a non-rotated, non-scaled box. The following screenshot shows the bounding box of a Panel that has been rotated and scaled with Blueprint mode turned off and turned on:

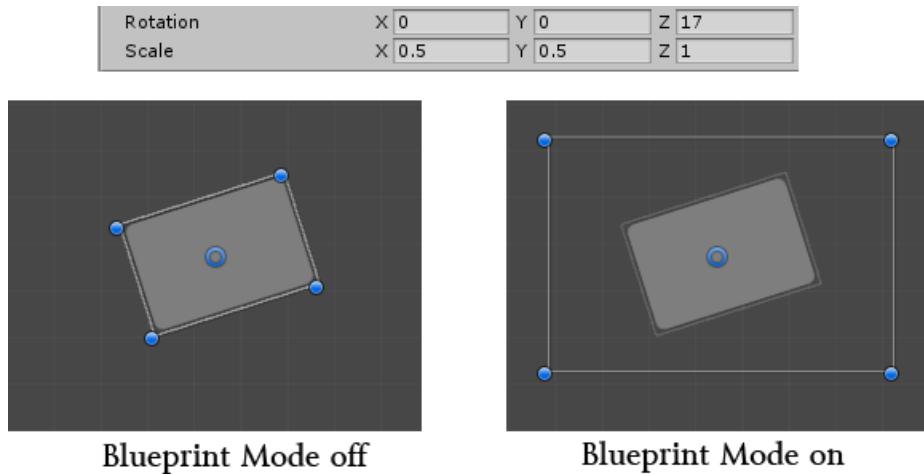


Figure 6.26: Blueprint mode on vs. Blueprint mode off

The Raw Edit mode will allow you to change the anchor and pivot points of a UI object without the object moving or scaling based on the changes you have made.

Anchor and Pivot Points

Every UI object has Anchor Handles and a Pivot Point. When used together, they will help ensure that your UI is positioned appropriately and scales appropriately if the resolution or aspect ratio of your game changes.

The Anchor Handles are represented by four triangles in the form of an X, as shown in the following diagram:

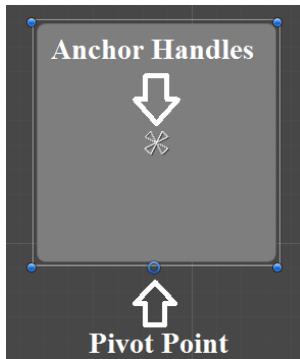


Figure 6.27: Anchor Handles and Pivot Points

The Anchors can be in a group together forming a single Anchor, as shown in the preceding diagram, or they can be split into multiple Anchors, as follows:

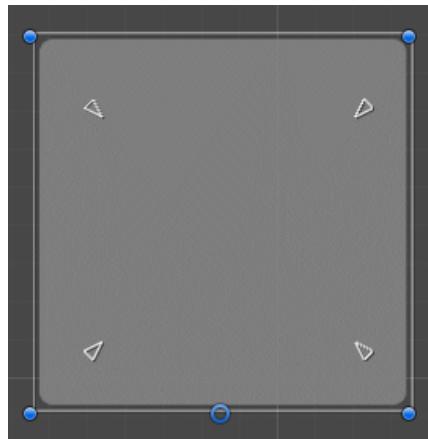


Figure 6.28: Splitting Anchor Handles

The Anchors will always form a rectangle. So, the sides will always line up.

The **Rect Transform** has properties for Anchor **Min** and Anchor **Max** points. These represent the position of the Anchor Handles relative to the parent's Rect Transform as percentages. For example, a 0 in an *x* value moves the handles all the way to the left, and a 1 moves the handles all the way to the right. You can see from the following screenshots how adjusting the *x* value will move the anchor left and right relative to the parent:

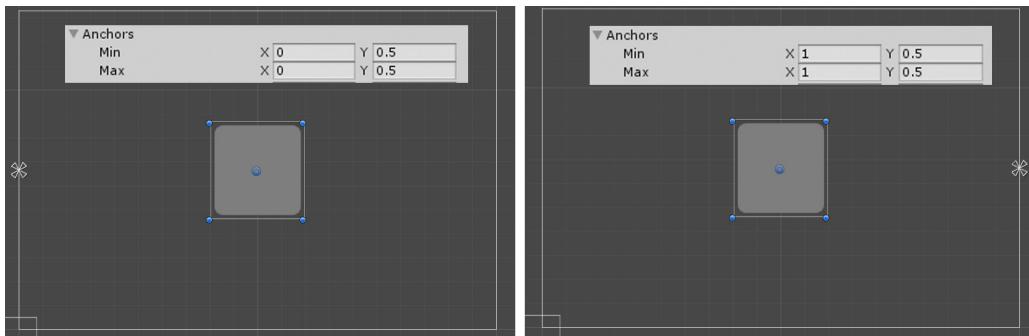


Figure 6.29: Adjusting Anchor Min and Max

The **Rect Transform** has properties for Anchor Presets and **Anchors Min** and **Max** points. The Anchor represents the point at which the UI element is connected to its parent's Rect Transform:

**Anchor
Presets**



Figure 6.30: Accessing the Anchor Presets

As a Canvas has no parent, you'll see that the Anchor Preset area is empty. This is true regardless of the Canvas Render Mode chosen:

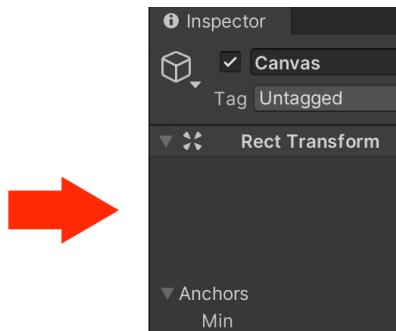


Figure 6.31: Lack of Anchor Presets on a Canvas Game Object

Clicking on the Anchor Presets box will display a list of all the possible **Anchor Presets**, as shown in the following screenshot:

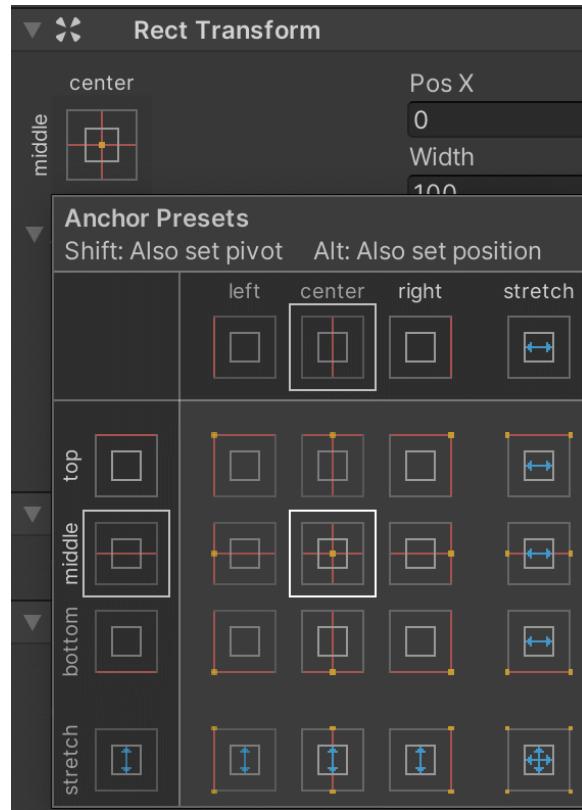


Figure 6.32: All available Anchor Presents

If you click on one of the presets, it will move the anchors to the position displayed in the screenshot. You can also adjust the position and pivot point using the anchor preset.

The images representing the presets will change if you hold down *Shift* and/or *Alt*. Holding *Shift* will show the positions for the pivot point represented by blue dots, holding *Alt* will show how the position will change, and holding both will show the pivot point and the position change.

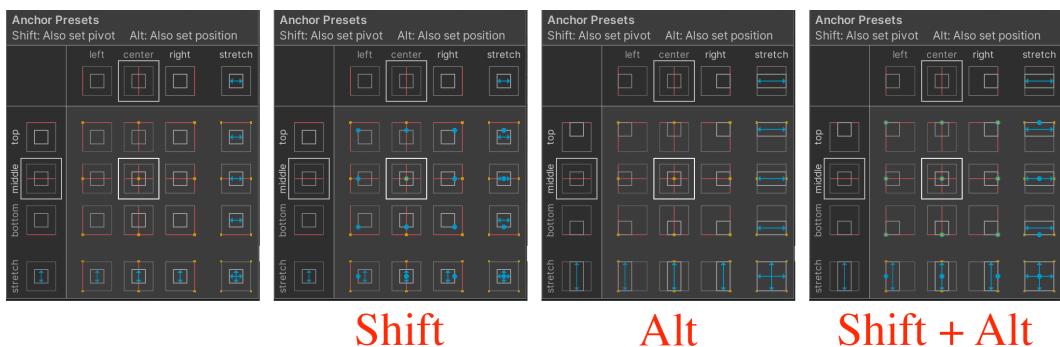


Figure 6.33: Setting pivot and position

Note

If using a Mac, since there is no *Alt* key, you will use the *Option* key instead. However, the instructions will still say *Alt* in the Editor and this does not change for the Mac version. The previous screenshots were taken on a Mac, despite it having no *Alt* key.

Now, let's look at the **Canvas Group** component.

Canvas Group component

You can add a **Canvas Group** component to any UI object. Attaching it to a UI object will let you adjust the specific properties of the object as well as all of its children with a single component rather than having to adjust these properties for each of the UI elements.

You can add a Canvas Group component to any UI object by selecting **Add Component | Layout | Canvas Group** (you can also just search for Canvas Group) from the UI object's Inspector.

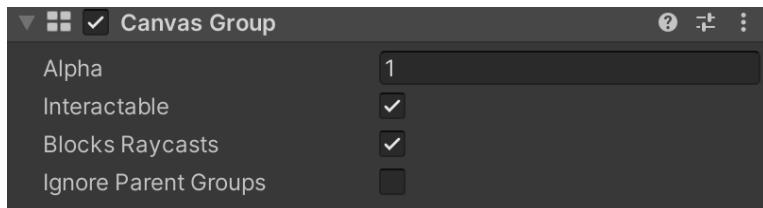


Figure 6.34: The Canvas Group component

You can adjust the following properties using a **Canvas Group** component:

- **Alpha:** This is the transparency of the UI objects within the Canvas Group. The number is between 0 and 1 and represents a percentage of opaqueness; 0 is completely transparent, while 1 is completely opaque.
- **Interactable:** This setting determines whether or not the objects within the group can accept input.
- **Blocks Raycasts:** This setting determines if the objects within the group will block raycasts from hitting things behind them.
- **Ignore Parent Groups:** If this **Canvas Group** component is on a UI element that is a child of another UI element with a Canvas Group component, this property determines whether this Canvas Group will override the one above it or not. If it is selected, it will override the parent's Canvas Group properties.

Introducing UI Text and Image

It's kind of hard to make any UI examples without using text or images. So, before we cover examples of layouts, let's first look at the basic properties of the UI Text and UI Image GameObjects. UI Text and UI Images are discussed more thoroughly in *Chapter 11* and *Chapter 12*.

When you create a new Text object using + | UI | Text, you will see that it has a **Text Component**.

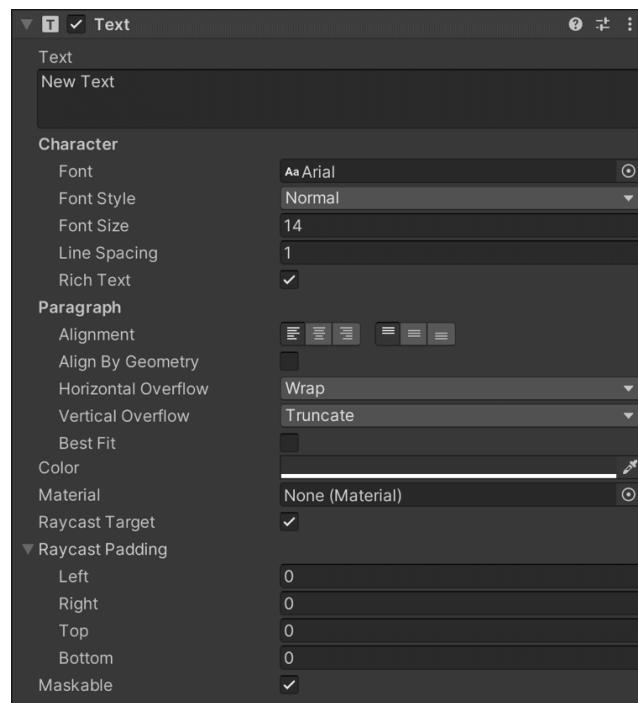


Figure 6.35: The Text component

You can change the displayed text by changing the words in the **Text** box. In *Chapter 11*, we'll take a closer look at the individual properties of the **Text** component, but for now, it should be fairly obvious what most of the properties do.

When you create a new Image object using **+ | UI | Image**, you will see that it has an **Image** component.

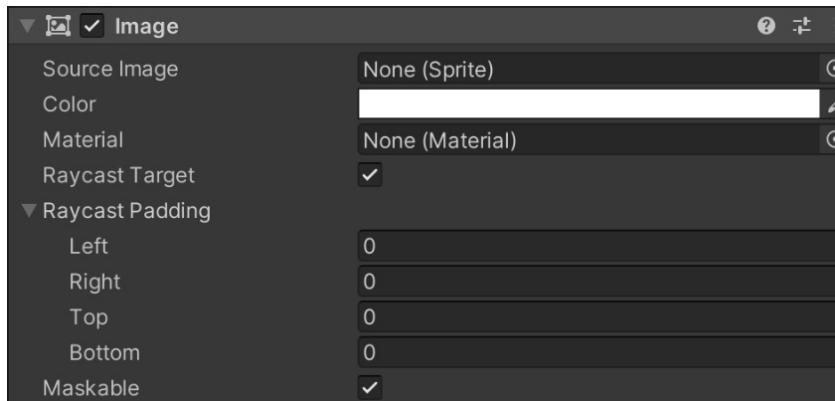


Figure 6.36: The Image component

Remember that a Panel is essentially an Image but with a few properties prefilled. When you create an Image, however, there are no prefilled properties.

We'll work with the **Source Image** property in this chapter, which allows you to change the displayed sprite. We'll look at the other properties in *Chapter 11*.

Examples

Now let's jump into some examples! We'll be creating a layout for a basic **heads-up-display (HUD)** and a background image that stretches with the screen and scales at multiple resolutions.

Before we begin building our UI, let's set up our project and bring in the art assets we will need.

We'll begin by setting up our project:

1. Create a new Unity Project and name it `Mastering Unity UI Project`. Create it in the 2D mode.

Note

We're selecting 2D Mode because it will make importing our UI sprites a lot easier. When in 2D Mode, all images import as Sprite (2D and UI) images rather than Texture images, as they do in 3D Mode. You can change to 3D Mode at any time by navigating to **Edit | Project Settings | Editor** and changing **Mode** to **3D**.

2. Create two new folders within the Assets folder named Scripts and Sprites.
3. Create a new scene and name it Chapter6.unity; ensure that you save it in the Scenes folder. You could also rename SampleScene.unity to Chapter6.unity.

We'll be using art assets that I've modified from free art assets found at the following sites:

- <https://opengameart.org/content/free-game-gui>
- <https://opengameart.org/content/cat-dog-free-sprites>

In the Chapter2/Sprites folder of the text's source files, locate the catSprites.png, pinkBackground.png, and uiElements.png images and import them into your project by dragging them into the Sprites folder.

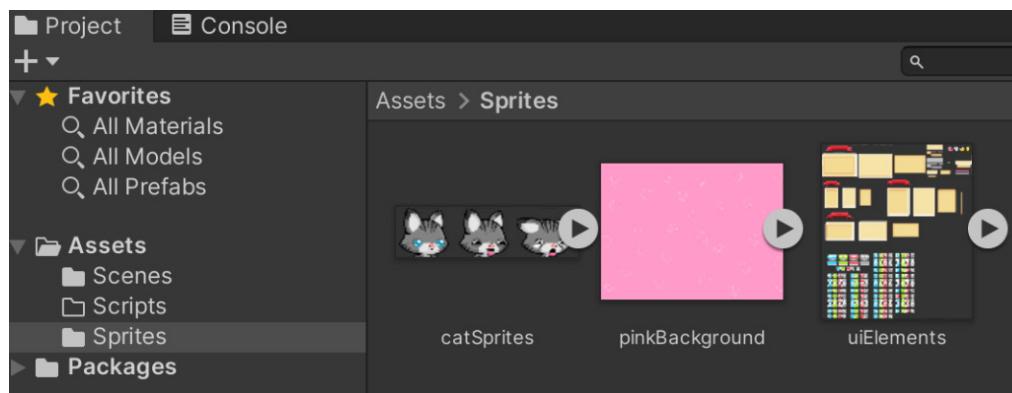


Figure 6.37: Importing the sprites

4. Now, we need to slice the sprite sheets into individual sprites. If you already know how to slice sprite sheets, do so now for the catSprites image and the uiElements image and proceed to the *Laying out the Basic HUD* section. If you are not familiar with the process, follow these steps, and continue to Step 5.
5. Select the catSprites image, hold *Ctrl*, and click on the uiElements image so that both are selected.

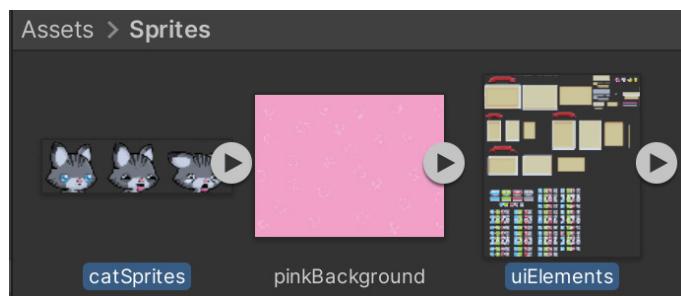


Figure 6.38: Selecting both sprite sheets

6. Now, in the **Inspector**, select **Multiple** for **Sprite Mode**. Then hit **Apply**. This will cause both the `catSprites` and `uiElements` sprites to be considered sprite sheets.

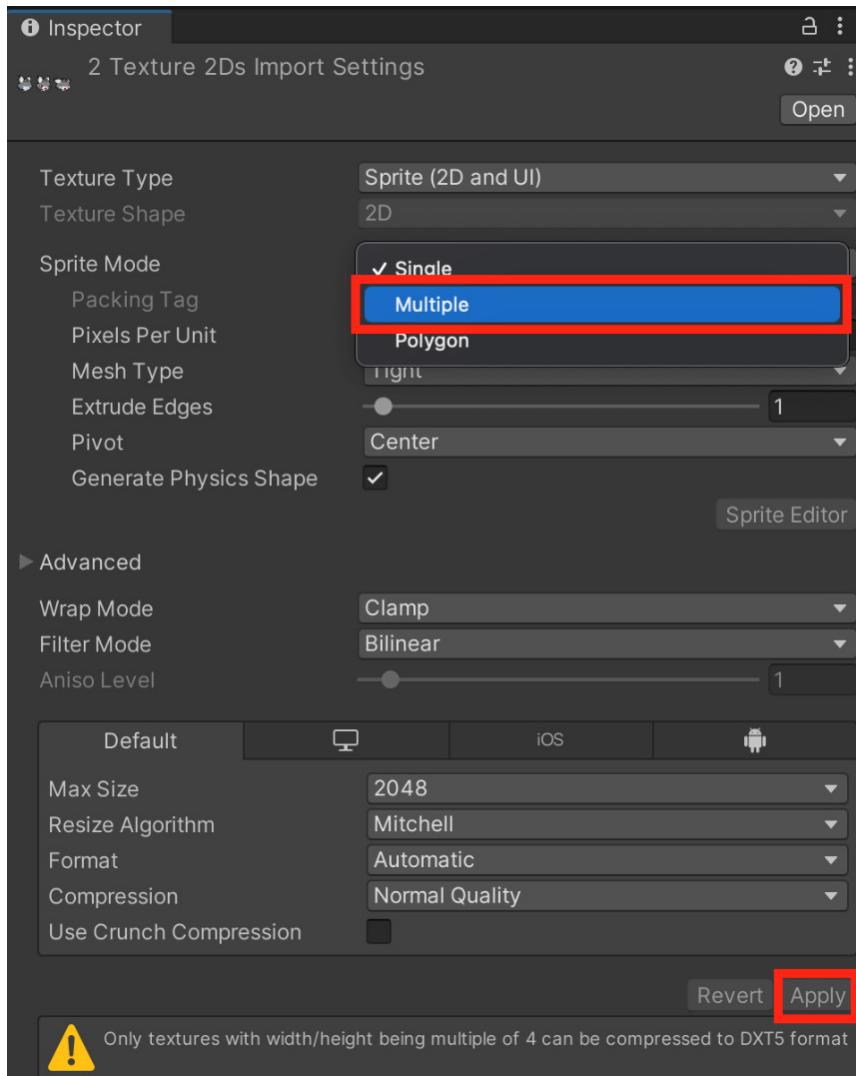


Figure 6.39: Converting the sprites to Multiple Sprite Mode

Note that the **Inspector** says **2 Texture 2Ds Import Settings** because we have selected two images.

7. Now select the `catSprites` image and open the **Sprite Editor** with the button in the **Import Settings Panel**.
8. With the **Sprite Editor** open, select **Slice**.

9. Now change the slice properties so that the **Slice Type** is **Automatic** and the sprite **Pivot** is applied to the **Bottom**. Once done, hit **Slice**.
10. You should now see the sprite broken into three separate regions. Hit **Apply** to save the changes.
11. Now, if you click on the arrow on the `catSprites` image in the project folder view, you should see the individual images:

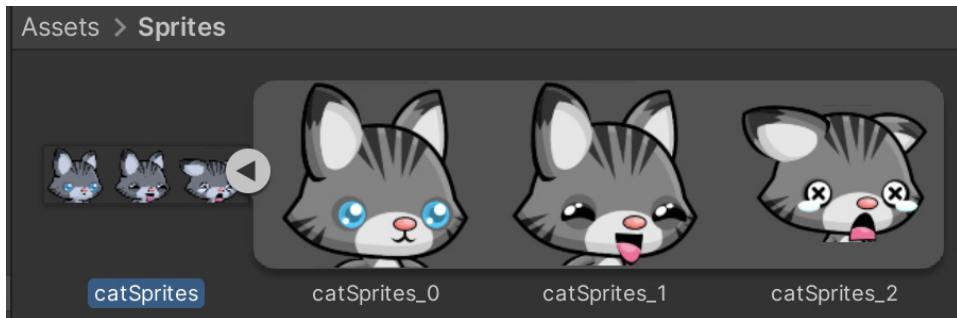


Figure 6.40: The split sprite sheet

12. Complete Steps 8 through 12 for the `uiElements` image, but set the pivot point to the **Center**.

Now that we have our project and sprites set up, we can begin with the UI examples.

Laying out a basic HUD

We will make a HUD that will look like the following:

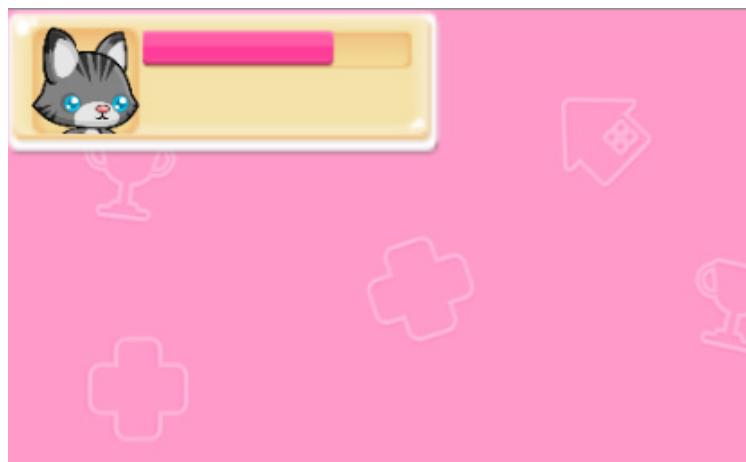


Figure 6.41: The HUD we will develop

It will be expanded upon in the upcoming chapters, but for now, it'll have a pretty simple layout that will focus on parent-child relationships and anchor/pivot point placement.

To create the HUD shown in the preceding image, complete the following steps:

1. Create a new Canvas using + | UI | **Canvas**.
2. In the Canvas **Inspector**, change the name to **HUD Canvas**.
3. It is best to set up all of your **Canvas Scaler** component information before you even begin adding elements to the UI. If you try to do it afterward, you may find that your UI elements will start rescaling. We'll work with an ideal resolution of 1024 x 768. If you look at the **pinkBackground** image (that we'll apply in the next example), it has a resolution of 2048 x 1536; 1024 x 768 has the same aspect ratio as the background image. So, set your **Canvas Scaler** component to the following settings:

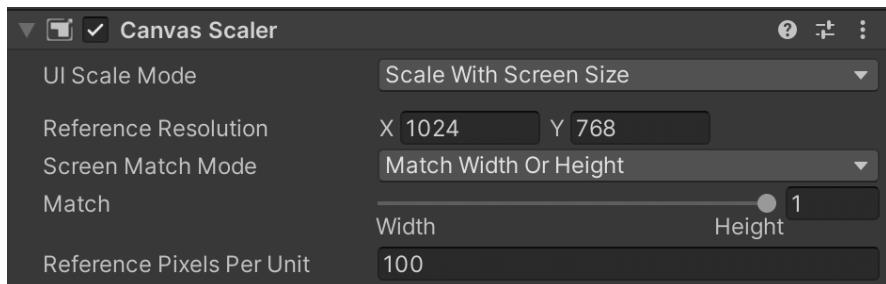


Figure 6.42: The Canvas Scaler properties

We have set the **Screen Match Mode** to **Match Width Or Height**, with the **Match** settings set to 1 so that it maintains the ratios in the vertical direction. If you remember from the *Scale with Screen Size* section, I find that this works best for most games made with a landscape resolution.

4. Set your Game view to 1024 x 768 so that you will see everything scaled appropriately (refer to the *Changing the Aspect Ratio and Resolution of the Game View* section of *Chapter 1* for directions on adding your own Game view resolution.)

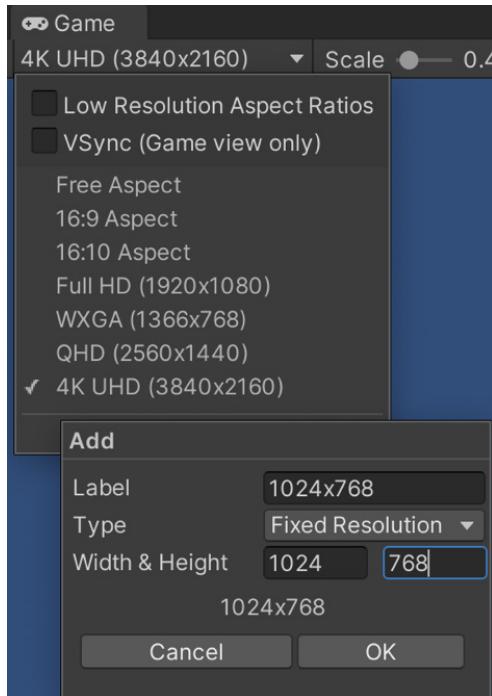


Figure 6.43: 1024 x 768 Game view resolution

5. Since we only have one Canvas, when we add any new UI elements to our scene, they will automatically be made children of our `HUD Canvas`. Create a new Panel using `+ | UI | Panel`, and rename it `HUD Panel`. You will see that it is a child of the `HUD Canvas`. This Panel will represent the rectangle that holds all the HUD elements.
6. Click on the Anchor Presets icon to open the **Anchor Presets**. Select the top-left Anchor Preset while holding down `Shift + Alt`.

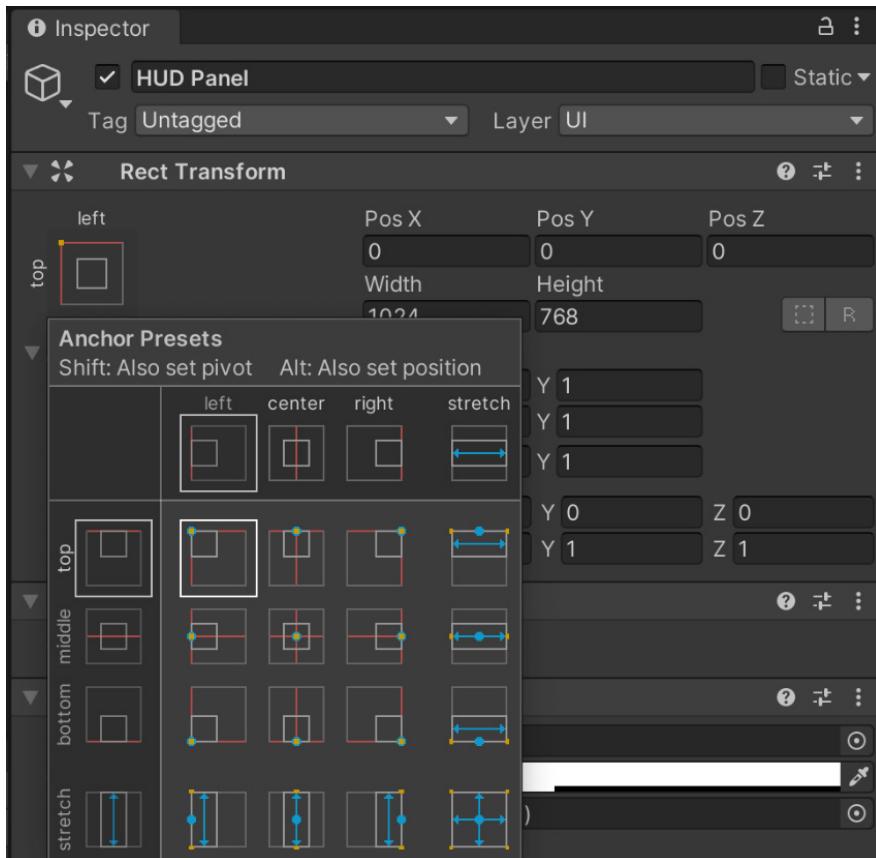


Figure 6.44: Set the Anchor Preset of the HUD Panel

7. Expand the `uiElements` image by selecting the arrow on its right. Locate the `uiElements_1` sub-image:



Figure 6.45: Set the Anchor Preset of the HUD Panel

8. Drag `uiElements_1` to the **Source Image** slot of the **Image** component on the **HUD Panel**:

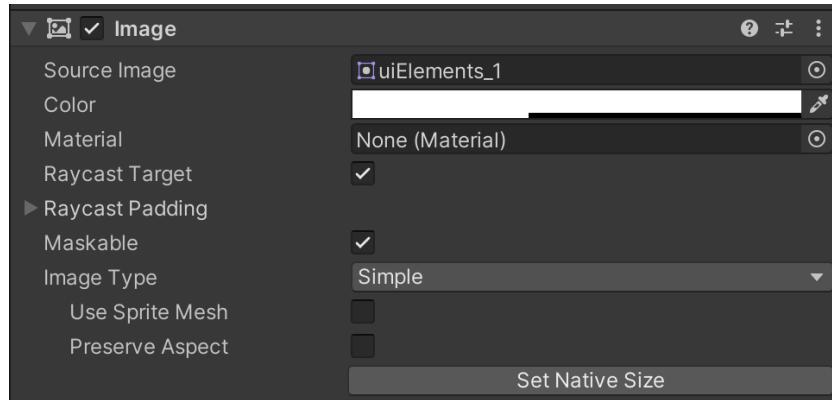


Figure 6.46: The Image component with `uiElements_1` assigned

9. Currently, the Panel is very faint and stretches across the whole screen. Let's make it easier to see by increasing the opacity. Click on the white rectangle in the **Color** slot of the **Image** component to open up a color picker. Move the Alpha slider all the way to the right or input the value 255 in the alpha value slot:



Figure 6.47: Adjusting the alpha value on the color picker to full alpha

10. Click on the checkbox next to the **Preserve Aspect** setting in the **Image** component. This property will make the image always maintain the aspect ratio of the original image, even if you set the width and height of the Image to something that does not have the same aspect ratio.

The Panel will now take up only the top portion of the scene:

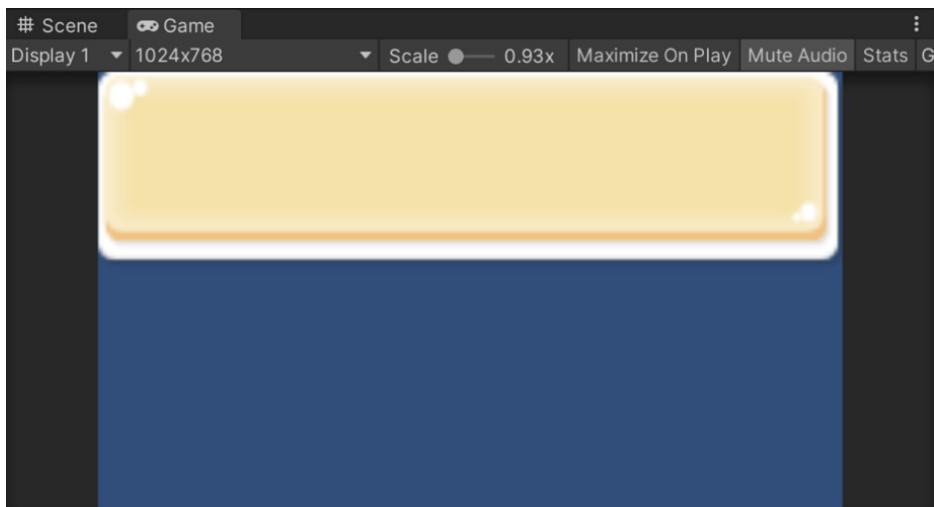


Figure 6.48: The Panel after preserving the aspect ratio

11. From the **Rect Transform** component of the **HUD Panel**, you'll note that the **Width** and **Height** are still set to 1024 and 768, respectively. You can also see more easily from the **Scene** view that the Rect Transform expands past the viewable region of the sprite. So, the object is much larger than it appears to be.



Figure 6.49: The Rect Transform exceeding the visible image area

12. Let's rescale the Rect Transform of the Panel so that it matches the size we are looking for and hugs the viewable image better. Change the **Width** to 300 and the **Height** to 102. The Rect Transform won't be a perfectly snug fit in the vertical direction, but it will be pretty close.

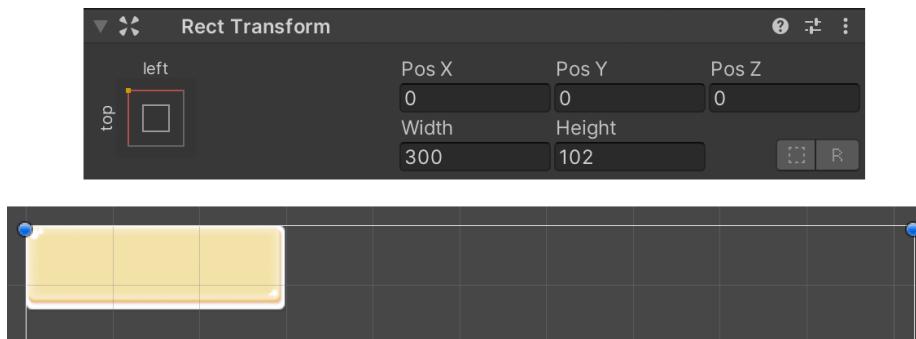


Figure 6.50: Rescaling the Rect Transform of the HUD Panel

13. We now have the main Panel set up. Since all other images will be contained within the HUD Panel, we want to make them children of the HUD Panel. That way, when the screen rescales, the other images will remain “inside” the HUD Panel and will maintain their size relative to the HUD Panel.

Let's start with the Image that holds the cat character's head. Right-click on the HUD Panel and select **UI | Image**. You'll see that it is a child of the HUD Panel. Rename it Character Holder.

14. Place the `uiElement_6` sprite in the **Source Image** slot and select **Preserve Aspect**.
15. Since the Character Holder image is a child of the HUD Panel, any anchoring we set will be relative to the HUD Panel. Choose the **left-stretch** Anchor Preset while holding **Shift + Alt**. Additionally, set the position and dimension variables as shown:

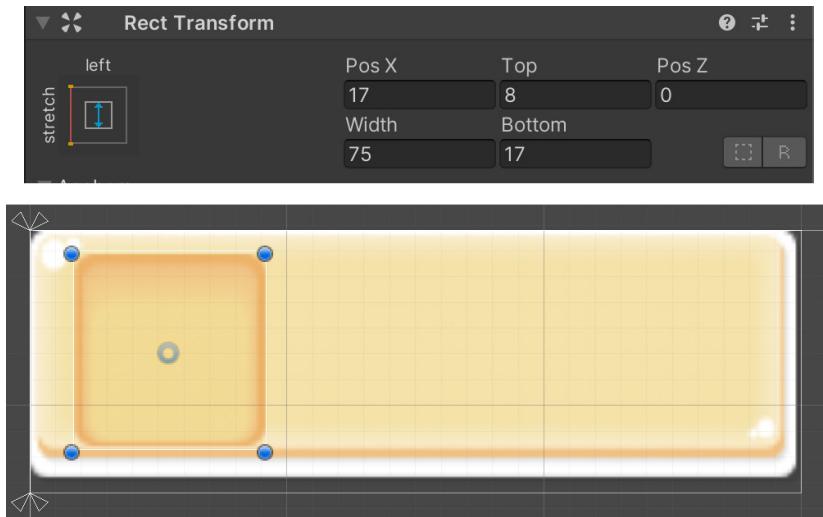


Figure 6.51: Rescaling the Rect Transform of the Character Holder

16. Let's add the Image for the cat head. We want it to fully fill out the slot represented by the Character Holder image. So, we will make it a child of the Character Holder image. Right-click on Character Holder and select **UI | Image**. Change its name to Character Image.
17. Now add the `catSprites_0` subimage to the **Source Image** of the **Image** component and select **Preserve Aspect**.
18. Set the **Anchor Preset** to **stretch-stretch** while holding *Shift + Alt*:

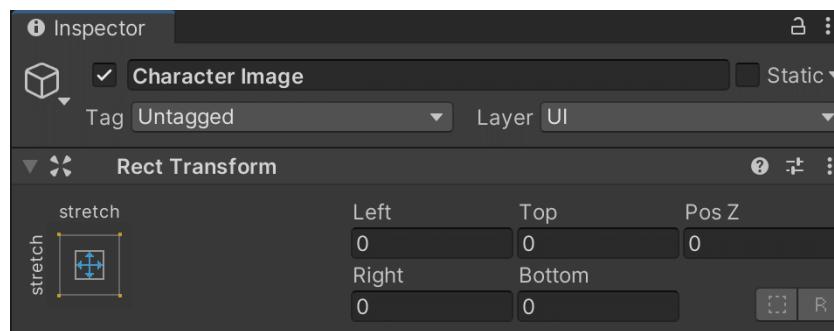


Figure 6.52: Rescaling the Rect Transform of the Character Image

Since we ensured that we have the **Rect Transform** of the `Character Holder` fit snugly around the holder's image, it should make the cat's head fit perfectly within the holder image without having to adjust any settings!

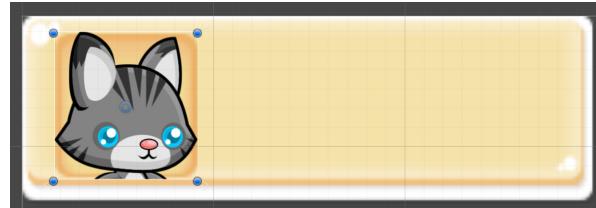


Figure 6.53: The Character Image fitting within the Character Holder

19. Now, we are ready to start making the health bar. We will create it similarly to the way we made the `Character Holder` and `Character`. Right-click on the `HUD Panel` and select **UI | Image**. Rename the new `Image Health Holder`.
20. Place the `uiElement_20` sprite in the **Source Image** slot and select **Preserve Aspect**.
21. Set the **Rect Transform** properties as shown in the following image, and ensure that you hold **Shift + Alt** when selecting the **Anchor Preset**:

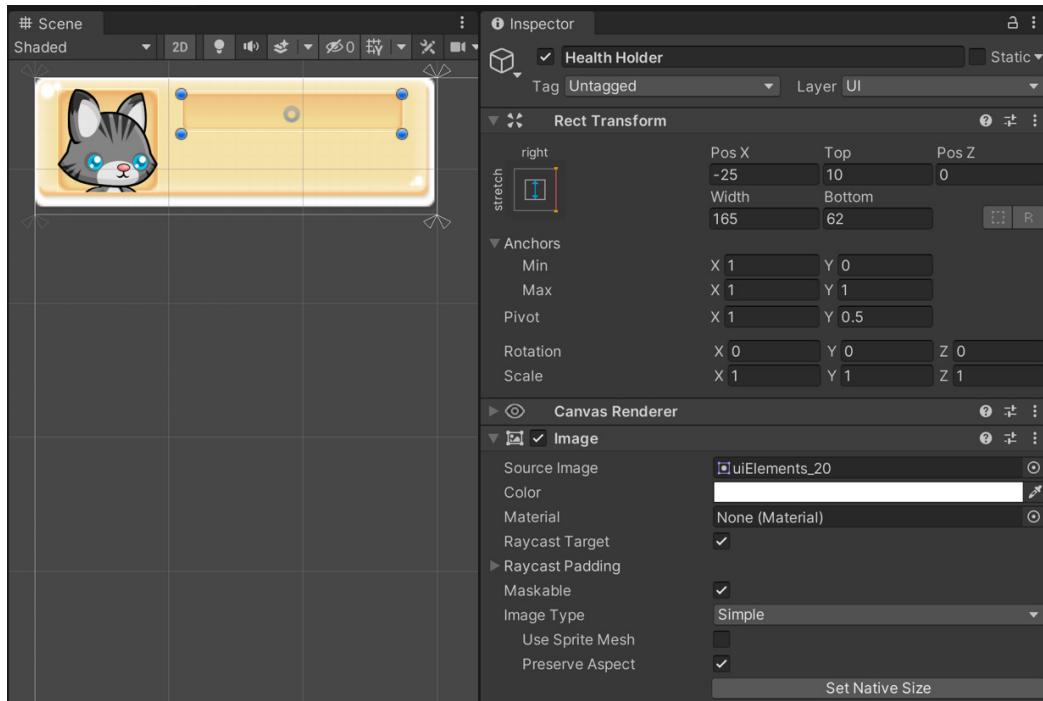


Figure 6.54: The properties of the Health Holder

22. Now, all we have left is the health bar! Just as we made the cat head Image a child of the Character Holder, we will need to make the health bar's Image a child of Health Holder. Right-click on Health Holder and select **UI | Image**. Rename the new Image **Health Bar**.
23. Place the **uiElement_23** image in the **Source Image** slot. This time, we will not be selecting **Preserve Aspect** because we want the image to scale this image horizontally.
24. Set the **Rect Transform** properties as shown in the following screenshot, and ensure that you hold *Shift + Alt* when selecting the **Anchor Presets**:

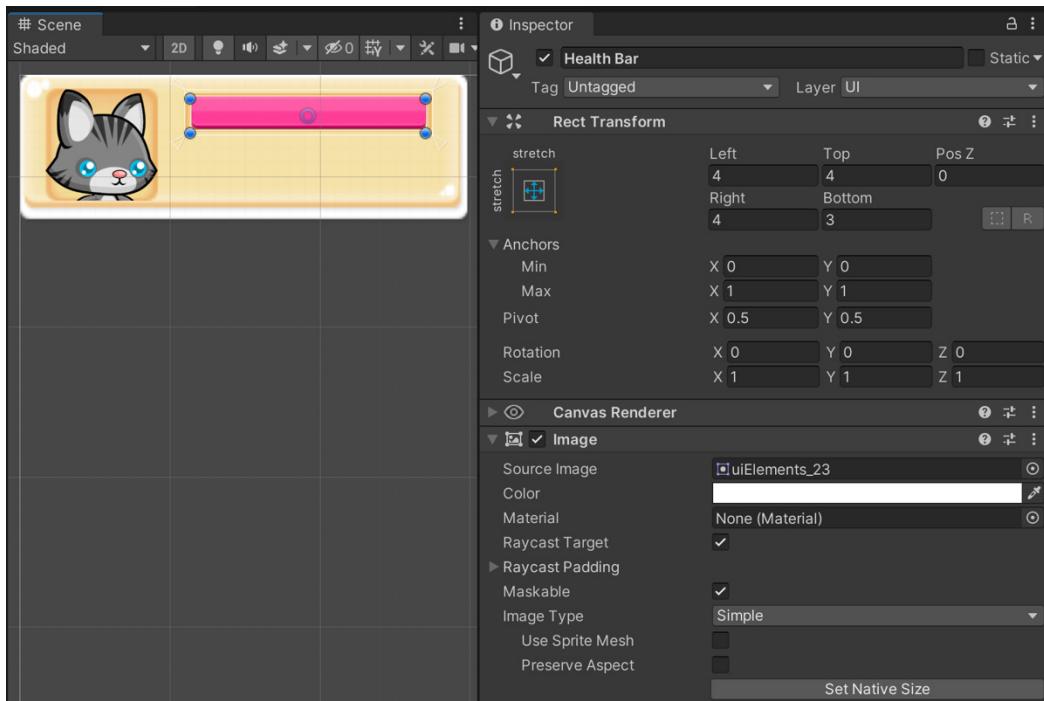


Figure 6.55: The properties of Health Holder

Note that a little padding was added so that you can see the edges of **Health Holder**.

25. Before proceeding, it is important that your Rect Transform Positioning mode is set to **Pivot**. Otherwise, you will not be able to move the pivot in the next step.

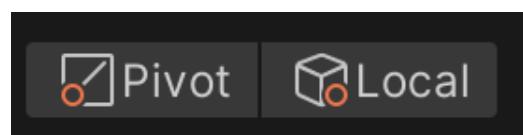


Figure 6.56: The Rect Transform Positioning mode

26. We're almost done! Right now, the pivot point of the image is right at the center. This means if we try to scale it, it will scale toward the center. However, we want it to be able to scale toward the left. So, open the **Anchor Presets**, and while holding *Shift* only, select **middle-left**. This will cause only the pivot point to move.

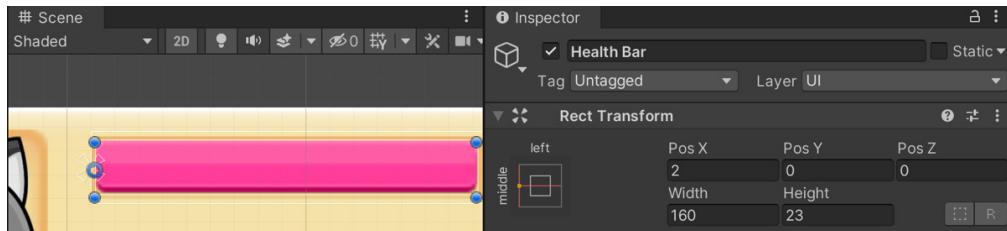


Figure 6.57: Moving the pivot point

27. Now, when we adjust the **Scale X** value on the **Rect Transform**, the health bar will scale toward the left:

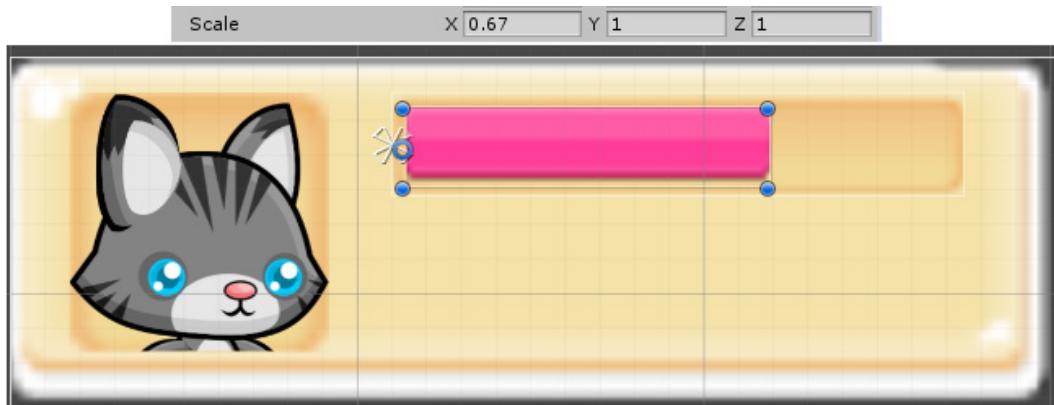


Figure 6.58: Adjusting the scale of the health bar

That's it for our HUD example! Try changing your Game view's aspect ratio to different settings so that you can see the Panel scale appropriately and see all the object-relative positions maintained.

If your HUD is doing some wonky stuff when you change the Game's aspect ratio, ensure that your objects have the correct parent-child relationship. Your parent-child relationships should be as follows:

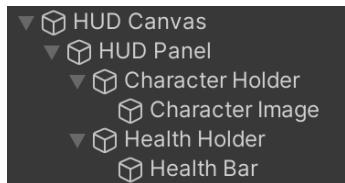


Figure 6.59: The Hierarchy's parent–child relationship

Also, check to ensure that the anchor and pivot points are set correctly.

Placing a 2D game background image

Placing a background image that scales with the screen is not too difficult as long as you use the appropriate Canvas properties. We will expand upon our HUD example and place a background image in the scene.

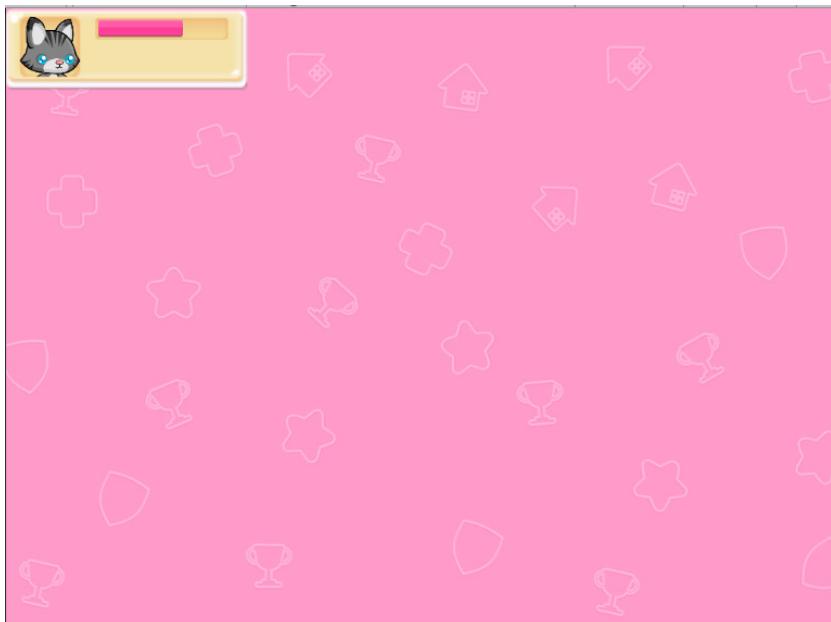


Figure 6.60: The result of the background image

We'll need to ensure that this background image doesn't just display behind other UI elements but also displays behind any game objects we may put in our scene.

To make a background image that displays behind all UI elements as well as all game elements, complete the following steps:

1. Create a new Canvas using **+ | UI | Canvas**. I like to use different Canvases to sort my different UI elements, but the need for a new Canvas stems from more than personal preference in this case. We need a new Canvas because we need a Canvas with a different Render Mode. This Canvas will use the **Screen Space-Camera** Render Mode.
2. In the **Canvas Inspector**, change the name to **Background Canvas**.
3. Change the **Render Mode** to **Screen Space-Camera** and drag the **Main Camera** into the **Render Camera** slot:

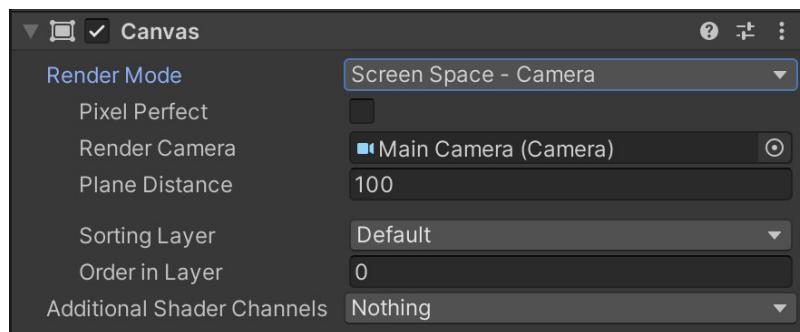


Figure 6.61: The Canvas component of the Background Canvas

4. To ensure that this Canvas appears behind all other UI elements and all the 2D sprites in the game, we will need to use Sorting Layers. In the top-right corner of the Unity Editor, you will see a dropdown menu labeled **Layers**. Select it and select **Edit Layers**:

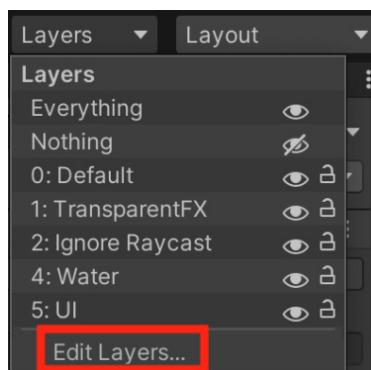


Figure 6.62: Editing Layers

5. Expand **Sorting Layers** by selecting the arrow. Add a new **Sorting Layer** by selecting the plus sign. Name this new layer **Background**.

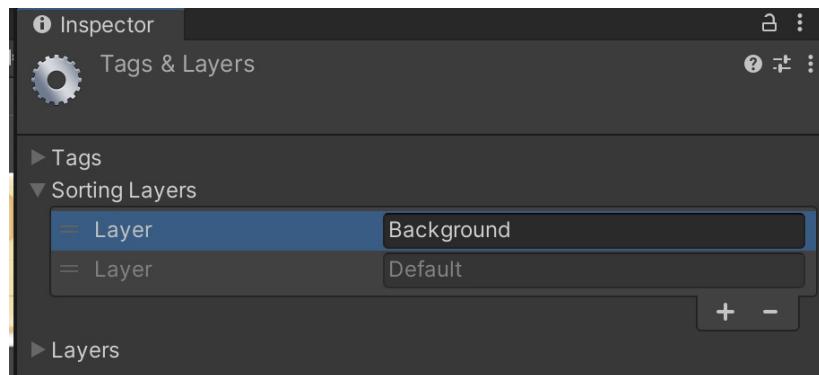


Figure 6.63: Adding the Background Sorting Layer

Sorting layers work so that whichever is on the top of this list will render the furthest back in the scene. So, if you wanted to add a foreground layer, you'd add it below **Default**. **Default** is the layer that all new sprites will automatically be added to, so unless you create new layers, the **Background** layer will be behind any new sprite you create. If you do create new layers, ensure that the **Background** layers stay on the top of this list.

6. Reselect **Background Canvas** and now change the **Sorting Layer** to **Background**:

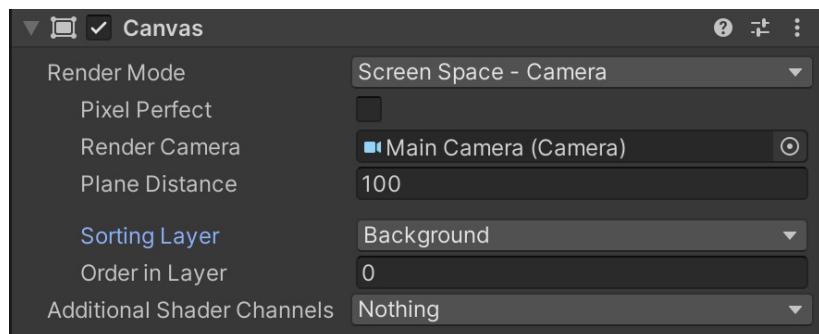


Figure 6.64: Setting the Background Sorting Layer

7. Now, all we need to do is add the background image. Right-click on the **Background Canvas** and select **UI | Image**. Rename the new image to **Background Image**.
8. Place the **pinkBackground** sprite in the **Source Image** slot. This time, we will not be selecting **Preserve Aspect** because we want the image to be able to squash and stretch as the game screen resizes and always fills the scene.

9. Set the **Rect Transform** properties as shown and ensure that you hold *Shift + Alt* when selecting the **Anchor Presets**:

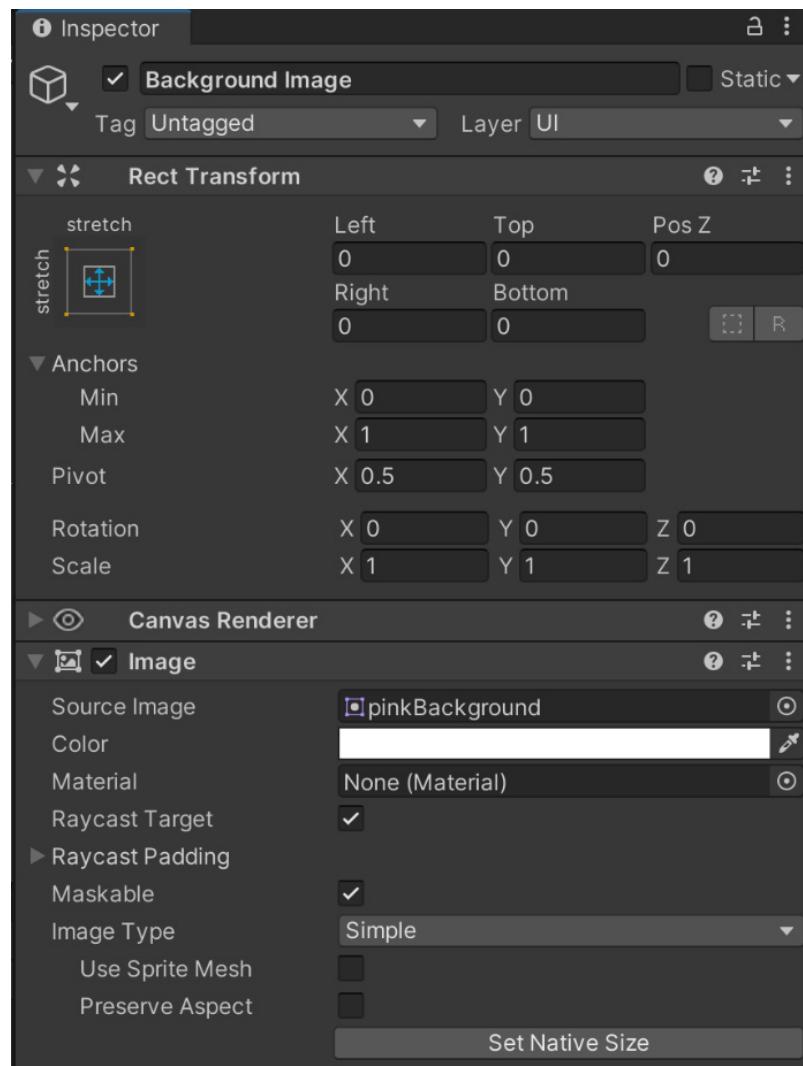


Figure 6.65: The Background Image settings

Because the Background Canvas is set to **Screen Space – Camera**, you may have to change your view so that you can see it. It will be where the Main Camera view is.

That's it! Try changing the Game view's aspect ratio around and resizing the screen in **Free Aspect** mode so that you can see the background image always filling the screen. Also, try adding some non-UI 2D sprites to the scene and see how they render on top of the background.

One thing that is not ideal about this example is that the background image is being allowed to change its aspect ratio. You'll see that the image looks pretty bad at some aspect ratios because of this. This background image will not be a good choice for a game that would be released on multiple aspect ratios. I chose this image for two reasons:

- You can see how it's important to pick an image that doesn't depend so highly on aspect ratio.
- It was free!

I highly recommend that if you use this method to create a background image, you use one with a pattern that doesn't so obviously display distortion.

Setting up a basic pop-up menu

The last example we will cover in this chapter will utilize the **Canvas Group** component. We won't be able to really see this component in action until we start programming in *Chapter 8*, but we can lay the groundwork now. We'll also get a bit more practice with laying out UI with this example.



Figure 6.66: The pop-up Panel we will lay out

To create the pop-up menu shown in the preceding image, complete the following steps:

1. Create a new Canvas using + | UI | **Canvas**.
2. In the **Canvas Inspector**, change the name to **Popup Canvas**.
3. I want to use the same properties for the **Canvas Scaler** on this Canvas that I used for **HUD Canvas**. Instead of setting up all that again, I'll use a shortcut and copy the **Canvas Scaler** from **HUD Canvas**. To do so, select the three dots (the "kabob" menu) in the right-hand corner of the **Canvas Scaler** component on **HUD Canvas** and select **Copy Component**.

4. Now select the “kabob” menu in the right-hand corner of the **Canvas Scaler** component on **Popup Canvas** and select **Paste Component Values**.
5. We will add a Panel that will hold all the items, similarly to the way we did with the HUD. This will ensure that everything stays together as it should. Right-click on **Popup Canvas** and select **UI | Panel**. Rename the new Panel **Pause Panel**.
6. Place the **uiElement_32** image in the **Source Image** slot, give it a full alpha value, and select **Preserve Aspect**.
7. Set the **Rect Transform** properties as shown in the following screenshot, and ensure that you hold *Shift + Alt* when selecting the **Anchor Preset**:

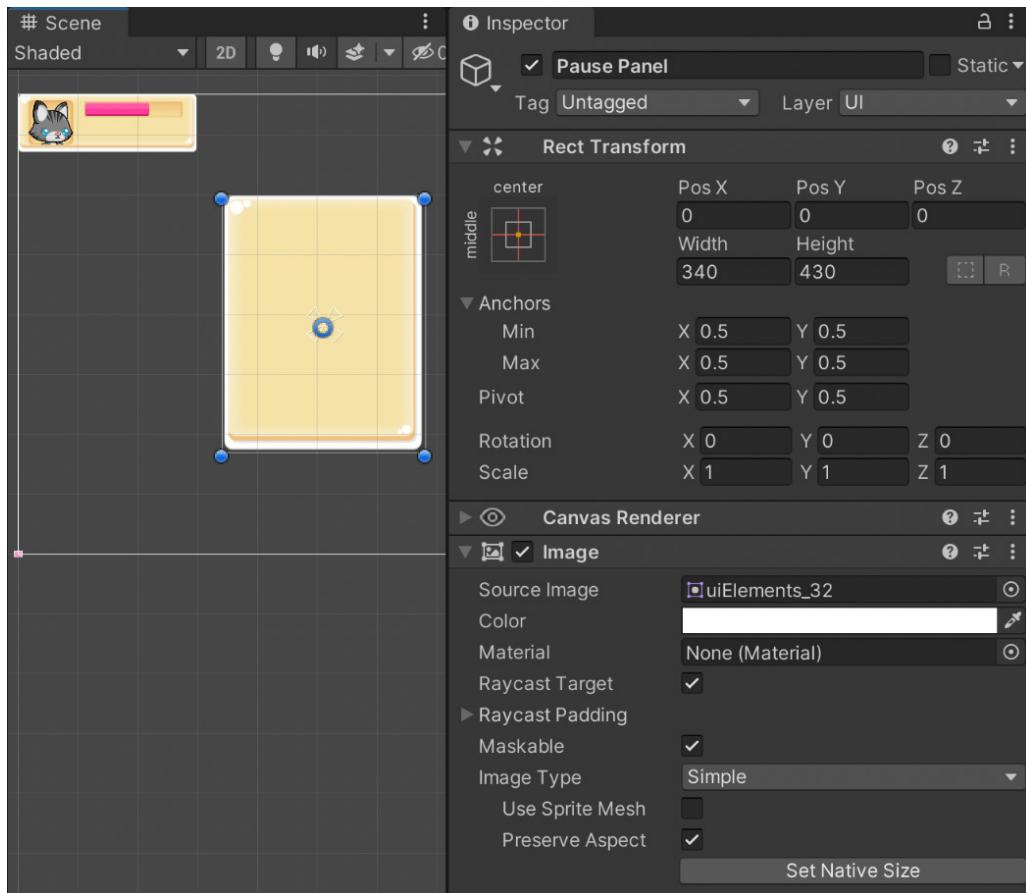


Figure 6.67: The properties of Pause Panel

8. Now, let's give the Panel a nice banner at the top. Right-click on **Popup Panel** and select **UI | Panel**. Rename the new Panel **Pause Banner**.

9. Place the uiElement_27 image in the **Source Image** slot and select **Preserve Aspect**.
10. Set the **Rect Transform** properties as shown in the following screenshot, and ensure that you hold *Shift + Alt* when selecting the **Anchor Presets**:

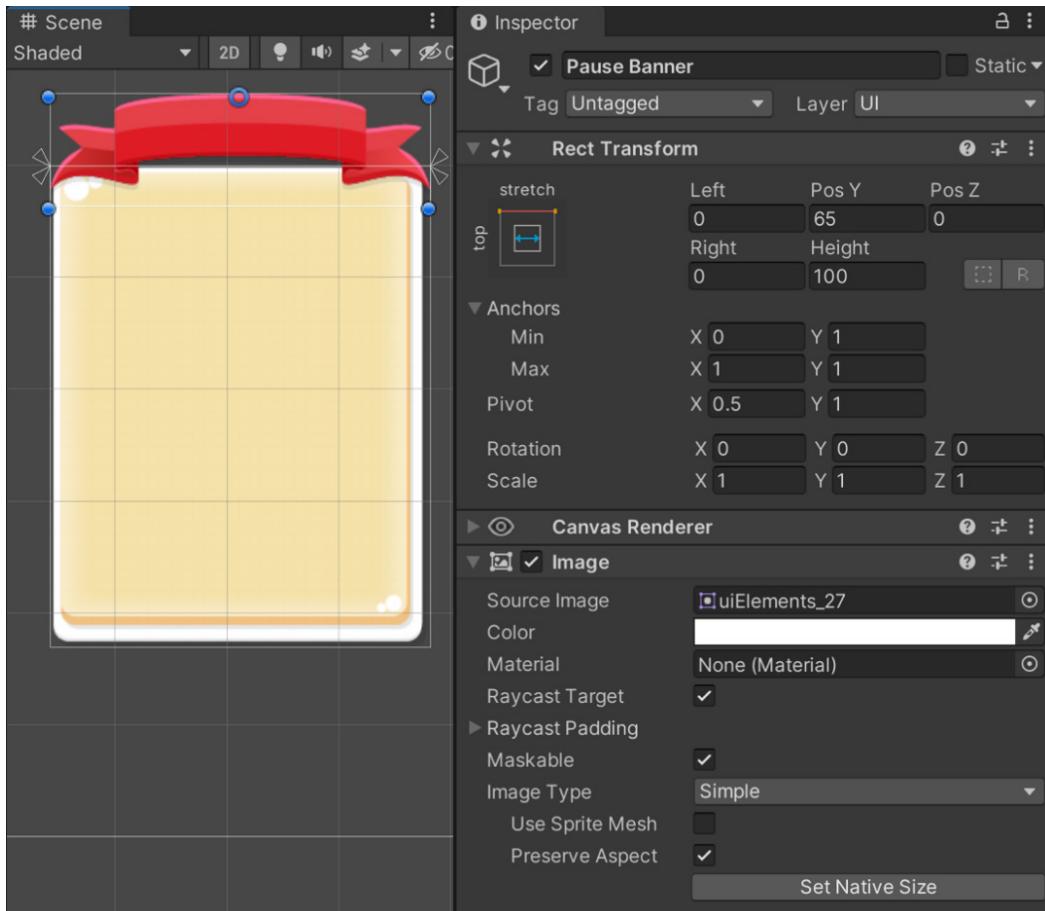


Figure 6.68: The properties of Pause Banner

We'll add the text to this banner in a later chapter.

11. The main point of this example was to demonstrate the use of the **Canvas Group** component (and to give you a little more layout practice). You can add a **Canvas Group** to any UI object. The **Canvas Group** will then be applied to all the children of the object to which you applied it. We will eventually put more pop-up Panels on this Canvas, and we want to be able to control each of them separately, so I will put the **Canvas Group** on the **Pause Panel**.

Select **Pause Panel**, and then select **Add Component | Layout | Canvas Group** (you can also just search for **Canvas Group**).

That's it for now! Change the values of **Alpha** in the **Inspector** of the **Pause Panel Canvas Group** component, and you will see that as you change it, both the **Pause Panel** and the **Pause Banner** alpha values change. This is great for pop-up menus you want to hide and show without having to program each item individually. Once we spend more time with **Pause Panel**, it will have a lot more items on it, and we will be happy that we don't have to program each piece individually.

Summary

Wow! This chapter was intense! There was a lot to cover, as this chapter set the groundwork that will be used throughout the rest of this book. We discussed the concept of a Canvas and how to correctly position it within your scene. Additionally, we discussed the basic UI Panel to allow us to explore the concept of positioning UI elements within a scene. Correctly setting up Canvases and their scalars is an important first step in developing UI.

The next chapter will cover how to create different automatic layouts that will let us line up our UI in grids.

7

Exploring Automatic Layouts

Now that we have the basics of manually positioning, scaling, and aligning UI elements with the Rect Transform and anchors, we can explore how to use automatic layouts. Automatic layouts allow you to group your UI elements so that they will position automatically relative to each other.

There are quite a few scenarios in which you will want Unity to automatically control the layout of your UI objects. If you are generating UI items via code and the number of items may change, but you still want them to line up, scale, and position properly, you can use automatic layouts. Also, if you want perfectly spaced UI objects, automatic layouts will help you create this perfect spacing without having to do any position calculating yourself. These automatic layouts work well for things like inventory systems aligned in a grid or list.

In this chapter, we will discuss the following topics:

- Using Layout Group components to automatically space, position, and align a group of UI objects
- Using the Layout Element component, the Content Size Fitter component, and the Aspect Ratio Fitter component to resize UI elements
- How to set up a horizontal HUD selection menu
- How to set up a grid inventory

Note

All the examples shown in this section can be found within the Unity package named Chapter 07.unitypackage, within the code bundle. Each example image has a caption stating the example number within the scene. In the scene, each example is on its own Canvas, and some of the Canvases are deactivated. To view an example on a deactivated Canvas, simply select the checkbox next to the Canvas' name in the **Inspector**.

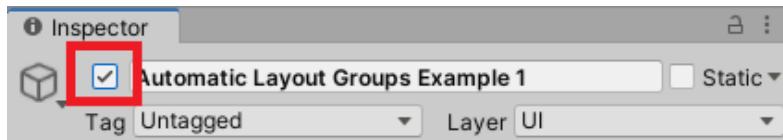


Figure 7.1: The checkbox to enable or disable a Canvas example

Let's explore the different types of Automatic Layout Groups.

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2007>

Types of automatic layout groups

When a UI object has an automatic layout group component attached to it, all of its children will be aligned, resized, and positioned based on the parameters of the layout component. There are three automatic layout group options: **Horizontal Layout Group**, **Vertical Layout Group**, and **Grid Layout Group**.

The following screenshot shows three Panels (represented by gray rectangles), each with six UI Image children (represented by the black rectangles); the first Panel has a Horizontal Layout Group component, the second Panel has a Vertical Layout Group component, and the third Panel has a Grid Layout Group component:

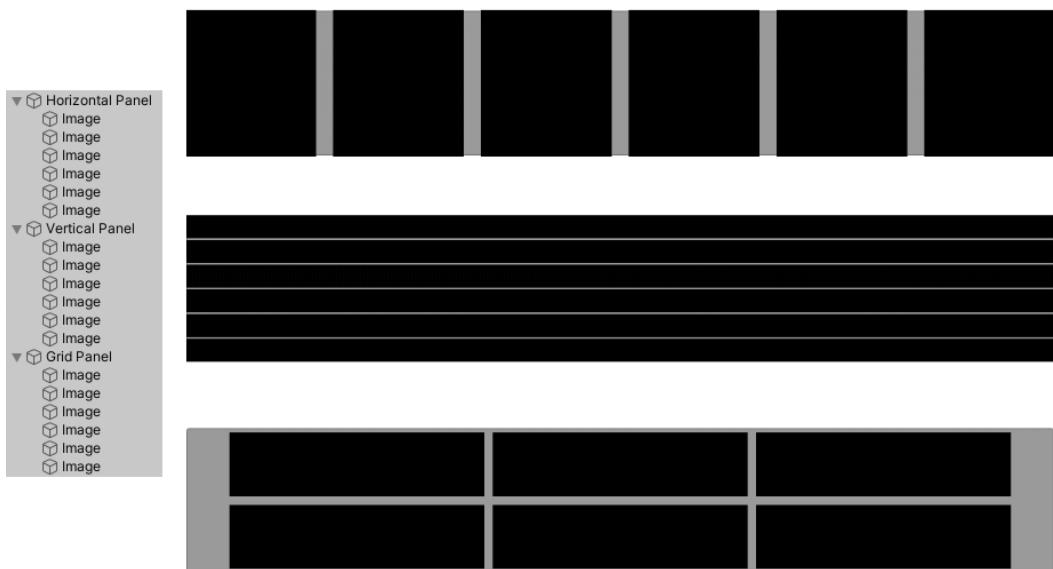


Figure 7.2: Automatic Layout Groups Example 1 in the Chapter7 scene

From the preceding screenshot, you can see clearly what the three types of automatic layout groups accomplish. You can use any combination of the three to create nested, perfectly spaced layouts, as follows:

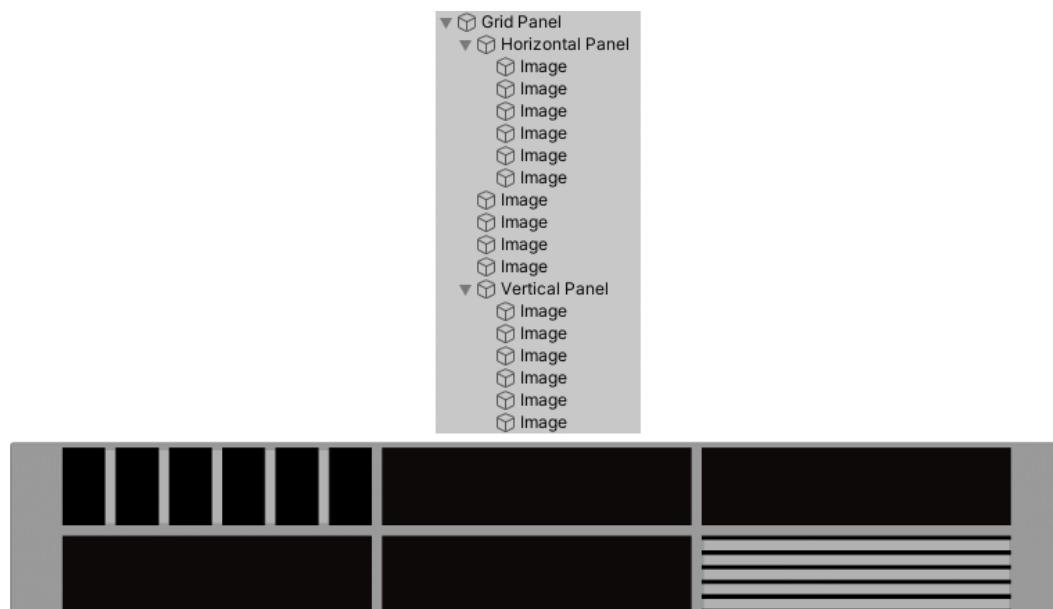


Figure 7.3: Automatic Layout Groups Example 2 in the Chapter7 scene

Let's look at each of these layout groups individually and explore their various properties.

Horizontal Layout Group

All the children of a UI object with a **Horizontal Layout Group** component will be automatically placed side by side. If you allow the Horizontal Layout Group to resize the children, they will be positioned and scaled so that they are fully within the bounds of the parent object's Rect Transform. Padding properties can be adjusted, however, if you'd like them to go outside the bounds of the parent's Rect Transform.

The order in which the children appear in the **Hierarchy** determines the order in which they will be laid out by the Horizontal Layout Group. The children will be laid out from left to right. The topmost child in the Hierarchy will be placed in the leftmost position, and the bottommost child in the Hierarchy will be placed in the rightmost position:

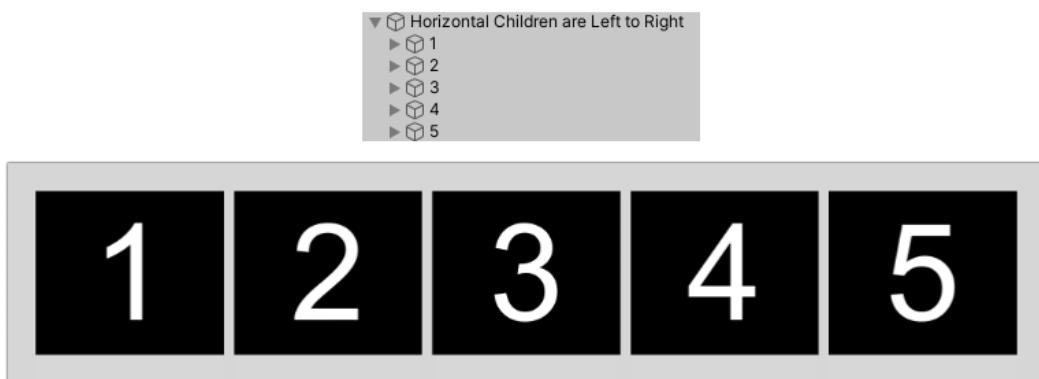


Figure 7.4: Horizontal Layout Groups Example 1 in the Chapter7 scene

To add a Horizontal Layout Group component to a UI object, select **Add Component | Layout | Horizontal Layout Group** from within the object's **Inspector**. If you click on the arrow next to the **Padding** property, you should see the following:

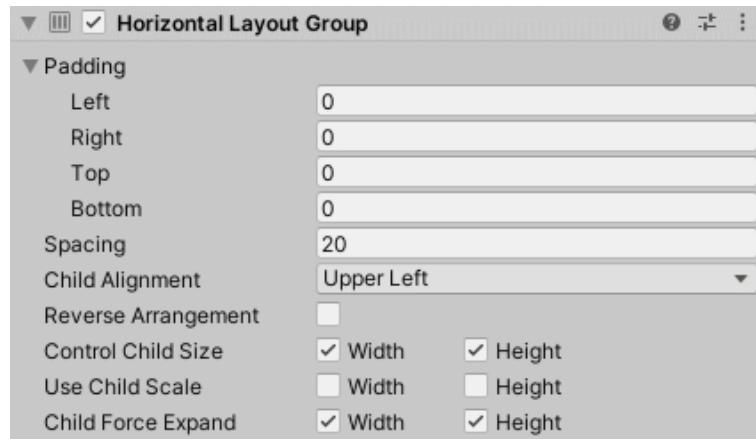


Figure 7.5: The Horizontal Layout Group component

Let's explore each of the properties of the Horizontal Layout Group component further.

Padding

The **Padding** property represents the padding around the edges of the parent object's Rect Transform. Positive numbers will move the child objects inward, and negative numbers will move the child objects outward.

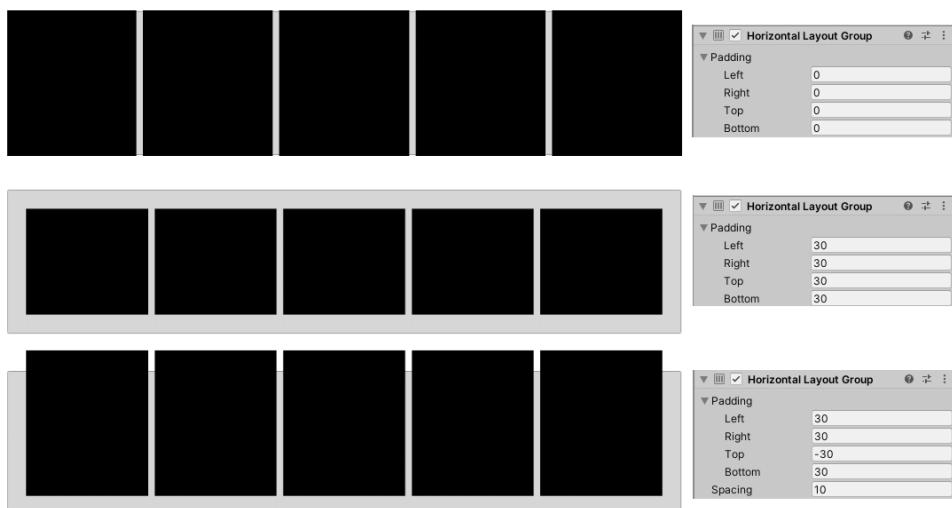


Figure 7.6: Horizontal Layout Groups Example 2 in the Chapter7 scene

As an example, the previous screenshot shows three Panels with various padding values applied. The first Panel has no padding, the second Panel has positive padding on all four sides, and the third Panel has positive padding on the left, right, and bottom but negative padding on the top.

Spacing

The **Spacing** property determines the horizontal spacing between the child objects. This may be overridden if you use the **Child Force Expand** property without the **Control Child Size** property, and the children may have larger spacing.

Child Alignment

The **Child Alignment** property determines where the group of children will be aligned. There are nine options for this property, as shown here:

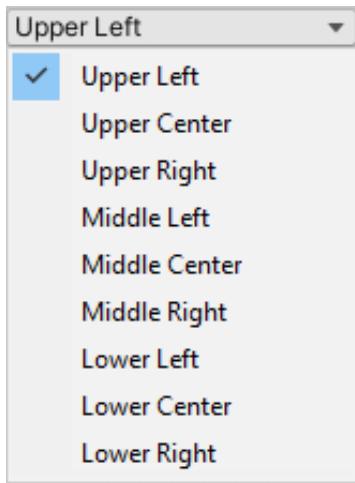


Figure 7.7: The Child Alignment options of the Horizontal Layout Group

As an example, the following diagram shows three overlapping Panels that fill the screen. The Rect Transform area for these parent Panels is represented by the selected Rect Transform. The first Panel has an **Upper Left** Child Alignment. Its children are represented by the white squares. The second Panel has a **Middle Center** Child Alignment, and its children are represented by gray squares. The third Panel has a **Lower Right** Child Alignment, and its children are represented by black squares:

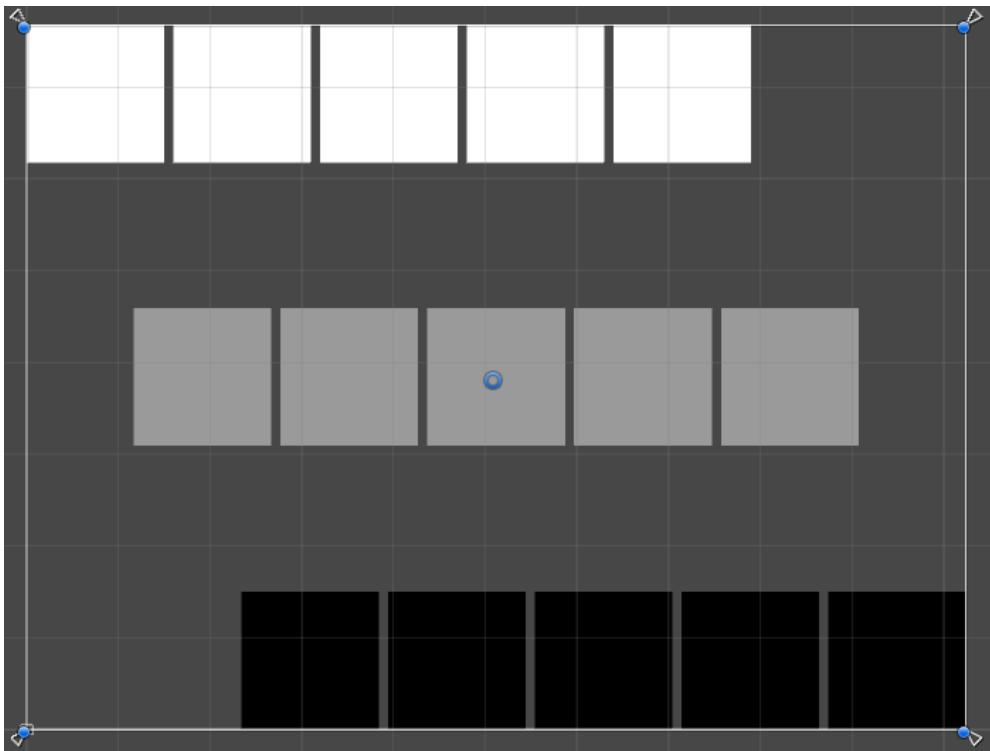


Figure 7.8: Horizontal Layout Group Example 3 in the Chapter7 scene

It is important to note that the **Child Alignment** property only shows an effect if the children (along with spacing) don't completely fill in the Rect Transform, as shown in the preceding diagram.

Reverse Arrangement

The **Reverse Arrangement** property is a toggle. Selecting the toggle will cause the elements to arrange in the reverse order than they appear in the Hierarchy.

Control Child Size

The **Control Child Size** options allow the automatic layout to override the current **Width** or **Height** of the child objects. If you select these checkboxes without selecting the corresponding **Child Force Expand** checkboxes, your child objects will no longer be visible (unless the children have Layout Element components with **Preferred Width** specified).

If you do not set this property, it is possible that the children will draw outside of the parent's Rect Transform – that is, if too many children exist.

Note

This property changes the width and height property of the child objects' Rect Transforms. So, if you select and then deselect it, the children will not go back to their original sizes. You will have to either use **Edit | Undo** (*Ctrl + Z*) or manually reset the size of the children via their Rect Transform components.

Since this property depends on the **Child Force Expand** property, examples of the **Control Child Size** property are presented in the next section.

Child Force Expand

The **Child Force Expand** property will cause the children to fill the available space. If the corresponding **Control Child Size** is not selected, this property will shift the children so that they and their spacing fill the space. This may override the **Spacing** property. If the corresponding **Control Child Size** is selected, it will stretch the children in the selected direction so that they and their spacing completely fill the space. This will maintain the **Spacing** property.

In the following screenshot, all three Panels have a Horizontal Layout Group component with a **Middle Left Child Alignment** and different combinations of **Child Control Size** and **Child Force Expand** selected. The top Panel has only **Child Force Expand Width** selected, the middle Panel has **Control Child Size Width** and **Child Force Expand Width** selected, and the last Panel has both **Control Child Size** properties selected and both **Child Force Expand** properties selected:



Figure 7.9: Horizontal Layout Group Example 4 in the Chapter7 scene

Next, let's look at the **Use Child Scale** properties.

Use Child Scale

The **Use Child Scale** properties are only available in recent versions of Unity. Checking this property will tell the Layout Group whether it should consider the scale of the children when automating the layout.

Vertical Layout Group

The **Vertical Layout Group** component works very similarly to the Horizontal Layout Group and has all the same properties, except children of a UI object with a Vertical Layout Group component will be automatically placed on top of each other, rather than side by side.

As with the Horizontal Layout Group, the order in which the children appear in the Hierarchy determines the order in which they will be laid out by the Vertical Layout Group. The children will be laid out from top to bottom in the same order in which they appear in the Hierarchy:

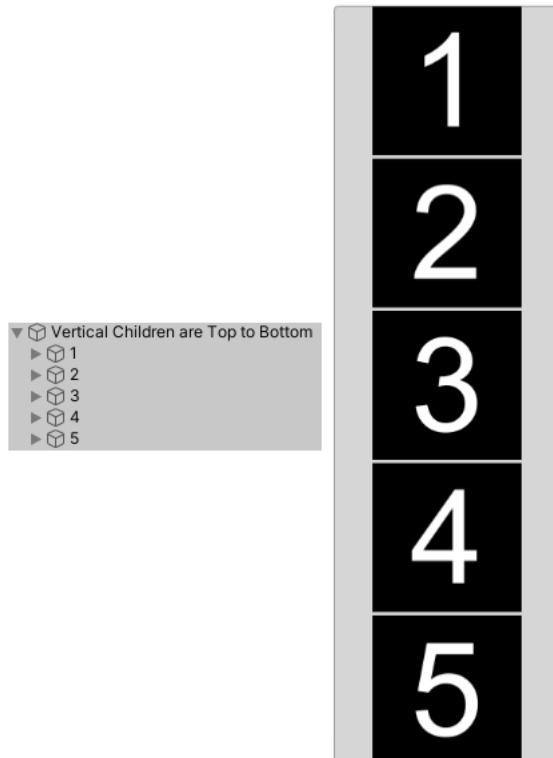


Figure 7.10: Vertical Layout Group Example in the Chapter7 scene

To add a Vertical Layout Group component to a UI object, select **Add Component | Layout | Vertical Layout Group** from within the object's Inspector. If you click on the arrow next to the **Padding** property, you should see the following:

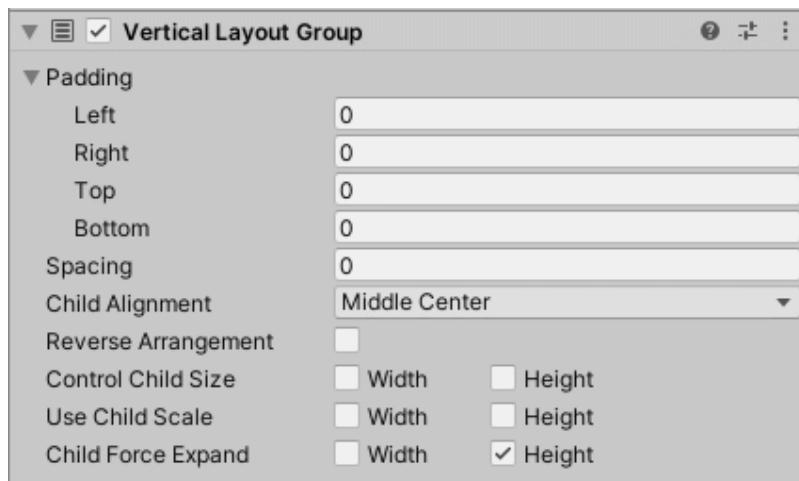


Figure 7.11: The Vertical Layout Group component

Since the properties of the Vertical Layout Group component are identical to those of the Horizontal Layout Group, we won't explore each of the properties further. For an explanation of each of the properties, refer to the *Horizontal Layout Group* section.

Grid Layout Group

The **Grid Layout Group** component allows you to organize child objects in columns and rows in (you guessed it) a grid layout. It works similarly to Horizontal and Vertical Layout Groups but has a few more properties that can be manipulated.

To add a Grid Layout Group component to a UI object, select **Add Component | Layout | Grid Layout Group** from within the object's Inspector. If you click on the arrow next to the **Padding** property, you should see the following:

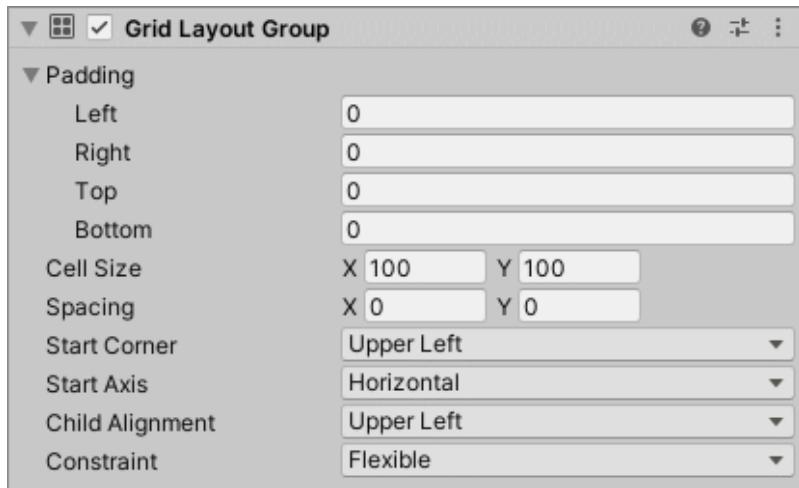


Figure 7.12: The Grid Layout Group component

A few of the properties of the Grid Layout Group are the same as the other two Layout Groups, but let's look more closely at the properties unique to the Grid Layout Group component.

Cell Size

Unlike the Horizontal and Vertical Layout Groups, which determine the size of the children either by their Rect Transform component or by scaling them to fit inside the parent's Rect Transform, the Grid Layout Group requires you to specify the width and height of the child objects. You accomplish this by setting the X and Y properties of the **Cell Size** property. This will automatically apply the specified X and Y sizes to each of the children's **Width** and **Height** properties of their Rect Transform, respectively.

Due to the **Cell Size** property and the lack of a **Control Child** size property, the children are not guaranteed to fit within the parent's Rect Transform. If too many children exist, it is possible they will be drawn outside the parent's Rect Transform. So, if you have a grid filling up dynamically that can change throughout the gameplay and want the grid to always fit within a specific area, you will have to prepare for that overflow scenario.

The Grid Layout Group allows you to specify both an **X Spacing** and **Y Spacing**. The **X Spacing** is the horizontal spacing, and the **Y Spacing** is the vertical spacing. These values will not be overridden by further property choices as they can be with the Horizontal and Vertical Layout Groups.

Start Corner and Start Axis

The **Start Corner** property determines where the very first child in the Hierarchy will be placed. There are four choices for the **Start Corner** property, as shown here:

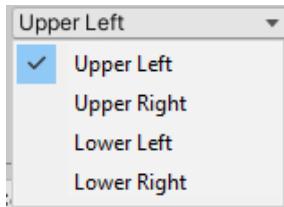


Figure 7.13: The Start Corner options of a Grid Layout Group component

The **Start Axis** property determines where all the other children will be placed relative to the first child. There are two options, as follows:

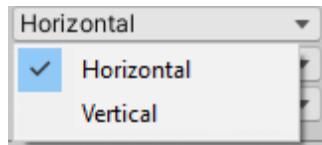


Figure 7.14: The Start Axis options of a Grid Layout Group component

A **Start Axis** property set to **Horizontal** means that the children will be laid out, starting with the first child, in a horizontal fashion. If the **Start Corner** is assigned to one of the **Left** options, the children will be placed from left to right. If the **Start Corner** is assigned to one of the **Right** options, the children will be placed from right to left. Once the new row is filled, it will continue to the next row and will restart on the same side as the **Start Corner**. If the **Start Corner** is one of the **Upper** options, the rows will continue downward. If the **Start Corner** is one of the **Lower** options, the rows will continue upward.

The following screenshot demonstrates the flow of the children, with a **Horizontal Start Axis** based on the different **Start Corner** options:



Figure 7.15: Grid Layout Group Example 1 in the Chapter7 scene

A **Start Axis** property set to **Vertical** means that the children will be laid out starting with the first child, and then in a vertical fashion. Whether the children will be placed from top to bottom or from bottom to top is determined in the same way as it is when this property is set to **Horizontal**, based on the position of the **Start Corner**. Then, when a column is filled, the children will be placed from left to right or from right to left, based on the position of the **Start Corner**.

The following screenshot demonstrates the flow of the children, with a **Vertical Start Axis** based on the different **Start Corner** options:



Figure 7.16: Grid Layout Group Example 2 in the Chapter7 scene

As you can see, the **Start Corner** and **Start Axis** options can greatly change the order in which your child objects are displayed.

Constraint

The **Constraint** property allows you to specify the number of rows or columns the grid will have. There are three options, as shown here:

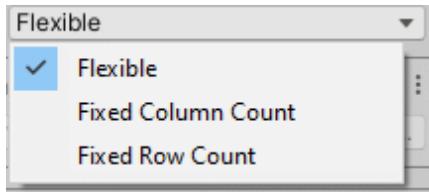


Figure 7.17: The Constraint options of a Grid Layout Group component

The **Fixed Column Count** and **Fixed Row Count** properties allow you to specify a number of columns or rows, respectively. If you select either of these options, a new property, **Constraint Count**, will become available. You then specify how many columns or rows you want. When you select **Fixed Column Count**, the number of rows will be variable. When you select **Fixed Row Count**, the number of columns will be variable.

The **Flexible** option automatically calculates the number of rows and columns for you, based on the **Cell Size** and the **Start Axis** options chosen. It will begin laying out the children in the defined pattern until there is no space left on the chosen axis. It will then continue. Whichever axis is specified in **Start Axis** will have a fixed amount of children, and the other axis will be variable. So, for example, if **Start Axis** is set to **Horizontal** and three children can fit horizontally within the defined space, there will be three columns, and the number of rows will be determined by how many total children there are.

Now that we've explored the three Automatic Layout Groups, let's look at a component that will let us change the way the children within these layout groups will be sized or positioned.

Layout Element

The **Layout Element** component allows us to specify a range of size values of an object if it is being sized with an automatic layout. If the parent object tries to size it outside of these preferences, the **Layout Element** will override any sizing information being sent from the parent object.

To add a **Layout Element** component to a UI object, select **Add Component | Layout | Layout Element** from within the object's **Inspector**. The **Layout Element** has the following properties:



Figure 7.18: The Layout Element component

To use these properties, you first select their checkboxes to enable them; boxes will become available so that you can enter your desired values:

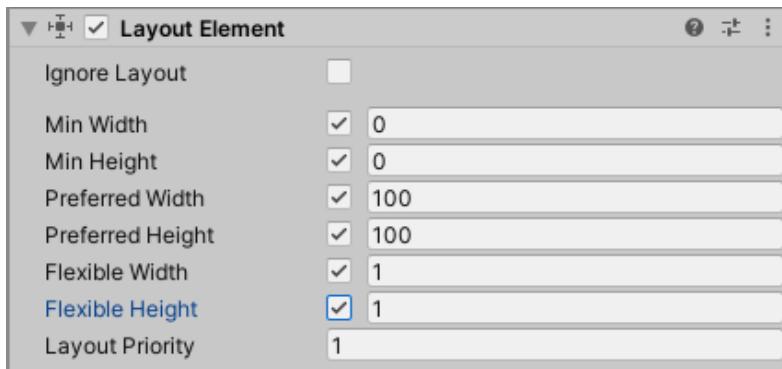


Figure 7.19: Setting the properties of the Layout Element component

Let's review how the individual properties of the Layout Element component will affect the elements to which they are added.

Ignore Layout

The **Ignore Layout** property can be used to make child objects ignore any automatic layout component of its parent object. A child with this property selected can be moved and resized freely, and all other children will be laid out without regard for the ignored child.

In the following example, the Panel has a **Horizontal Layout Group** component and five child objects. The first child, labeled with a 1, has a **Layout Element** component with the **Ignore Layout** property selected:

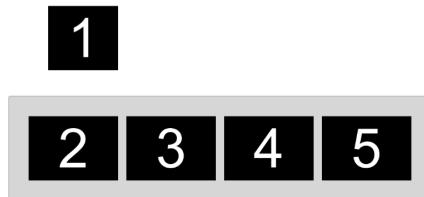


Figure 7.20: Layout Element Example 1 in the Chapter7 scene

You can see that since the **Ignore Layout** property is selected for the first child; it can be moved around outside of the parent Panel, and it was ignored when the position and scale of the other children were determined. It also maintained its original Rect Transform scale.

If the **Ignore Layout** property is deselected, the first child will be added to the Horizontal Layout Group with the other children.

The Width and Height properties

The **Layout Element** component has three sets of properties that can be used to specify the way you want an object to resize. These properties will override the size being assigned to the child by the parent object if the assigned size is outside of the provided values.

Note

It is important to note that these properties will not override the **Cell Size** settings of the Grid Layout Group component. *They will have no effect on a child within a Grid Layout Group.*

Min Width and Height

The **Min Width** and **Min Height** properties are the minimum width and height a child object can achieve. If the parent object is scaled down, the child will scale down until it meets its **Min Width** or **Min Height**. Once it does so, it will no longer scale in that direction.

In the following diagram, the Panel has a **Horizontal Layout Group** component and five child objects. The first child, labeled with a 1, has a **Layout Element** component with the **Min Width** and **Min Height** properties set:



Figure 7.21: Layout Element Example 2 in the Chapter7 scene

You can see that the parent object's **Horizontal Layout Group** tried to scale all the children down with it as it scaled down itself. The other four children scaled, but since the first child had a **Min Width** and **Min Height** properties set, it refused to scale down any further.

Preferred Width and Preferred Height

The **Preferred Width** and **Preferred Height** properties are a little confusing because they perform differently, depending on the settings you have for the parent's layout group. There is no official **Max Width** and **Max Height** setting, despite there being a **Min Width** and **Min Height** setting. The **Preferred Width** and **Preferred Height** properties, however, can be used to specify the maximum size the child object will achieve, but only if the correct settings on the parent's layout group are selected.

The following diagram contains three Panels with **Vertical Layout Group** components and various settings. Their children also have various settings for **Preferred Height** within a **Layout Element** component:

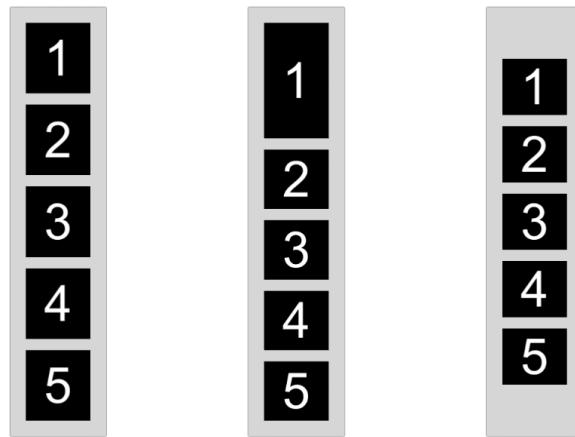


Figure 7.22: Layout Element Example 3 in the Chapter7 scene

The first parent Panel has a **Vertical Layout Group** component with **Control Child Size Width** and **Height** selected, as well as **Child Force Expand's Width** and **Height**. None of its children have a **Preferred Width** or **Preferred Height** setting within a **Layout Element** component. The first Panel will act as the default for reference when comparing the others.

The second parent Panel has the same properties as the first – a **Vertical Layout Group** component with **Control Child Size's Width** and **Height** selected, as well as **Child Force Expand Width** and **Height**. However, its first child has a **Preferred Height** setting of 100 within a **Layout Element** component.

You can see that because the second parent Panel has **Child Force Expand's Height** selected, the **Preferred Height** of 100 in the **Layout Element** component of the first child causes the first child to be exactly 100 units taller than the other four children. So, when the **Child Force Expand** property is selected on the parent, the child with the **Preferred Height** will not use **Preferred Height** as its maximum possible height; it will add that value to the height assigned by the parent's layout group component.

The third Panel has a **Vertical Layout Group** component with **Control Child Size Width** and **Height** selected, as well as **Child Force Expand's Width**. It does not have **Child Force Expand's Height** selected as the other two do. All of its children have a **Preferred Height** setting of 100 within a **Layout Element** component.

If you compare the children in the third Panel to the children in the first Panel (the default), you can see that the children are shorter. This is because their **Preferred Height** is set to a smaller number than the height that the Vertical Layout Group component attempts to assign to them. So, when the **Child Force Expand's Height** property is deselected, the children will use their **Preferred Height** setting in the expected way – making it the maximum size the children should attain.

Therefore, if you want the **Preferred Width** or **Preferred Height** settings to work as a maximum attainable width or height, you will need to deselect the corresponding **Child Force Expand** property on the parent object.

Flexible Width and Flexible Height

The **Flexible Width** and **Flexible Height** properties represent a percentage, where the percentage is the size of the child relative to the other children. Since these values are percentages, a value of 0 would represent 0% and a value of 1 would represent 100%.

As with **Preferred Width** and **Preferred Height**, this setting doesn't work as expected unless the **Child Force Expand** property is deselected. In the following example, the two Panels and children have nearly identical settings. The only difference between the two is that the top parent Panel has **Child Force Expand's Width** property selected and the bottom parent Panel does not. So, you can see that the values set for **Flexible Width** of the children are ignored if **Child Force Expand's Width** is selected on the parent:

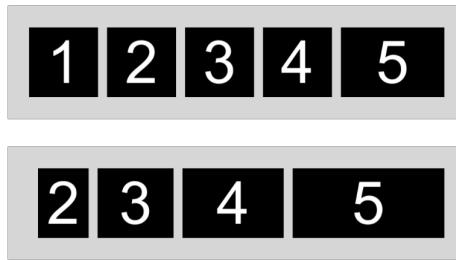


Figure 7.23: Layout Element Example 4 in the Chapter7 scene

The children in the second row of the preceding figure have the following **Flexible Width** settings from left to right: 0, 0 . 5, 0 . 75, 1, and 1 . 5. You can see that the children have scaled relative to each other based on the percentages. The first child is not visible because it has a **Flexible Width** of 0.

The Layout Element component essentially lets us override the automatic size and position of an element. Now, let's review some components that will allow us to automatically size UI elements.

Fitters

There are two fitter layout components. These components make the Rect Transform of the object on which they are attached fit within a specified area.

Content Size Fitter

The **Content Size Fitter** component allows you to force the size of the parent to fit around the size of its children. This fitting can be based on the minimum or preferred size of the children.

To add a **Content Size Fitter** component to a UI object, select **Add Component | Layout | Content Size Fitter** from within the object's **Inspector**. The **Content Size Fitter** component has the following properties:

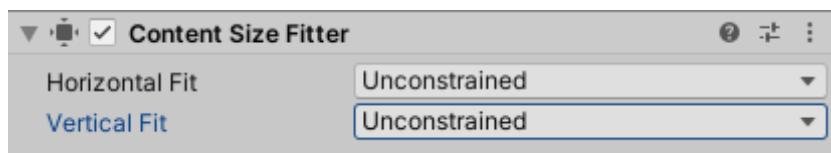


Figure 7.24: The Content Size Fitter component

You can choose the following properties for the **Horizontal Fit** and the **Vertical Fit**:

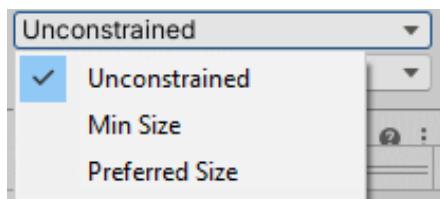


Figure 7.25: The possible fit options of the Content Size Fitter component

If the **Unconstrained** property is selected, **Content Size Fitter** will not adjust the size of the object along that axis.

If the **Min Size** property is selected, the **Content Size Fitter** will adjust the size of the object based on the minimum size of the children. This minimum size is determined by the **Min Width** and **Min Height** properties of the **Layout Element** component of the children.

The children do not have to have a **Layout Element** component if the parent has a **Grid Layout Group** component for this property to work. If this property is selected for an object with a **Grid Layout Group** component, the Rect Transform of the parent will hug the children based on the **Cell Size** and **Padding** properties, as shown in the following diagram:

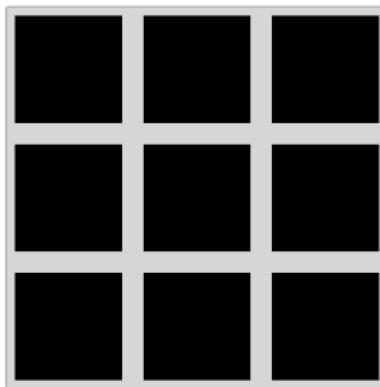


Figure 7.26: The Content Size Fitter Example in the Chapter7 scene

If the **Preferred Size** property is selected, the **Content Size Fitter** will adjust the size of the object based on the preferred size of the children. This preferred size is determined by the **Preferred Width** and **Preferred Height** properties of the **Layout Element** component of the children. If the object has a **Grid Layout Group** component, this setting will perform in the exact same way as **Min Size**.

Aspect Ratio Fitter

The **Aspect Ratio Fitter** component works similarly to the **Layout Element** component, as it allows you to override the size constraints being sent to it. It will force the UI object on which it is attached to resize based on an aspect ratio.

To add an **Aspect Ratio Fitter** component to a UI object, select **Add Component | Layout | Aspect Ratio Fitter (Script)** from within the object's **Inspector**. The **Aspect Ratio Fitter** component has the following properties:

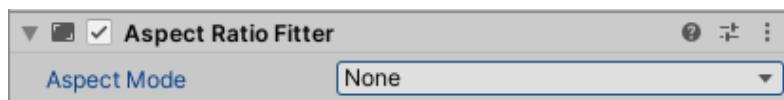


Figure 7.27: The Aspect Ratio Fitter component

Once you select an **Aspect Mode** option, the **Aspect Ratio** property will be editable. The **Aspect Ratio** property defines the aspect ratio that the Rect Transform will maintain. For example, if you want an aspect ratio of 4:3, you can simply enter 4/3 in the box, and it will convert it to the decimal value:



Figure 7.28: Entering fractions into an Aspect Ratio Fitter Component

You can choose the following properties for the **Aspect Mode** property:



Figure 7.29: The Aspect Mode options for the Aspect Ratio Fitter component

If the **None** property is selected, the **Aspect Ratio Fitter** will not adjust the size to fit within the **Aspect Ratio**.

If the **Width Controls Height** property is selected, the **Aspect Ratio Fitter** will adjust the size of the height based on the width of the object.

If the **Height Controls Width** property is selected, the **Aspect Ratio Fitter** will adjust the size of the width based on the height of the object.

If the **Fit In Parent** property is selected, the **Aspect Ratio Fitter** will adjust the size of the object to fit within its parent object but will maintain the **Aspect Ratio**. This will make the child object stay within the bounds of the parent.

If the **Envelope Parent** property is selected, the **Aspect Ratio Fitter** will adjust the size of the object to cover its parent object but will maintain the **Aspect Ratio**. This is similar to the **Fit In Parent** property, except that instead of staying within the bounds of the parent, it can go outside the bounds.

If you try to add an **Aspect Ratio Fitter** component to a child with a parent that has a layout group component, you'll see the following message on the child:

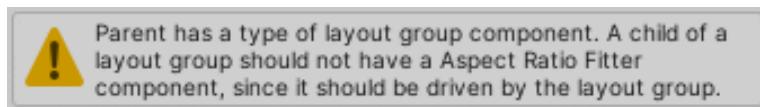


Figure 7.30: The Aspect Ratio Fitter warning message

While you can ignore this message and do it anyway, it doesn't work entirely as expected. The recommended workaround is to add the **Aspect Ratio Fitter** component to a child of the child within the group. For example, in the following diagram, a Panel was added as a child of the **Horizontal Layout Group**. Then, a child with an **Aspect Ratio Fitter** component was added to the Panel so that the child could have the 4:3 **Aspect Ratio**:

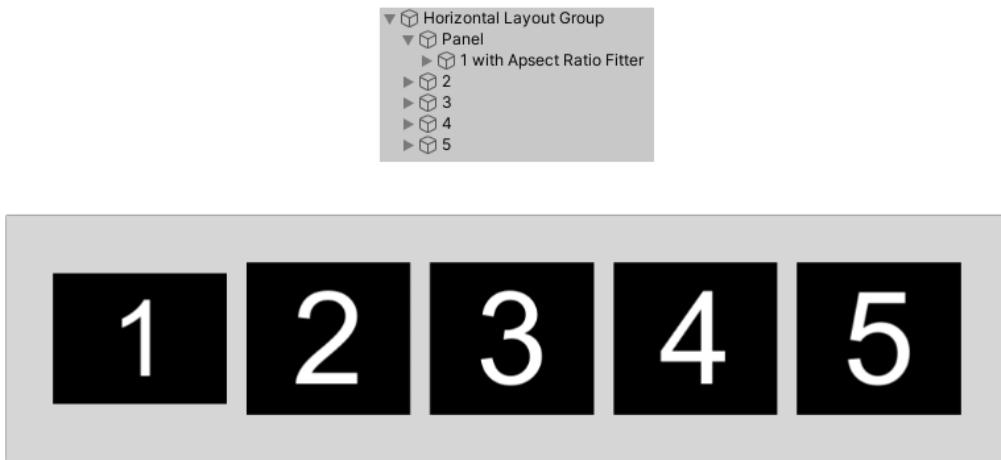


Figure 7.31: Aspect Ratio Fitter Example in the Chapter7 scene

Now that we've looked at all the properties of the various automatic layout components, let's look at some examples of how to use them!

Examples

We'll continue working on the scene created in *Chapter 6* and use the art assets imported for them.

Note

If you did not follow along with the examples in *Chapter 6* but would like to follow along with these, you can download the unity package named `Chapter 07 - Examples - Start.unitypackage` from the code bundle.

In addition to the art already added to our project, we'll be using art assets that I've modified from free art assets found at <https://opengameart.org/content/platformer-pickups-pack>.

The download from the previous link provides many individual images. I could have used those, but for performance reasons, it is best to use sprite sheets whenever possible. So, you can find the sprite sheet labeled `foodSpriteSheet.png` in the code bundle. To combine all the images into a sprite sheet, I used the program Texture Packer, which can be found at <https://www.codeandweb.com/texturepacker>.

Before you begin with the following examples, complete the following steps:

1. Import the `foodSpriteSheet.png` sprite sheet into your project's Asset/Sprites folder.
2. Change the **Sprite Mode** of `foodSpriteSheet.png` to **Multiple**. Use the **Sprite Editor** to automatically slice the sprite sheet.

3. Automatic slicing results in a blank image being created in the sprite sheet. Find the rectangle shown in the following screenshot within the **Sprite Editor**, and then select and delete it:



Figure 7.32: An empty sprite that needs removing

4. Once you apply your changes, you should have the following in your **Sprites** folder:



Figure 7.33: All sprites currently in the project

5. Duplicate your scene named **Chapter6** by pressing **Ctrl + D**, and name it **Chapter7**. Open the **Chapter7** scene and complete the following examples within that scene.

Now that you have the scene duplicated and the art imported, let's look at applying some automatic layouts.

Laying out a HUD selection menu

The first example we will cover in this chapter is a HUD selection menu in the lower-right corner of the screen that uses the **Horizontal Layout Group** component. When we are done, it will look like the following figure:



Figure 7.34: The HUD selection menu we will build in this example

To create the HUD group shown in the preceding screenshot, complete the following steps:

1. Currently, we have a Panel in the upper-left corner of the screen named **HUD Panel**. For clarity's sake, rename this Panel **Top Left Panel**.
2. We will create a new HUD Panel to hold our fruity inventory. We want to put our new HUD Panel in the **HUD Canvas**. Right-click on the Canvas named **HUD Canvas** in the **Hierarchy** and select **UI | Panel**.
3. Rename the new Panel **Bottom Right Panel**:

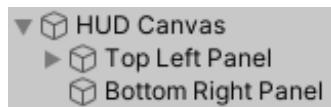


Figure 7.35: The Panels in the Hierarchy

4. Change the Rect Transform properties of the **Bottom Right Panel** so that the Panel is anchored in the lower-right corner, has a **Width** of 500, and has a **Height** of 100. Remember to hold down *Shift + Alt* when selecting the lower-right anchor preset:

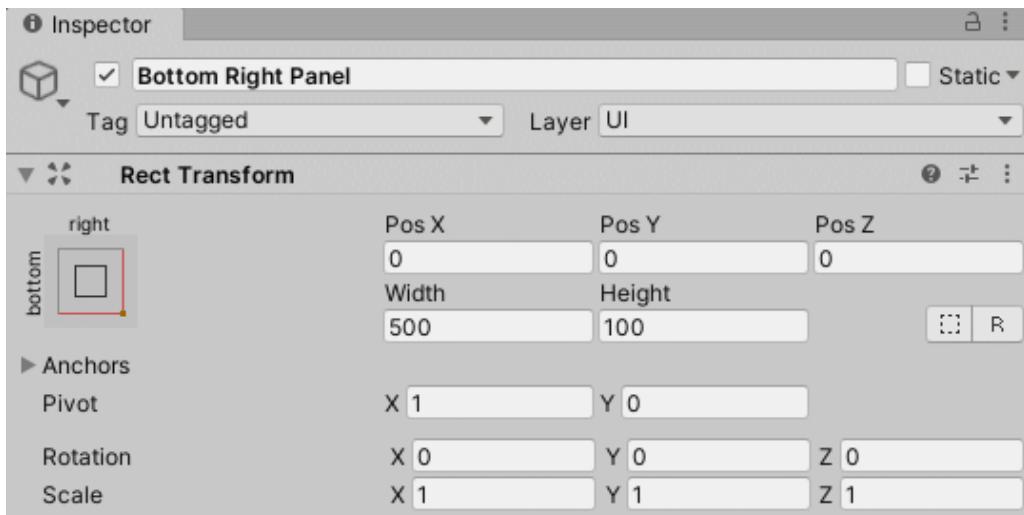


Figure 7.36: The Rect Transform of the Bottom Right Panel

You should see the following in your Game view:



Figure 7.37: The resulting Panel

- Now, we will replace the image with one of the `uiElements_1.png` sprites. Drag `uiElements_1` into the **Source Image** property of the **Image** component. Change the **Color** property so that it has full opacity:

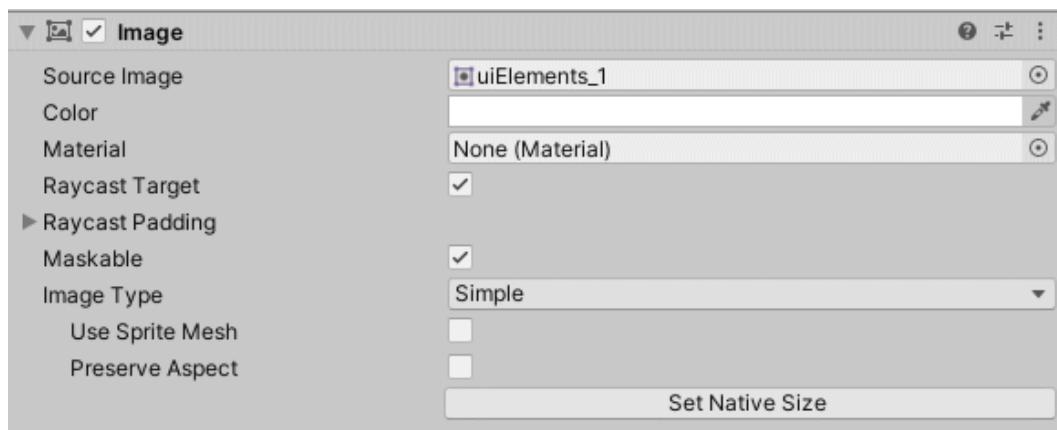


Figure 7.38: The Image component properties of the Panel

You should now see the following in your **Game view**:



Figure 7.39: The resulting Panel

6. To create the layout we want, we need to add a **Horizontal Layout Group** component to the **Bottom Right Panel**. Select **Add Component | Layout | Horizontal Layout Group**. We will adjust its properties momentarily. First, let's give this Panel some children so that we can see the effects of the properties take place.
7. Right-click on the **Bottom Right Panel** in the **Hierarchy** and select **UI | Image**. Rename this **Image Item Holder**. We won't be changing the **Rect Transform** component of this **Image** because we will allow the **Horizontal Layout Group** of its parent to control its size, position, and anchor.
8. This **Image** will be the background holder for the item. So, drag **uiElement_6** into its **Image** component's **Source Image**.
9. Now, let's add the **Image** for the fruit. Right-click on **Item Holder** in the **Hierarchy** and select **UI | Image** to give it a child **Image**. Rename this **Image Food**:



Figure 7.40: The Hierarchy of UI elements

10. To ensure that we aren't just looking at a white block, let's replace the **Source Image** with one of the food sprites from **foodSpriteSheet.png**. I've used **foodSpriteSheet_18**, which is a full orange:

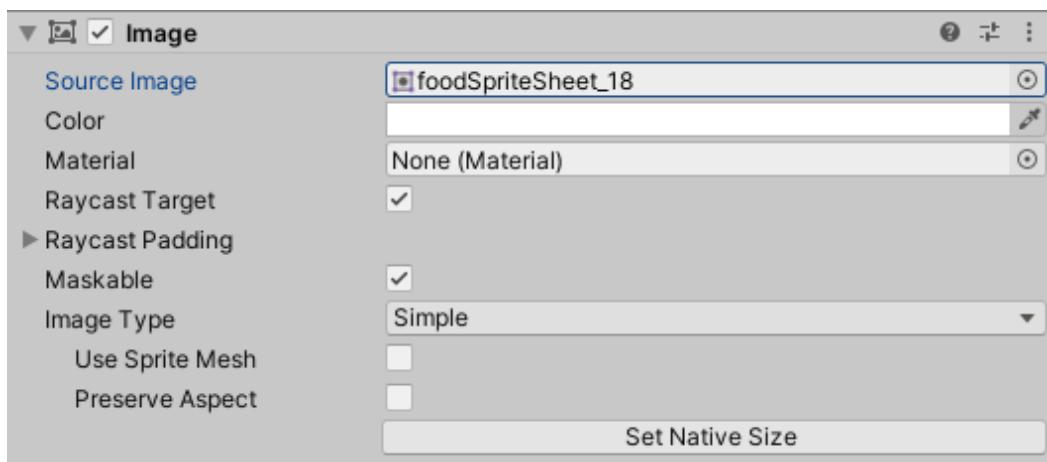


Figure 7.41: The Image component of the Food element

You should see something that looks like this:



Figure 7.42: The resulting Panel with an orange

If your orange and its holder are in a different place than mine, don't worry. When we start adding more children and adjusting the **Horizontal Group Layout**, everything should pop into its proper place.

11. We don't want our orange Image to have its aspect ratio distorted, and we also want to ensure that it always fills the **Item Holder** image without expanding past it. So, let's adjust a few properties on the **Rect Transform** and **Image** components. First, let's ensure that the orange Image will always fill the slot and not expand past it by changing its **Rect Transform** anchor preset to **stretch-stretch**. This won't appear to have changed much, but it will help our orange scale properly if we change the screen size. Also, to ensure that there is a little bit of spacing between the edge of the container and the food, change the **Left**, **Top**, **Right**, and **Bottom** properties to 5.
12. Now, select **Preserve Aspect** on the **Image** component of the orange. Your Food Image should now have the following properties:

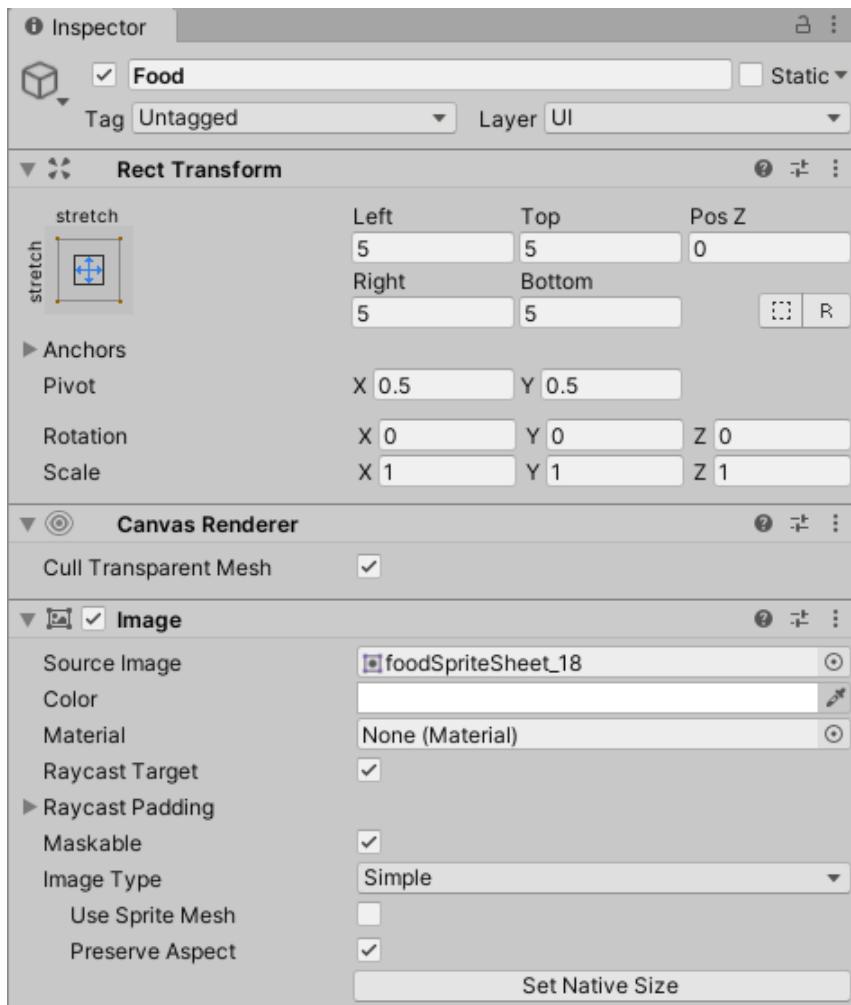


Figure 7.43: The Rect Transform and Image components of the Food UI Image

13. Now, we're ready to start adding some more children. Select the Item Holder Image in the Hierarchy and press *Ctrl + D* four times so that there is a total of five Item Holder GameObjects:

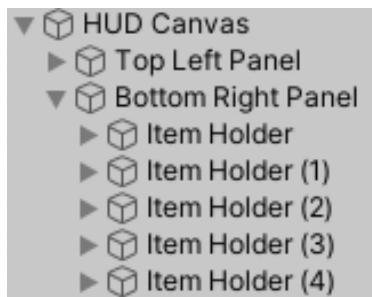


Figure 7.44: The Hierarchy of UI elements

You should see the following in the **Game view**:



Figure 7.45: The resulting Panel with five oranges

14. I don't like my objects to have names with numbered parentheses in them, so I'll rename all the duplicated Images `Item Holder` without the number. Select `Item Holder (1)`, hold `Shift`, and select `Item Holder (4)` so that you have all of them selected. Now, in the **Inspector**, type `Item Holder` in the name slot and press `Enter`. They should all be renamed `Item Holder` now:

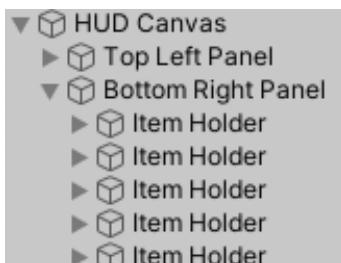


Figure 7.46: The Hierarchy of renamed elements

15. Now, let's adjust the properties on the **Horizontal Layout Group** component of the Bottom Right Panel. To do that, select the Bottom Right Panel, and in its **Horizontal Layout Group** component, give it the following properties:

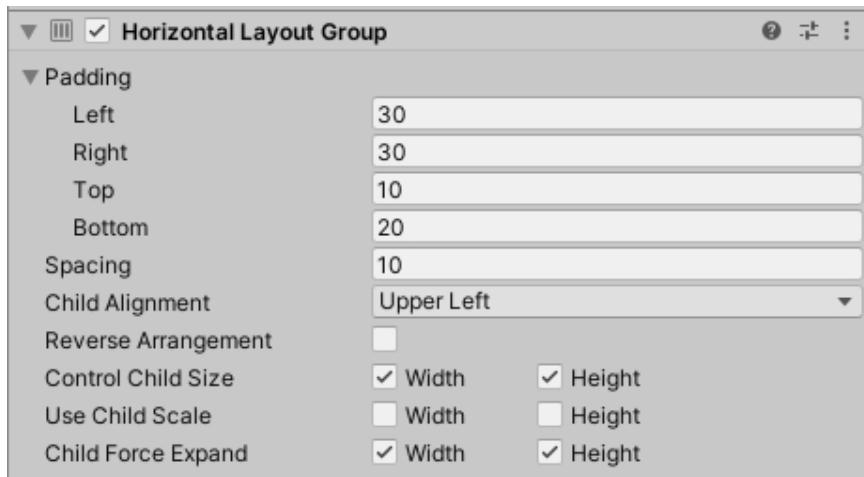


Figure 7.47: The Horizontal Layout Group component of the Bottom Right Panel

You will now be able to see the following:



Figure 7.48: The resulting Panel of oranges

By adjusting the **Padding** properties, we brought the group of objects off the edge of the parent Panel. We spaced them apart by changing the **Spacing** property, and we ensured that the sizes of the **Item Holder** images were adjusted to fit onto the parent Panel by enabling **Control Child Size Width** and **Height**.

16. Now, all that's left to do is swap out the orange images for four other items. Select the Food image of the second through fourth Item Holder GameObjects and change their **Source Image** to a different food. I used foodSpriteSheet_13, foodSpriteSheet_22, foodSpriteSheet_34, and foodSpriteSheet_45 to get the following results:



Figure 7.49: The resulting Panel of a variety of fruit

As you can see from this example, **Horizontal Layout Group** components and (similarly) **Vertical Layout Group** components aren't too difficult to set up and are extremely useful for creating well-organized lists.

Laying out a grid inventory

The last example we'll cover in this chapter is the creation of a gridded inventory system using a **Grid Layout Group** component and the **Content Fitter** component. We'll continue to work on this Panel in later chapters:



Figure 7.50: The grid inventory we will build in this example

To create the gridded inventory system shown in the preceding screenshot, complete the following steps:

1. The shell that holds this inventory system looks remarkably similar to our Pause Panel (see *Figure 7.49*). Since they are so similar, and there is no reason to reinvent the wheel, we will just duplicate the Pause Panel we created in *Chapter 6*, and adjust some of its settings to get the square shape. Select Pause Panel in the Hierarchy and press *Ctrl + D* to duplicate it. Now, rename the duplicate Inventory Panel. Rename its child Image Inventory Banner:

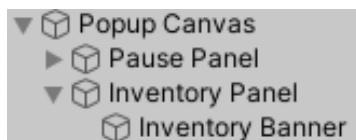


Figure 7.51: The resulting Hierarchy after duplicating and renaming

Note

Remember that we added a Canvas Group component to the Pause Panel in *Chapter 6*. By duplicating it to create the Inventory Panel, the Inventory Panel has a **Canvas Group** component as well. This component will allow us to easily hide and show the two Panels, which we will do in the next chapter.

2. To get the square look of the Inventory Panel in the example screenshot, we need to deselect the **Preserve Aspect** property from the **Image** component and adjust the **Rect Transform** component properties. Change both the **Width** and **Height** values to 500:

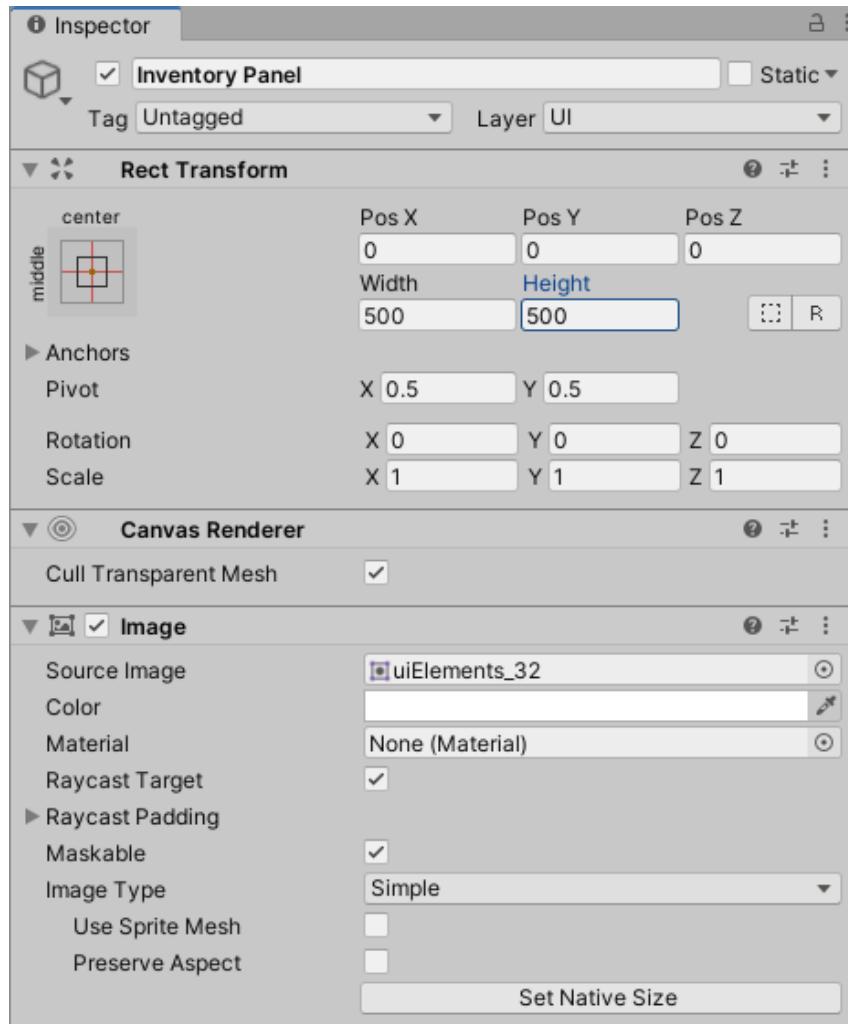


Figure 7.52: The Rect Transform and Image components of the Inventory Panel

You should now see the following in your **Game view**:



Figure 7.53: The resulting Inventory Panel

3. If you look at *Figure 7.50*, you will see that the group of inventory items has an sprite that outlines it. This will act as the parent object of our grid. Create this parent object by right-clicking on the **Inventory Panel** in the Hierarchy and selecting **UI | Panel**. Rename this Panel **Inventory Holder**:

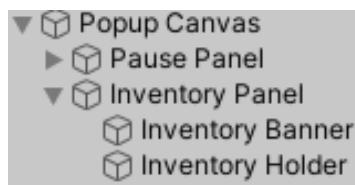


Figure 7.54: The Hierarchy of items

4. Change the **Source Image** to `uiElement_38` and use the **Color** property to make the Image fully opaque.

5. Right now, the **Inventory Holder** is completely covering the **Inventory Panel**. However, we don't need to change any of the **Rect Transform** component properties to resize it because we'll use a **Content Size Fitter** component to adjust the size. Add a **Content Size Fitter** component to the **Inventory Holder** by selecting **Add Component | Layout | Content Size Fitter**. Don't change any of the properties on this component yet. Since the **Inventory Holder** has no children, adjusting the **Horizontal Fit** and **Vertical Fit** settings now will cause it to "disappear."
6. Add a **Grid Layout Group** component to the **Inventory Holder** by selecting **Add Component | Layout | Grid Layout Group**. Once again, don't adjust the settings yet. We'll do this once we add the children.
7. Note from *Figure 7.50* that the children of the inventory are set up just like the children in the horizontal HUD we created in the previous example. So, we'll duplicate the children of the **Bottom Right Panel** and move the duplicates so that they are children of the **Inventory Holder**. Select the first **Item Holder** child of the **Bottom Right Panel**, hold down **Shift**, and select the last **Item Holder** child of the **Bottom Right Panel**. This will select all the children. Now, with all the children selected, press **Ctrl + D** to duplicate them.
8. Click and drag the duplicated **Item Holder** GameObjects in the **Hierarchy** from the **Bottom Right Panel** to the **Inventory Holder**, making them children of **Inventory Holder**:

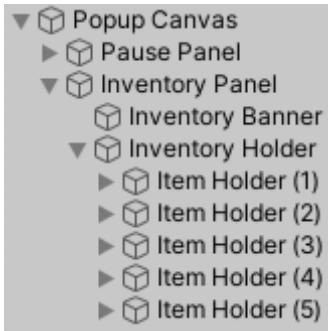


Figure 7.55: The Hierarchy of items

9. Select one of the **Item Holder** GameObjects and duplicate it four times so that there is a total of nine **Item Holder** children. Select all the **Item Holder** children and rename them **Item Holder** so that they no longer have a number in the name:

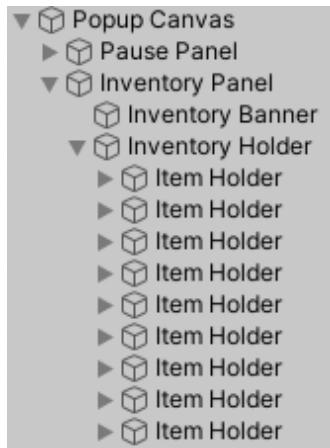


Figure 7.56: The Hierarchy of items

You should now see something similar to *Figure 7.57* in your Game view. Depending on what you duplicated or the order in which you did it, the fruit may be in a slightly different order. That's fine, though!



Figure 7.57: The grid of fruit

10. Now, let's adjust the properties on the **Grid Layout Group** component on the **Inventory Holder Panel** so that the children will be laid out in a 3x3 grid. Adjust the properties to match those in the following screenshot:

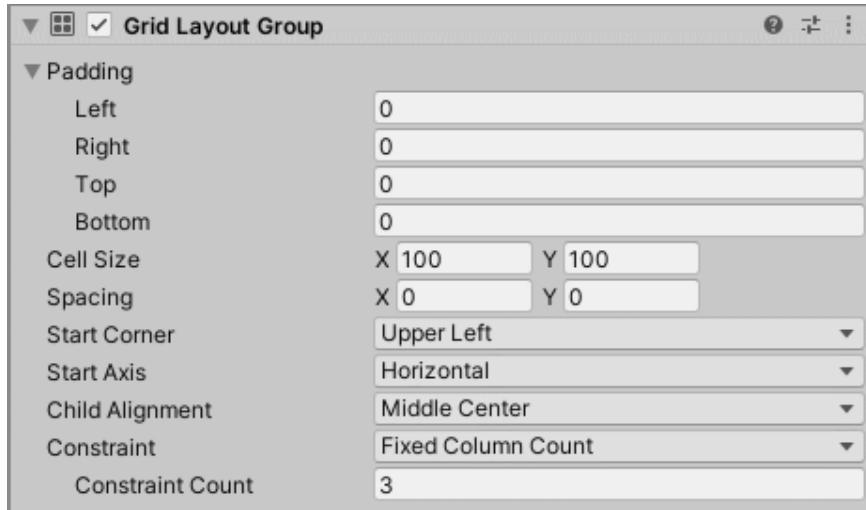


Figure 7.58: The Grid Layout Group component of Inventory Holder

11. You should now see the following in your Game view:



Figure 7.59: The Grid Layout Group component of Inventory Holder

We put spacing between each of the cells using the **Spacing** property, centered the children in the object using the **Child Alignment Middle Center** property, and gave it a 3x3 layout using **Fixed Column Count Constraint** and by setting **Constraint Count** to 3.

- Now that `Inventory Holder` has children, we can change the settings of its **Content Size Fitter**. Set **Horizontal Fit** and **Vertical Fit** to **Min Size**:

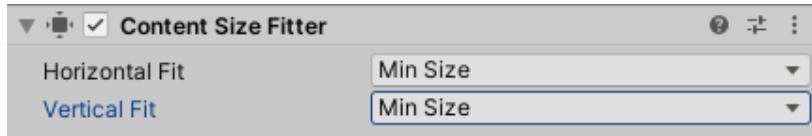


Figure 7.60: The Content Size Fitter component of Inventory Holder

You should now see the following in your Game view:



Figure 7.61: The fitted grid of fruit

It's a little hard to see, but the `Image` of `Inventory Holder` now fits snuggly around the grid of `Item Holder` `Image`s. We want a bit of padding, though.

- Add padding to the sides of the `Item Holder` `GameObjects` by adjusting the **Padding** properties in the **Grid Layout Group**, as shown here:



Figure 7.62: The padded grid of fruit

14. Now that everything is lined up and positioned properly, the only thing left to do is change the order of the images and change the images of the last four slots. To change the order of the images, simply change their order in the **Hierarchy**. The **Grid Layout Group** component will automatically reposition the items within the scene for you. To get the result in the example image, I changed the **Source Image** of the last four Food items to foodSpriteSheet_41, foodSpriteSheet_52, foodSpriteSheet_55, and foodSpriteSheet_53. These changes result in the following completed Inventory Panel:



Figure 7.63: The grid of various foods

That's it. You should now have a perfectly laid out inventory grid.

With the **Grid Layout Group** component set up along with our **Content Size Fitter**, we can now change the number of items in the grid, and the **Inventory Holder** will automatically resize to fit all the items, as you can see here:



Figure 7.64: The smaller grid of various foods

This actually works really well, until we try to add more items to the inventory. You'll see that once we have 10 items, everything looks pretty bad:



Figure 7.65: The expanded grid of various foods

There are a few things we can do to handle this, including changing the cell size and using a mask along with a **Scroll Rect**. We'll discuss how to make those changes in a later chapter. For now, though, just leave your inventory at nine items so that everything looks nice.

After completing all the examples in the last two chapters, you should have the following:



Figure 7.66: The result of all examples from Chapters 6 and 7

And that's in for adding automatic layouts to our scene. We'll continue to improve upon it in future chapters.

Summary

Now, we know all sorts of techniques to lay out our UI elements. The information covered in this chapter, and the last one, has provided enough tools to create almost any UI layout that you can imagine.

The automatic layouts discussed in this chapter aren't just helpful when you want to manually add UI items, as we did in this chapter. These automatic layouts are particularly helpful if you want to dynamically create and add your UI items based on specific conditions.

In the next chapter, we will learn how to access UI components via code and how to use the Event System to allow the player to interact with the UI objects.

8

The Event System and Programming for UI

One of the key features of the Unity UI system is the ability to easily program how the UI elements receive interactions from the player via events. The **Event System** is a robust system that allows you to create and manage events.

Once you learn how to take advantage of the Event System, you will be able to create interactable UI as well as UI that responds to events in your game.

In this chapter, we will discuss the following topics:

- How to access UI elements and their properties via code
- What the Event System is and how to work with it
- How to customize input axes with the Input Manager
- What an Input Module is, and which ones are provided by Unity
- How to use the Event Trigger component to receive events on UI objects
- What Raycasters are and what types of Raycasters are provided by Unity
- How to show and hide pop-up Panels using keyboard inputs
- How to pause the game
- How to create a drag and drop inventory system
- How to use mouse or multi-touch input to pan and zoom the camera

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2008>

Accessing UI elements in code

All the UI elements can be accessed and manipulated in code like other GameObjects. To access a UI element in code, you must include the `UnityEngine.UI` namespace and the correct variable type. Let's look at the `UnityEngine.UI` namespace.

UnityEngine.UI namespace

A **namespace** is a collection of classes. When you include a namespace in your class, you are stating that you want to access all the variables and methods (functions) in your class. Namespaces are accessed at the top of a script with the `using` keyword.

By default, all new C# scripts include the `System.Collections`, `System.Collections.Generic` and `UnityEngine` namespaces. To access the properties of UI elements via code, you must first use the `UnityEngine.UI` namespace.

Therefore, at the top of your C# script, you will need to include the following line to signify that you want to use the `UnityEngine.UI` namespace:

```
using UnityEngine.UI;
```

Without using the namespace, any variable type related to UI elements will be colored red in your code editor, and you will be given a compiler error. Once you include the namespace, the variable type will change to the blue-colored text, signifying that it is an available variable type, and the compiler error will disappear.

UI variable types

Each variable type is a class within the `UnityEngine.UI` namespace. Therefore, each of these variable types, in turn, has its own set of variables and functions that can be accessed. We'll discuss each variable type more thoroughly in future sections and chapters, but for now, let's just look at the standard template for accessing a property of a UI element in code.

You can find within the source files a Unity package named `Chapter 08.unitypackage`. Importing it will bring in a scene named `Chapter8.unity` and various code files. Import the items from the package and open the scene. In the `Chapter8` scene, you will see a UI Image named `UI Variables Example`. It does not have a sprite assigned to it and appears as a white square. The following script, `AddSprite.cs`, is attached to the UI Image:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class AddSprite : MonoBehaviour {
```

```
Image theImage;
public Sprite theSprite;

void Awake() {
    theImage = GetComponent<Image>();
}

void Start () {
    theImage.sprite = theSprite;
    theImage.preserveAspect = true;
}
```

The UI-specific pieces of code are highlighted in the preceding code. Note that the `UnityEngine.UI` namespace is included at the top of the class.

There are two public variables defined in the class: `theImage`, which is an `Image` type, and `theSprite`, which is a `Sprite` type. The `theImage` variable is referencing the UI `Image` in the scene and the `theSprite` variable is referencing the sprite that will become the source image of the UI `Image`.

The `Image` variable type is within the `UnityEngine.UI` namespace and represents UI `Image` `GameObject`. The `Sprite` variable type is not a UI element and is included in the `UnityEngine` namespace.

Within the `Start()` function, the properties of the `Image` component on `theImage` are referenced by typing a period and then the property after the variable name. You can access any property that appears in a UI element's corresponding component in this way. You can also access properties that are not listed in the component this way.

The `AddSprite` script attached to UI Variables Example (`Image`) appears in the inspector, as shown in the following screenshot:

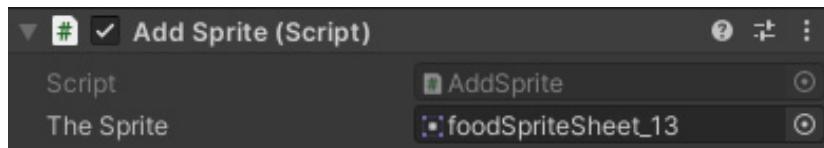


Figure 8.1: The `AddSprite` script and its properties

Now, when the scene is played, the sprite will change from a blank white square to an image of a banana with its aspect ratio preserved.

Let's explore the Event System, which will allow us to interact with our UI.

The Event System

In *Chapter 6*, we learned that when the first Canvas is added to a scene, a GameObject named `EventSystem` is automatically added to the Hierarchy. The Event System allows you to easily receive player interactions and send those interactions to objects in your scene through events. Note that I said, *objects in your scene* and not *UI objects*. The Event System allows you to send events to non-UI items, too!

Before we proceed, I'd like to note my use of `EventSystem` (one word) and Event System (two words), because I will be switching back and forth between the two. I want you to know that I am doing it deliberately and am not just randomly deciding that sometimes I hate the spacebar.

I will use `EventSystem` (one word) to reference the actual GameObject that appears in the Hierarchy of your scene and Event System (two words) to reference the system that handles events.

The Event System does quite a few things for you other than just sending events to objects. It also keeps track of the currently selected GameObject, the Input Modules, and Raycasting.

The `EventSystem` GameObject initializes, by default, with three components: the **Transform**, **Event System** Manager, and **Standalone Input Module**, as shown in the following screenshot:

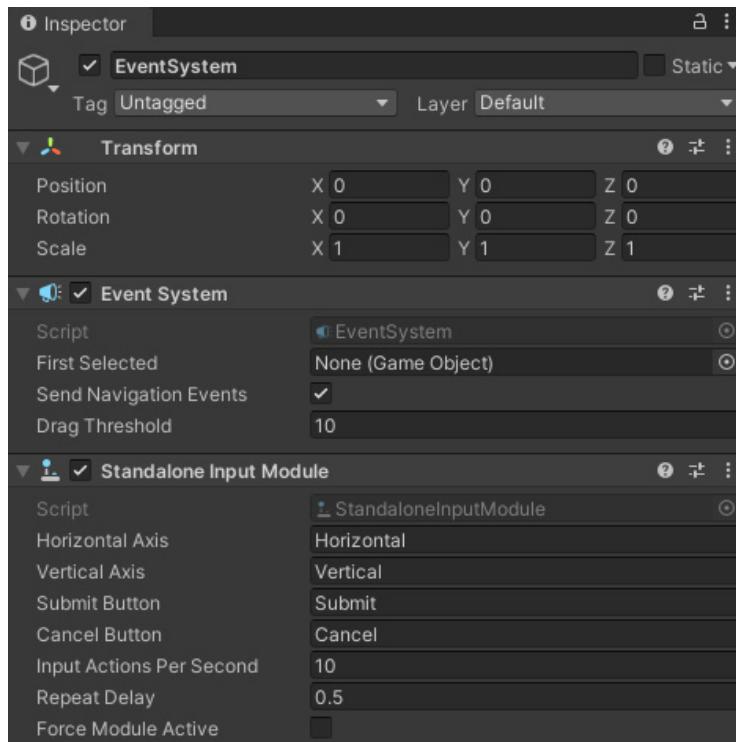


Figure 8.2: The `EventSystem` GameObject and its components

Since `EventSystem` is a `GameObject`, it physically exists within the scene (even though it has no renderable component making it visible) and therefore has a **Transform** component like all other `GameObjects`. You should be familiar with the **Transform** component by now, so we won't discuss it further. However, the other two components do merit further discussion. Let's look at the **Event System** component more closely now. We'll also discuss the **Standalone Input Module** component in the *Input Modules* section of this chapter.

You cannot have more than one `EventSystem` `GameObject` in your scene. If you try to add a new one in the scene via `+ | UI | Event System`, a new one will not be added, and the one currently in the scene will be selected for you.

Note

If you manage to add a second `EventSystem` to your scene (by perhaps using `Ctrl + D` to duplicate the existing one), you will see a warning message on your **Console**.

If you have more than one `EventSystem` `GameObject` in your scene, only the first one added will actually do anything. Any additional `EventSystems` will be non-functional.

Let's look at the **Event System Manager** next.

Event System Manager

Event System Manager is the component that actually does all the tracking and managing of the various **Event System** elements.

If you want to work with the **Event System** without using UI, the `EventSystem` `GameObject` will not be automatically created for you. You can add an **Event System Manager** to a `GameObject` by selecting **Add Component | Event | Event System** on the object's Inspector. Let's talk about the properties under the **Event System Manager**.

First Selected

You know when you start up a game and the **Start Game** button is highlighted for you so that hitting `Enter` will start the game without you having to use your mouse? That's what the **First Selected** property does for you. It selects a UI element in the scene for you automatically when it starts up.

You can drag and drop any intractable UI element into this slot to make it the first selected UI item in your scene. This is particularly helpful for games that do not use a mouse or touchscreen but rely solely on a gamepad, joystick, or keyboard.

Send Navigation Events

The **Send Navigation Events** property can be toggled on and off. When this property is enabled, you can navigate between UI elements via a gamepad, joystick, or keyboard. The following navigation events can be used:

- **Move:** You can select the various UI elements via arrow keys on the keyboard or the control stick on a gamepad (or whichever keys/buttons you have designated as the movement keys). Movement will start at the UI item designated **First Selected**. We will discuss how to specify the order in which UI items are selected using movement in *Chapter 10*.
- **Submit:** Commit to the UI item selected.
- **Cancel:** Cancel the selection.

Drag Threshold

The **Drag Threshold** property represents the number of pixels a UI object can be moved before it is considered being *dragged*. People don't have perfectly steady hands, so when they are trying to click or tap a UI item, their mouse or finger may move slightly. This **Drag Threshold** allows the player to move their input slightly (or a lot if you make this number high) before the item they are selecting is *dragged* rather than *clicked*.

Input Manager

Before we discuss the next component of the **Event System** Manager, I want to discuss the Input Manager. The Input Manager is where you define the axes in your game by assigning them to the buttons on your mouse, keyboard, or joystick (gamepad). This also allows you to use the axis name when coding to easily reference all inputs that you want to perform in an action.

Note

Remember, as we discussed in *Chapter 5*, there are actually two systems that will allow you to handle input in your game: the Input Manager and the new Input System. This chapter will focus on the Input Manager. We will discuss the new Input System in a future chapter.

To open the **Input Manager**, select **Edit | Project Settings | Input Manager**.

If you select the arrow next to **Axes**, you will see the default list of axes:

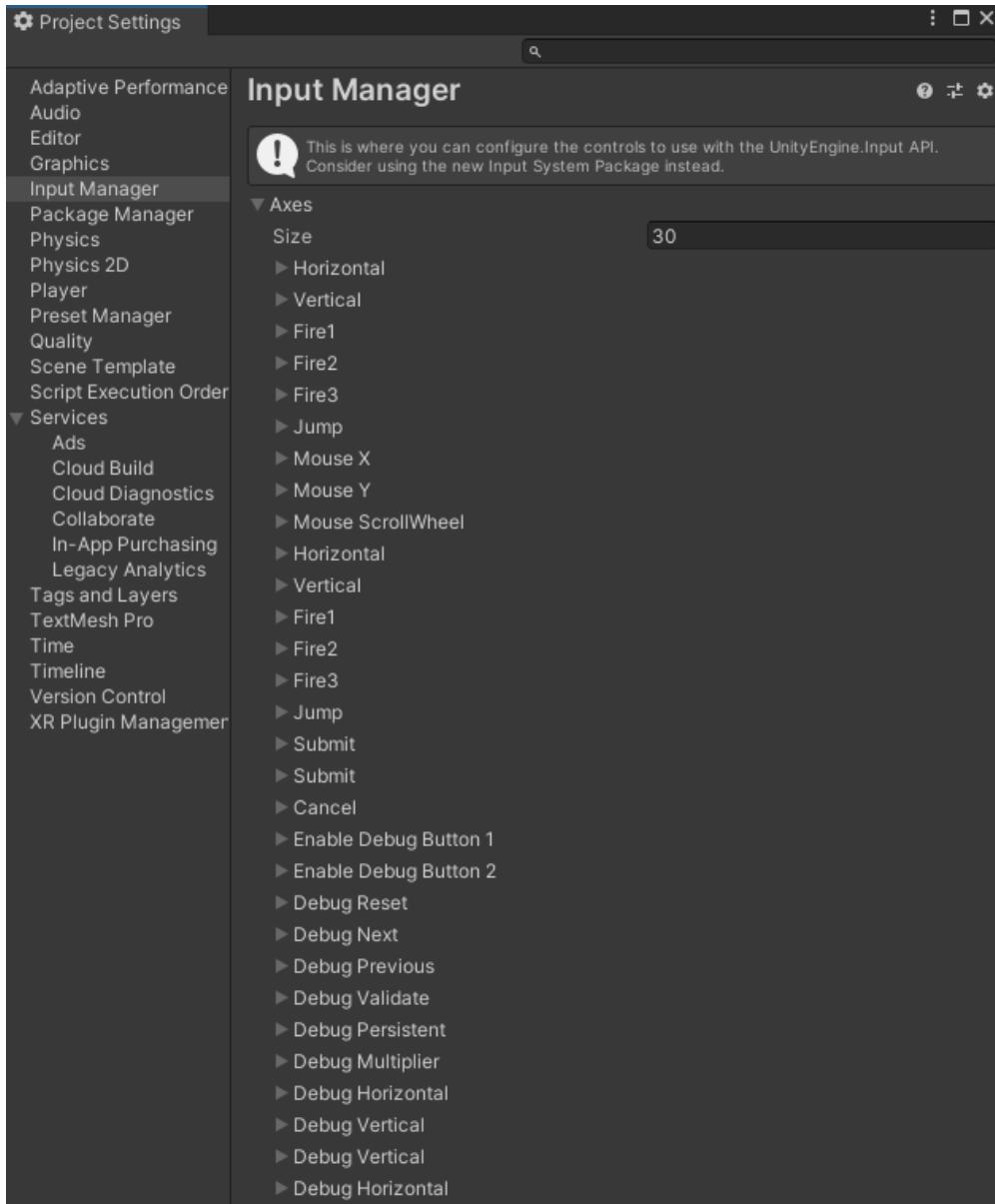


Figure 8.3: The Input Manager and all its pre-defined axes

There are 30 total axes by default. Changing the number next to **Size** will give you more or less axes. Expanding the individual axes will reveal the following:

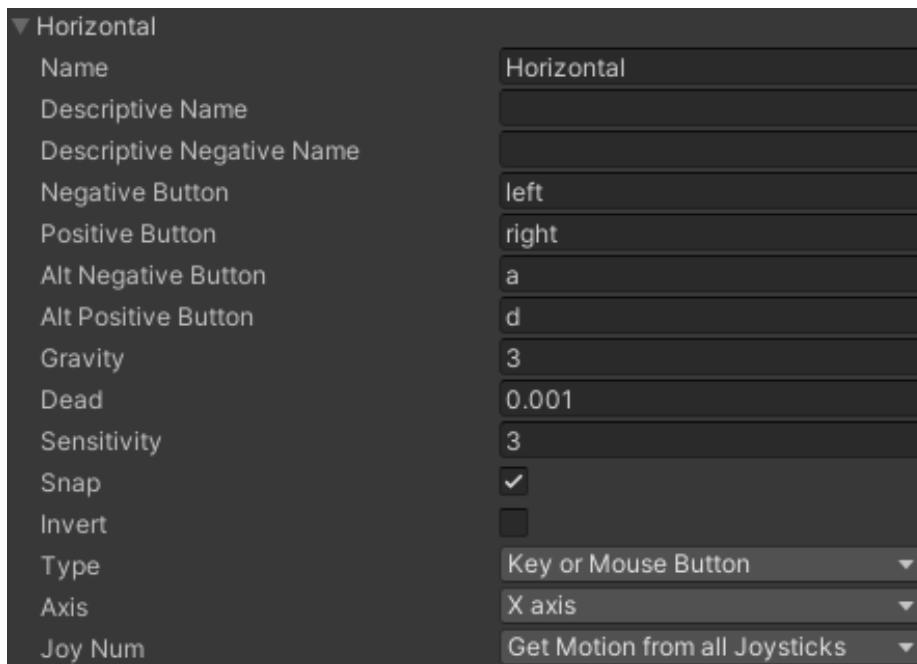


Figure 8.4: The first Horizontal input axis

The word entered in the **Name** slot is what will appear next to the expandable arrow. In the preceding screenshot, all the keys that allow for horizontal movement have been defined.

Note that the left and right arrows, along with the *A* and *D* keys of a keyboard, are defaulted to the **Horizontal** movement.

There is also a second **Horizontal** axis further down the list. It is configured to work with a joystick or a gamepad.

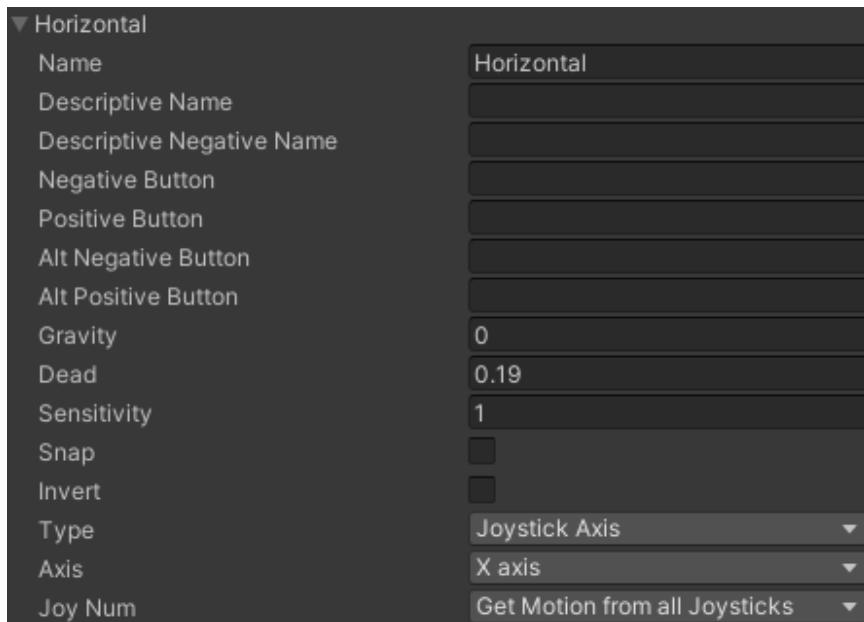


Figure 8.5: The second Horizontal input axis

As there are two **Axes** labeled **Horizontal**, they can both be easily referenced in code with the "Horizontal" label.

Note

To view a list of the keywords for each keyboard key as well as a description for each of the properties of an axis input, visit <https://docs.unity3d.com/Manual/class-InputManager.html>.

This will allow you to reference all these buttons and joysticks together as a group. This is much simpler than having to write code that gets each of the individual keyboard keys along the individual joysticks.

You can delete any of these 30 default axes you want by right-clicking on them and selecting **Delete Array Element**.

However, be careful when you delete them. You need at least one **Submit** axis and one **Cancel** axis to be able to use the **Standalone Input Manager** (unless you change the **Submit Button** and **Cancel Button** in the **Standalone Input Manager**). For more information, refer to the *Standalone Input Manager* section of this chapter.

Now that we have explored the Input Manager, we can review the various input functions for buttons and key presses.

Input functions for buttons and key presses

There are quite a few ways to access key and button presses via code. How you do this depends on whether you have the key specified as an axis in the **Input Manager** and whether you want the key to register once or continuously. I'll discuss a few in this text, but you can find a full list of the functions at <https://docs.unity3d.com/ScriptReference/Input.html>.

A script named `KeyPresses.cs` is attached to the `Main Camera` in the `Chapter8` example scene we were reviewing earlier in this chapter. The `KeyPresses.cs` script contains all the code demonstrated in this section if you'd like to play around with key presses.

GetButton

If you have a button defined as an axis in the **Input Manager**, you can use `GetButton()`, `GetButtonDown()`, and `GetButtonUp()` to determine when a button has been pressed.

`GetButton()` returns `true` while the button is being held, `GetButtonDown()` returns `true` only once, on the frame that the button is initially pressed, and `GetButtonUp()` returns `true` only once, on the frame that the button is released.

Within each of the functions, you place the axis name from the **Input Manager** within the parentheses within quotes. Generally, these functions should be called within the `Update()` function of a script so that they can be triggered at any time.

So, for example, if you wanted to check whether the *Enter* key is being pressed, since it is assigned to a **Positive Button** for the **Submit** axis, you can write the following code to trigger when the *Enter* key is pressed down:

```
void Update () {
    if (Input.GetButtonDown("Submit")){
        Debug.Log("You pressed a submit key/button!");
    }
}
```

Keep in mind that this will not just trigger with the *Enter* key, as the **Submit** axis has a few keys assigned to the **Positive Button** and **Alt Positive Button**.

Note

It's important to note that if you play the `Chapter8` scene and want to watch these button and key presses fire the console log messages, you must first click within the **Game View** so that the inputs will register in the game.

GetAxis

If you're looking for a function that will trigger continuously without any breaks between firing, you want to use `GetAxis()` rather than `GetButton()`. `GetButton()` is good for buttons you want to hold down but want a slight pause between events firing (think of holding down a fire button, and the gun shoots bullets with breaks in between them). `GetAxis()` works better for events involving movement because of this continuous frame-rate independent execution.

`GetAxis()` works a bit differently, as it returns a `float` value rather than a `bool`, such as `GetButton()`. It is also best suited within an `Update()` function. So, for example, you can check whether the horizontal movement is occurring as follows:

```
void Update () {
    float horizontalValue = Input.GetAxis("Horizontal");
    if (horizontalValue != 0){
        Debug.Log("You're holding down a horizontal button!");
    }
}
```

GetKey

If you want to get a keyboard key press that is not assigned to an axis, you can use `GetKey()`, `GetKeyDown()`, or `GetKeyUp()` to reference keyboard keys via their `KeyCode`.

The `GetKey()` functions work pretty similar to the `GetButton()` functions. `GetKey()` returns `true` while the key is being held down; `GetKeyDown()` returns `true` only once, on the frame that the key is initially pressed; and `GetKeyUp()` returns `true` only once, on the frame that the key is released.

Each key has its own `KeyCode` that needs to be referenced in the parentheses of the `GetKey()` functions. You can find a list of all the keyboard `KeyCode` values at <https://docs.unity3d.com/ScriptReference/KeyCode.html>.

So, for example, if you wanted to check whether the `8` key from the alphanumeric keyboard is being pressed, you could write the following code to trigger when the `8` key is pressed down:

```
void Update () {
    if (Input.GetKeyDown(KeyCode.Alpha8)){
        Debug.Log("You pressed the 8 key for some reason!");
    }
}
```

GetMouseButton

Just as with `GetButton()` and `GetKey()`, there are three functions for checking when a mouse button has been pressed: `GetMouseButton()`, `GetMouseButtonDown()`, and `GetMouseButtonUp()`. They return `true` in the same way that the `GetButton()` and `GetKey()` functions do.

You'd place these functions within the `Update()` function as well. Within the parentheses, you check to see which button is being pressed; 0 represents a left-click, 1 represents a right-click, and 2 represents a middle-click.

So, for example, if you wanted to check that the middle mouse button was clicked, you could write the following code to trigger when the middle mouse button is pressed down:

```
void Update () {
    if (Input.GetMouseButtonDown(2)) {
        Debug.Log("You pressed the middle mouse button!");
    }
}
```

Now that we've reviewed the input function for buttons and key presses, let's review the Input Modules.

Input Modules

Input Modules describe how the Event System will handle the inputs to the game via the mouse, keyboard, touchscreen, gamepad, and so on. You can think of them as the bridge between the hardware and events.

There are three input modules provided by Unity:

- Standalone Input Module
- Base Input Module
- Pointer Input Module

To utilize these input modules, you attach them as components to your `EventSystem` `GameObject`.

You are not restricted to using these three input modules and can create your own, so if you have an input device that is not covered by one of those three, you'd create your own input module script and then attach it to the Event System.

There is another input module called Touch Input Module, which used to be necessary for touchscreen inputs. However, this module has been deprecated and its functionality is now lumped into the Standalone Input Module. Since this input module has been deprecated, it will not be discussed in this text.

Let's look at the three input modules provided by Unity in depth.

Standalone Input Module

The **Standalone Input Module** is a pretty robust input module that will work with most of your input devices. It works with a mouse, keyboard, touchscreen, and gamepad.

The **Standalone Input Module** is automatically added to your `EventSystem` `GameObject` when it is created. However, you can attach the **Standalone Input Module** as a component using **Add Component** | **Event** | **Standalone Input Module** on the object's **Inspector**. You could do this if you wanted to add a second one, previously deleted it, and want to re-attach it, or want to add the **Standalone Input Module** to another `GameObject`.

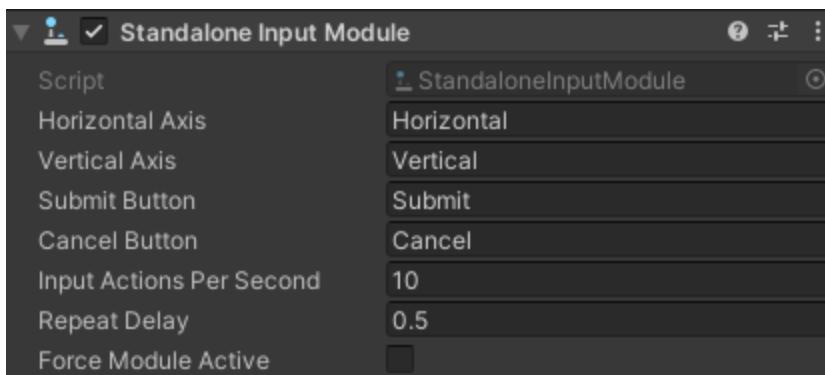


Figure 8.6: The Standalone Input Module component

You'll see that the first four properties of the **Standalone Input Module** are **Horizontal Axis**, **Vertical Axis**, **Submit Button**, and **Cancel Button**. These properties are the reason I wanted to discuss the Input Manager before discussing the Input Modules. The default properties assigned to these slots are **Horizontal**, **Vertical**, **Submit**, and **Cancel**. These assignments are referencing the axes assignments from the Input Manager.

The **Input Actions Per Second** property defines how many inputs are allowed per second. This is in relation to the keyboard and the gamepad inputs. The default value is 10. This means that there will be a tenth of a second delay after an input action before the next input action is registered. The **Repeat Delay** property is the amount of time, in seconds, before **Input Actions Per Second** occurs.

Setting the **Force Module Active** property to true will make this **Standalone Input Module** active.

Note

You can learn more about the Standalone Input Module at the following locations:

<https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/script-StandaloneInputModule.html>

<https://docs.unity3d.com/2019.1/Documentation/ScriptReference/EventSystems.StandaloneInputModule.html>

BaseInputModule/PointerInputModule

The `BaseInputModule` and `PointerInputModule` are modules that are only accessible via code.

If you need to create your own Input Module, you will create it by extending from the `BaseInputModule`. You can view a full list of the variables, functions, and messages that can be utilized by extending the `BaseInputModule` at <https://docs.unity3d.com/Packages/com.unity.ugui@1.0/api/UnityEngine.EventSystems.BaseInputModule.html>.

The `PointerInputModule` is a `BaseInputModule` that is used by the Standalone Input Module described earlier. It can also be used to write custom Input Modules. You can view a full list of the variables, functions, and messages that can be utilized by extending the `PointerInputModule` at <https://docs.unity3d.com/Packages/com.unity.ugui@1.0/api/UnityEngine.EventSystems.PointerInputModule.html>.

Now, let's look at how we can access multi-touch input on mobile and touchscreen devices.

Input for multi-touch

Accessing multi-touch is pretty easy. You access touches with `Input.GetTouch(index)`, where the index represents the index of the touch, with the first touch occurring at index 0. From there, you can access information pretty much in the same way as accessing information about a mouse. You can also find out how many total touches are occurring with `Input.touchCount`. See the *Examples* section of this chapter for an example of how to access multi-touch input.

Mobile devices also have accelerometers and gyroscopes providing input to the device. Let's look at how you can access those inputs.

Input for accelerometer and gyroscope

You can access data from the device's accelerometer using the `Vector3 Input.acceleration` property. The coordinates of `Input.acceleration` line up with the scene based on the rotation of the device, as shown:

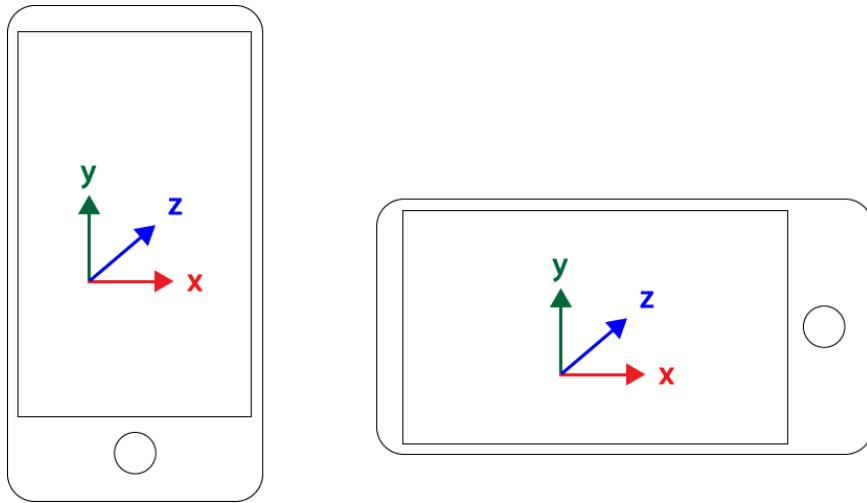


Figure 8.7: The world axes based on screen rotation

Simple examples of this involve moving an object around a scene when the device is moved, using something like the following within an `Update()` function on the object:

```
transform.Translate(Input.acceleration.x, 0, -Input.acceleration.y);
```

The gyroscope uses more complicated mathematics to get a more precise movement of the screen using the `Gyroscope` class. Remember, the gyroscope is not supported on many devices, so it's best to use the accelerometer when possible.

Note

An example of how to use the gyroscope on an iOS device can be found here: <https://docs.unity3d.com/ScriptReference/Gyroscope.html>.

Now that we've reviewed the various input modules, let's review Event Triggers.

Event Trigger

The **Event Trigger** component can be attached to any UI (or non-UI) element to allow the object to receive events. Some of the UI elements are preconfigured to intercept specific events. For example, buttons have the `onClick` event. However, if you'd like to add an event to an object that either isn't already set up to receive events or you want it to receive different events, you can attach an **Event Trigger** component to the `GameObject`.

You can attach an **Event Trigger** component by selecting **Add Component** | **Event** | **Event Trigger**.

One caveat of using the **Event Trigger** component is that the object it is attached to receives all the events, not just the ones you added. So, even if you don't tell the object what to do with the specified event, it will receive that event and acknowledge that the event occurred—it just won't do anything in response. This can slow the performance of your game. If you are worried about performance, you will want to write your own script that attaches only the events you want to use to your component. The next section, *Event Inputs*, discusses how to achieve this.

If you use an **Event Trigger** component on an object other than a UI element, the object must also have a collider component, and you must include a raycaster on the camera within the scene.

Which collider and raycaster you use depends on whether you are working in 2D or 3D.

If you are working in 2D, you can add a 2D collider to the object with **Add Component | Physics 2D** and then select the appropriate 2D collider from within the object's **Inspector**. You can then add a raycaster to the camera by selecting **Add Component | Event | Physics 2D Raycaster** from within the camera's **Inspector**.

If you are working in 3D, you can add a 3D collider to the object with **Add Component | Physics** and then select the appropriate 3D collider from within the object's **Inspector**. You can then add a raycaster to the camera by selecting **Add Component | Event | Physics Raycaster** from within the camera's **Inspector**.

Let's look at the various event types that the Event Trigger can receive.

Event types

You can tell the object which type of input event you want to receive by selecting **Add New Event Type**.

Many of these events are tied to the bounding region of the object. The bounding region of a UI object is represented by the area of the Rect Transform. For a non-UI object, the bounding region is represented by a 2D or 3D collider.

Pointer events

Pointer events can be called by the pointer in a **Standalone Input Module**. Remember that a pointer is not exclusively a mouse. The pointer in a **Standalone Input Module** can be a mouse, finger touch, or a reticle tied to gamepad movement.

Two of the event types are related to the position of the pointer in relation to the object's bounding box region. The **PointerEnter** event is called when the pointer enters the bounding box of the object and **PointerExit** is called when the pointer exits the bound box area.

There are three events related to clicking on the object. The **PointerDown** event is called when the pointer is pressed down within the bounding region of the object, and **PointerUp** is called when the pointer is released within the bounding region of the object. It's important to note that with **PointerUp**, the pointer can be pressed outside of the object, held down, and then released inside the object for the **PointerUp** event to trigger. The **PointerClick** event is called when the pointer is pressed and then released within the bounding region of the object.

Drag and Drop events

When working with the various drag and drop events, it's important to differentiate between the object being dragged and the object on which the dragged object is dropped.

The **InitializePotentialDrag** event is called whenever a drag object is found, but before an object is actually being dragged.

The **Drag** event is called on the object being dragged when it is being dragged. A **Drag** event occurs when a pointer is pressed within the bounding box of an object and then moved without releasing. It's ended by releasing the pointer. The **BeginDrag** event is called from the object being dragged when its drag begins, and the **EndDrag** event is called when its drag ends.

The **Drop** event is different from the **EndDrag** event. The **EndDrag** event is called on the object that was just being dragged. The **Drop** event is called by the object on which the dragged object was dropped. Therefore, the **Drop** event is called by the object touching the dragged object when the dragged object stops dragging. So, if you were making a drag and drop menu, you'd add the **Drag** event to the objects you want to drag and the **Drop** event to the slots they will be dropped into.

Selection events

The **Select** event is called when the object is considered the selected object and **Deselect** is called when the object is no longer considered selected. Each of these events only fires once—the moment the object is considered selected or deselected. If you want an event that will trigger continuously while the object is selected, you can use the **UpdateSelected** event. The **UpdateSelected** event is called every frame.

Other events

Other events are called based on assignments in the Input Manager. Remember that you can assign buttons, keys, and such to axes that define movement, submit, and cancel. Let's talk about a few of these events.

The **Scroll** event is called when the mouse wheel scrolls and the **Move** event is called when a movement happens. When the button assigned to the **Submit** axis is pressed, the **Submit** event is called and when the button assigned to the **Cancel** axis is pressed, the **Cancel** event is called.

Adding an action to the event

Once you have actually selected an event type, you must specify what will happen when that event type triggers. The following screenshot shows the results of selecting **Pointer Enter** as an event type:

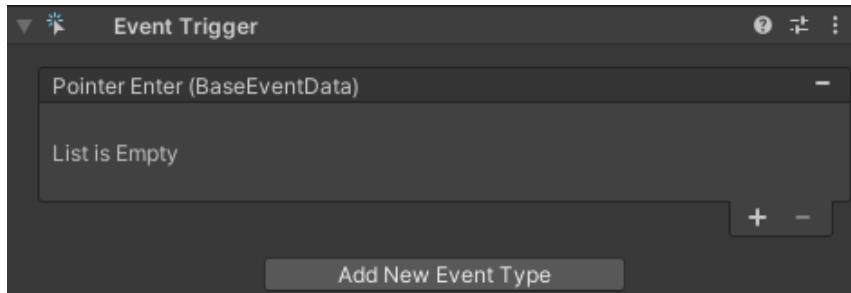


Figure 8.8: The Event Trigger component

The preceding screenshot shows that an event type of **Pointer Enter** has been selected, but what happens when the pointer enters the object's bounding area is yet to be defined. To define what happens when the event triggers, you must select the + sign at the bottom-right corner of the event's box. You can add multiple actions when the event triggers by selecting the + sign multiple times.

Once an event type has been added to the **Event Trigger** component, it cannot be added a second time and will be grayed out in the **Add New Event Type** list.

To remove an event type from the **Event Trigger** component, select the – sign at the top-right corner of the event type's box.

Once the plus sign is selected, the event type should look as follows:

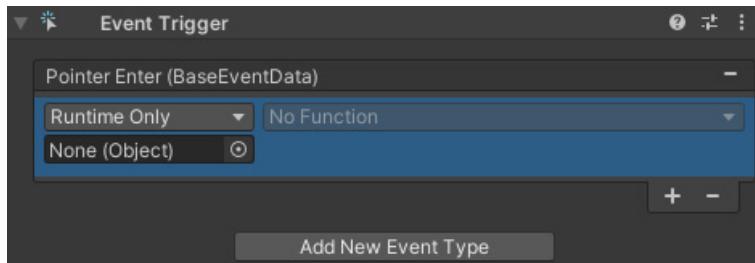


Figure 8.9: The Event Trigger component with a Pointer Enter event

The first setting on this event is a dropdown menu with the **Runtime Only** (by default), **Editor and Runtime**, and **Off** options. This is where we specify when the event can be triggered. Setting this to **Off** will make the event never trigger. Setting this to **Runtime Only** will have the event trigger when the game is being played. Setting this to **Editor and Runtime** will make events trigger when the game is being played, but it also accepts the triggers in the Editor when the game is not in play mode. Most of the time, **Runtime Only** is sufficient for what you will be doing and hence it is the default.

Below that dropdown menu is a slot with **None (Object)** in it. You are to drag from the Hierarchy whichever item the function you want to run is attached to into this slot. Once that is assigned, a list of all the available components and scripts attached to that object will display in the second dropdown menu. You can drag and drop the object the **Event Trigger** is attached to in this slot and are not restricted to only using other objects.

The following screenshot shows an **Image** GameObject with an **Event Trigger** added to it and the **Pointer Enter** event type. The same image is added to the slot, signifying to look at the components on itself. The image component's sprite property is to change to the `foodSpriteSheet_1` sprite when the pointer enters its **Rect Transform**.

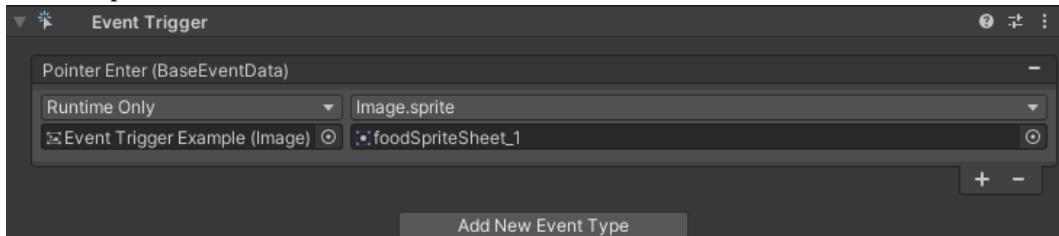


Figure 8.10: The Event Trigger component with a Pointer Enter event that swaps a sprite

To see this **Event Trigger** in action, play the `Chapter8` scene. Hover your mouse over the image. It will initially look like a potion bottle but will change to a triangle when your mouse hovers over it.

You can also run functions within scripts attached to objects. For example, the next screenshot shows the same image but now with a **Pointer Click** event as well. `Main Camera` has a script attached to it called `HelloWorld.cs` with a function called `HeyThere()`.

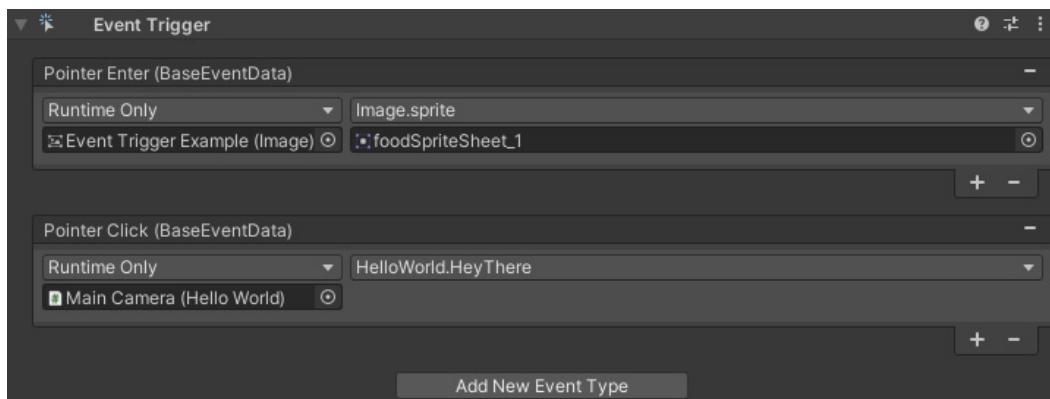


Figure 8.11: The Event Trigger component with Pointer Click event that triggers a method

The `HeyThere()` function simply prints `Hello world!`. This is main camera speaking! in the **Console** whenever the image to the right is clicked.

To run a function from the **Event Trigger** component, it must be public, have a return type of void, and have no more than one parameter.

Now, let's review how we can write code that performs similarly to the Event Trigger component through the use of event inputs.

Event inputs

As stated in the *Event Trigger* section, you may not want to use an **Event Trigger** component because the **Event Trigger** component causes the object on which it is attached to receive all the events listed in the *Event Trigger* section. So, if you are worried about performance issues, you will want an alternate way to receive events on an object.

All event types that were available to add in the *Event Trigger* section can also be added to an object via code without using the **Event Trigger** component. To use an event without the **Event Trigger** component, you must derive your script from the appropriate interface and know the type of event data class that the event uses.

An **interface** is a template that defines all the required functionality that a class can implement. So, by using an interface, you can then use any of the methods or functions that have been defined within that interface. I'll show you some examples of how to do this, but first, let's look at the available events and their required interfaces.

There are three classes that the event data can be derived from, which are `PointerEventData`, `AxisEventData`, and `BaseEventData`:

- `PointerEventData` is the class that contains events associated with the pointer
- `AxisEventData` contains events associated with the keyboard and gamepad
- `BaseEventData` contains events that are used by all event types

There is a fourth event data class, `AbstractEventData`. It is the class from which the other three inherit.

The list of events available for a `StandaloneInputModule` along with their required interfaces and event data class are provided in the following chart. The events are listed in the same order they are listed within the Event Trigger component for continuity purposes:

Event	Interface	Event Data Type
<code>OnPointerEnter</code>	<code>IPointerEnterHandler</code>	<code>PointerEventData</code>
<code>OnPointerExit</code>	<code>IPointerExitHandler</code>	<code>PointerEventData</code>
<code>OnPointerDown</code>	<code>IPointerDownHandler</code>	<code>PointerEventData</code>
<code>OnPointerUp</code>	<code>IPointerUpHandler</code>	<code>PointerEventData</code>
<code>OnPointerClick</code>	<code>IPointerClickHandler</code>	<code>PointerEventData</code>

Event	Interface	Event Data Type
OnDrag	IDragHandler	PointerEventData
OnDrop	IDropHandler	PointerEventData
OnScroll	IscrollHandler	PointerEventData
OnUpdateSelected	IUpdateSelectedHandler	BaseEventData
OnSelect	IselectHandler	BaseEventData
OnDeselect	IDeselectHandler	BaseEventData
OnMove	IMoveHandler	AxisEventData
OnInitializePotentialDrag	IIInitializePotentialDragHandler	PointerEventData
OnBeginDrag	IbeginDragHandler	PointerEventData
OnEndDrag	IEndDragHandler	PointerEventData
OnSubmit	ISubmitHandler	BaseEventData
OnCancel	ICancelHandler	BaseEventData

Table 8.1: Interfaces and event data types for the various events

To write a class with one of these events, you will use the following template:

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;
public class ClassName : MonoBehaviour, InterfaceName{
    public void EventName(EventDataTypeName eventData) {
        //what happens after event triggers
    }
}
```

The items highlighted in the preceding code will be replaced by the items within the preceding table.

For example, if you wanted to implement an OnPointerEnter event, the code would look as follows after the highlighted code has been replaced with an appropriate event, interface, and event data type:

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;
public class ClassName : MonoBehaviour, IPPointerEnterHandler
    public void OnePointerEnter(PointerEventData eventData) {
        //what happens after the event triggers
    }
}
```

You must include the `UnityEngine.EventSystems` namespace to write code with event data. The `UnityEngine.UI` namespace is optional and is only required if you will also be writing your events for UI objects.

Now that we've reviewed various ways to send and receive events, let's look at raycasters.

Raycasters

Remember that the Event System keeps track of raycasting along with all the other things we have discussed. Raycasting is used to determine which UI elements are being interacted with by projecting a ray from the user's pointer into the scene. This ray is considered to originate at the camera's plane and then proceed forward through the scene. Whatever this ray hits receives an interaction. You can have the ray continue through the first UI element it hits or stop at the first UI element it hits. To get a ray to stop at the first UI element it hits, the object must block raycasting. This will stop items behind it from being interacted with. Next, we'll discuss the types of raycasters.

Graphic Raycaster

When a Canvas is added to the scene, it is automatically given a **Graphic Raycaster** component.

This is the raycasting system that will allow you to interact with all UI objects that are children of that Canvas. It has three properties: **Ignore Reversed Graphics**, **Blocking Objects**, and **Blocking Mask**.

The **Ignore Reversed Graphics** toggle determines whether or not graphical objects within the Canvas can be interacted with if they are facing backward (in relation to the raycaster). The **Blocking Objects** and **Blocking Mask** properties allow you to assign the types of objects that are in front of the Canvas (between the camera and the Canvas) that can block raycasting to the Canvas.

Other Raycasters

As stated earlier, if you want to use the Event System with a non-UI object, you must attach a Raycaster component to a camera within the scene. You can add either a **Physics 2D Raycaster** or a **Physics Raycaster** (or both) to your camera based on whether you are using 2D or 3D.

From within the camera's inspector, you can add the **Physics 2D Raycaster** by selecting **Add Component** | **Event** | **Physics 2D Raycaster** and the **Physics Raycaster** by selecting **Add Component** | **Event** | **Physics Raycaster**.

The two components appear as follows:



Figure 8.12: The two types of Physics Raycasters

The **Event Mask** property determines which types of objects can receive raycasting.

If you attempt to add either of these components to a non-camera GameObject, a **Camera** component will automatically be attached to the GameObject as well.

Now that we've reviewed the various systems that we can use to program interactions for our UI, let's look at some examples.

Examples

We will continue to work on the UI we have been building for the last two chapters. To help organize the project, duplicate the **Chapter7** scene that you created in the last chapter; it will automatically be named **Chapter8**.

Note

If you did not work through the examples for *Chapter 6*, and *Chapter 7*, but would like to work through the examples in this chapter, you can import the package labeled **Chapter 08 – Examples 1 – Start**. You can also view the completed examples in the **Chapter 08 – Examples 1 – End** package.

Showing and hiding pop-up menus with keypress

So far, we have made two Panels that we plan on turning into popups: the **Pause Panel** from *Chapter 6*, and the **Inventory Panel** from *Chapter 7*. Right now, they are both visible in the scene (even though **Pause Panel** is hidden behind the **Inventory Panel**). We want them to pop up when we press **P** and **I** on the keyboard. For demonstration purposes, we'll access the keyboard keys differently for each Panel.

Remember that both of these Panels have Canvas Group components on them. These components will allow us to easily access the Panels' alpha, intractable, and blocks raycasts properties.

Using KeyCode with the Inventory Panel

Let's begin with the `Inventory Panel`. We want the Panel to pop up and close when the `I` key is pressed on the keyboard. To make the `Inventory Panel` appear and disappear with the `I` key, complete the following steps:

1. Create a new C# script in the `Assets/Scripts` folder by right-clicking within the **Project** view of the folder and selecting **Create | C# Script** from the pop-up Panel.
2. Name the script `ShowHidePanels.cs`, and then double-click on it to open.
3. Now, let's use a `public CanvasGroup` variable called `inventoryPanel` to represent the Panel. We use a `CanvasGroup` variable type to reference the Panel since we want to access the properties of the **Canvas Group** component. Update your `ShowHidePanels` script to include the following highlighted line of code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ShowHidePanels : MonoBehaviour {
    public CanvasGroup inventoryPanel;
}
```

The `CanvasGroup` variable type, even though it is used with UI elements, is not in the `UnityEngine.UI` namespace, but the `UnityEngine` namespace, so we do not need to include the `UnityEngine.UI` namespace in our script at the moment.

4. Let's create another variable that will keep track of whether or not the `Inventory Panel` is visible. Add the following code to the next line of the script to initialize the variable:

```
bool inventoryUp = false;
```

5. We will be toggling the Panels on and off by adjusting their `alpha`, `interactable`, and `blocksRaycasts` properties. When the Panel is hidden, it should also not accept interactions or block raycasts. So, let's create a method that we can call to perform the toggle. Add the following namespace to your script:

```
using System;
```

Add the following method to your script:

```
public void TogglePanel(CanvasGroup Panel, bool show)
{
    Panel.alpha = Convert.ToInt32(show);
    Panel.interactable = show;
    Panel.blocksRaycasts = show;
}
```

As you can see, the method has two parameters. The first parameter is a CanvasGroup called `Panel1` and the second parameter is a Boolean called `show`. It will set the `alpha` property to 0 when `show` is `false` and 1 when `show` is `true`. It will also set the `interactable` and `blocksRaycasts` properties to `false` when `show` is `false` and `true` when `show` is `true`.

- We want `Inventory Panel` to be hidden when the scene starts playing. So, update the `Start()` function to include the following code:

```
void Start () {
    TogglePanel(inventoryPanel, inventoryUp);
}
```

- Now, we need to write code that triggers whenever the `I` key on the keyboard is pressed down. We will use the `Input.GetKeyDown()` function in a way that the function is called the moment the key is pressed down. We will also use `KeyCode.I` to reference the `I` key on the keyboard. Add the following code to your `Update` function to check whether the `I` key is pressed down:

```
void Update () {
    //inventory Panel
    if (Input.GetKeyDown(KeyCode.I)) {

    }
}
```

- We want this key to disable and enable the Panel, so we will change the value of `inventoryUp` to whatever the opposite of its current value is. That is, if it is `true`, we will set it to `false`, if it is `false`, we will set it to `true`. Then, we will call the `TogglePanel()` method.

Add the following highlighted code to your `Update()` function:

```
void Update()
{
    // Inventory Panel
    if (Input.GetKeyDown(KeyCode.I))
    {
        inventoryUp = !inventoryUp;
        TogglePanel(inventoryPanel, inventoryUp);
    }
}
```

- Now, for this code to work, we need to attach it to a `GameObject` within our scene. It really doesn't matter what `GameObject` we attach it to, since we used a public variable to access our `Inventory Panel`, we can assign that via the `Inspector`. However, since we are planning on using this script to affect both Panels, I want to add it to `Main Camera`. Drag and drop the `ShowHidePanels` script into the `Inspector` of `Main Camera`. You should now see the following as a component on your `Main Camera`:

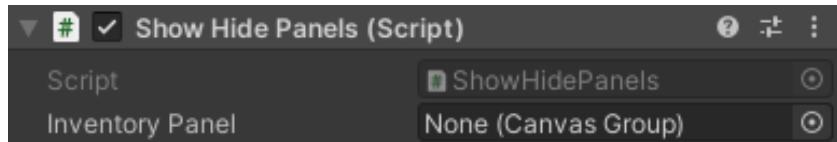


Figure 8.13: The ShowHidePanel.cs script component

- Now, we need to assign the `Inventory Panel` GameObject to the slot labeled **Inventory Panel**. Drag and drop the **Inventory Panel** from the Hierarchy into this slot:

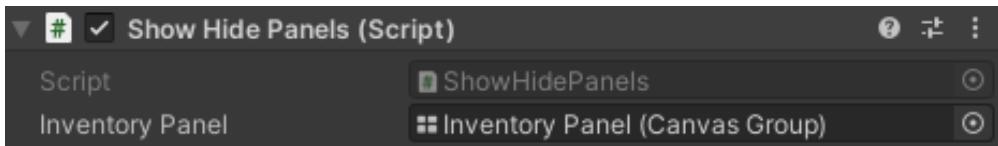


Figure 8.14: Adding the Inventory Panel ShowHidePanel.cs script component

- Play the game to ensure that the code is working correctly. You should see the inventory Panel start out invisible and then turn on and off as you press the *I* key on the keyboard.

Now that we've completed the work needed to show and hide the `Inventory Panel`, we can move on to the `Pause Panel`.

Using Input Manager with the Pause Panel

Now, let's do the same thing for the `Pause Panel`. We'll do this slightly differently than the `Inventory Panel`. To make sure that you can see how to access a key with the Input Manager, we'll use the Input Manager instead of a KeyCode. We also need to actually pause the game.

To display the `Pause Panel` using the *P* key and pause the game, complete the following steps:

- First, we need to set up the Input Manager to include a `Pause` axis. Open the Input Manager with **Edit | Project Settings | Input Manager** and expand the axes by selecting the arrow next to the word **Axes**.
- By default, your project has 30 axes. You can replace one of these with the new `Pause` axis if you aren't planning on using them, but we might as well just go ahead and make a new one. Definitely don't delete the **Submit** and **Cancel** axes, as we have them being referenced in our **Standalone Input Manager**. To add a new axis, change the size to 31. This will duplicate the last axis in the list, **Debug Horizontal**, as shown in the following screenshot:

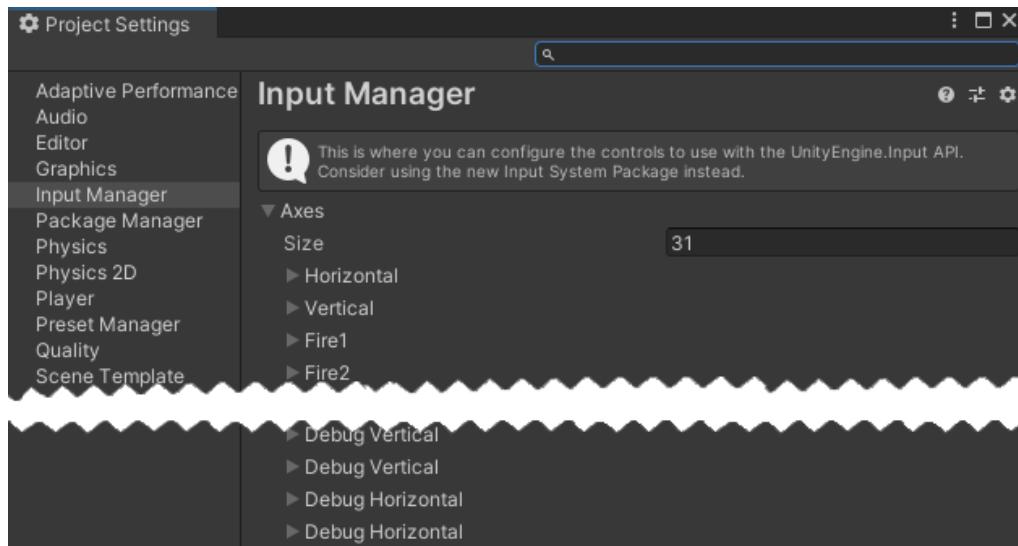


Figure 8.15: The Input Manager with an extra axis

3. Change the second Debug_Horizontal axis to a Pause axis by changing the **Name** to Pause, the **Positive Button** to p, and changing the rest of the properties to the following:

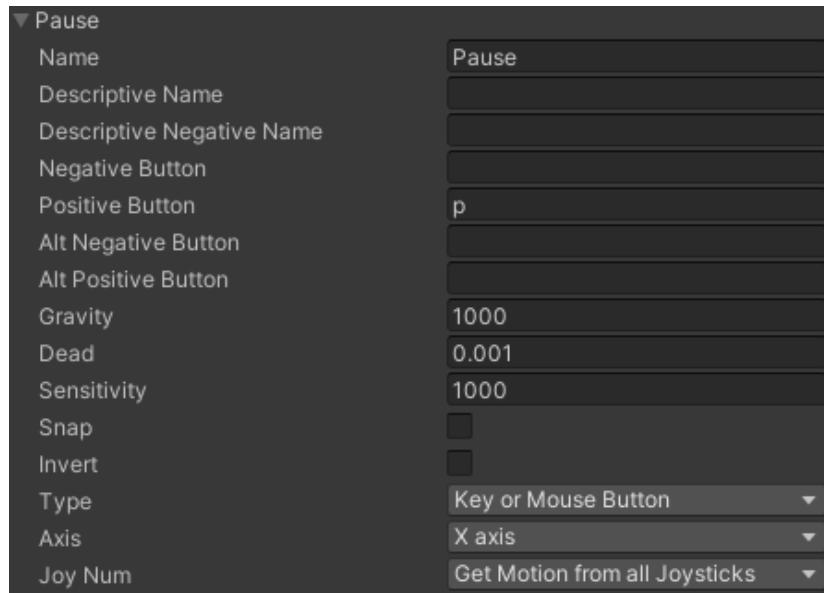


Figure 8.16: The Pause axis added to the Input Manager

4. Now that we have our Pause axis set up, we can start writing our code. Let's define some variables to use with the Pause Panel similar to the way we defined variables for Inventory Panel. Add the following variable definitions at the top of your class under your previous variable definitions:

```
public CanvasGroup pausePanel;bool pauseUp = false;
```

5. Add the following to the Start () function and make the Pause Panel invisible at start:

```
TogglePanel(pausePanel, pauseUp);
```

6. Since we added the Pause axis to our Input Manager, we can use Input .GetButtonDown () instead of Input .GetKeyDown (), like we did with Inventory Panel. We want to use GetButtonDown () rather than GetAxis () because we want a function that will return true once, not continuously. If it returned continuously (using GetAxis ()), the Panel would flicker in and out while the P key was being pressed. Add the following code at the end of your Update () function. Note that it's very similar to the code we used for Inventory Panel:

```
// pause Panel
if (Input.GetButtonDown ("Pause")) {
    pauseUp = !pauseUp;
    TogglePanel(pausePanel, pauseUp);
}
```

7. Now that we've added new public variables to our script, it should be showing up in the Inspector of Main Camera. Drag and drop the Pause Panel from the Hierarchy to the Pause Panel slot.

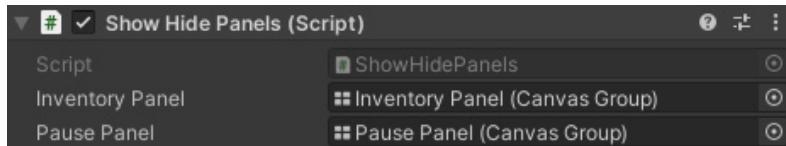


Figure 8.17: The ShowHidePanel.cs script component with the Pause Panel added

8. Now, play the game and watch the Pause Panel become visible and invisible when you press the P key on the keyboard.

Next, we'll learn about pausing the game.

Pausing the game

The game doesn't actually pause right now. If we had animations or events running in the scene, they would continue to run even with the Pause Panel up. A really easy way to pause a game is to manipulate the time scale of the game. If the time scale is set to 1, time will run as it normally does. If the time scale is set to 0, the time within the game will pause.

Also, our current setup doesn't quite work as a pause menu would be expected to. Inventory Panel and Pause Panel can be displayed at the same time. If Inventory Panel is up, the Pause Panel is covered up by it since it is rendering behind it. Also, the Inventory Panel can be activated when the game is *paused*.

We'll need to pause the time scale of our game, change the order that our Panels render, and disable functionality when the game is paused to have a Pause Panel that functions properly. To create a properly functioning Pause Panel, complete the following steps:

1. Add the following to the `Update()` function to the `ShowHidePanels` script to pause the time in the game:

```
// pause Panel
if(Input.GetButtonDown("Pause")){
    pauseUp = !pauseUp;
    TogglePanel(pausePanel, pauseUp);
    Time.timeScale = Convert.ToInt32(pauseUp);
}
```

2. Now, let's deal with the fact that Pause Panel is behind the Inventory Panel. This is an easy fix. Simply change their order in the Hierarchy by dragging the Pause Panel below the Inventory Panel. The items that are listed lower in the Hierarchy render on top of the ones listed above it within the scene. Now, the Pause Panel will be above the Inventory Panel in the scene:

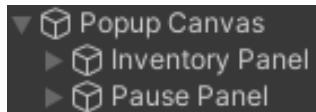


Figure 8.18: The children of the Popup Canvas

3. The only thing left to do is to disable the ability of the Inventory Panel to appear and disappear if the Pause Panel is up. Adjust the `if` statement that checks for the `I` key being pressed to also check whether `pauseUp` is false, like so:

```
if(Input.GetKeyDown(KeyCode.I) && !pauseUp){
```

4. Play the game now and see that when the game is paused, Inventory Panel cannot be activated or deactivated. If the Inventory Panel is activated when the Pause Panel is already up, it cannot be deactivated until after the game is unpause.

It's important to remember that when you have a Pause Panel, other events need to be turned off. Setting the timescale to 0 does not stop the ability for other events to occur; it only really stops animations and any clocks you may have displayed that use the time scale. So, we will need to ensure that any other event we program is turned off when the game is paused.

Dragging and dropping inventory items

We have an `Inventory Panel` that can be displayed and hidden and a `HUD inventory`. I want to be able to drag objects from my larger `Inventory Panel` to my smaller `HUD inventory` called `Bottom Right Panel` that we created in the previous chapter.

To make things a little easier for ourselves, let's disable the `ShowHidePanels` script that we added to the `Main Camera` earlier in this chapter. You can do this by deselecting the checkbox next to the script's component on the `Main Camera`:

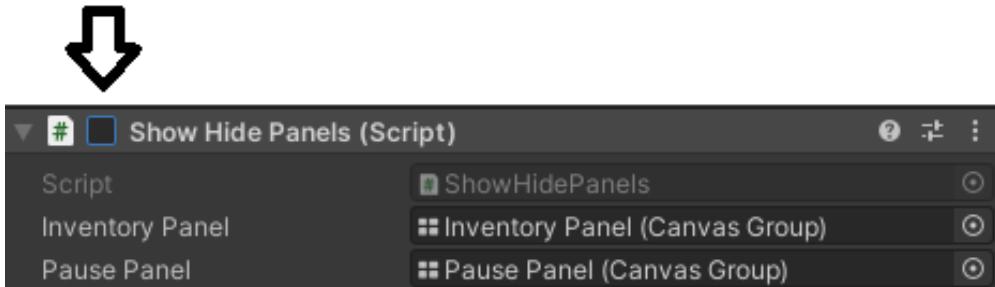


Figure 8.19: Disabling the `ShowHidePanel.cs` script component

Let's also disable `Pause Panel` so that it will not be in our way. Do this by deselecting the checkbox next to the name of the `Pause Panel` in its Inspector.

Now, our Panel will stay visible, making it easier for us to debug the code we're about to write.

There are quite a few different ways to make a drag and drop mechanic. To ensure that this chapter provides an example of how to use the `Event Trigger` component, we will write a drag and drop mechanic utilizing it. To create a drag and drop mechanic for the `Inventory Panel` and `Bottom Right Panel`, complete the following steps:

1. Create a new C# script in the `Assets/Scripts` folder called `DragAndDrop.cs` and open it.
2. We will be referencing UI elements in this script, so add the `UnityEngine.UI` namespace to the top of the script with this:

```
using UnityEngine.UI;
```

3. We only need to add two variables to this script: one will represent the `GameObject` being dragged, and the other represents the `Canvas` that the items will be dragged on. Add the following public variables to the top of the class:

```
public GameObject dragItem; public Canvas dragCanvas;
```

4. Before we write any more code, let's go back to the Editor and do a bit more prep work. Drag the `DragAndDrop.cs` script to the `Inspector` of the `Main Camera` to attach it as a component:

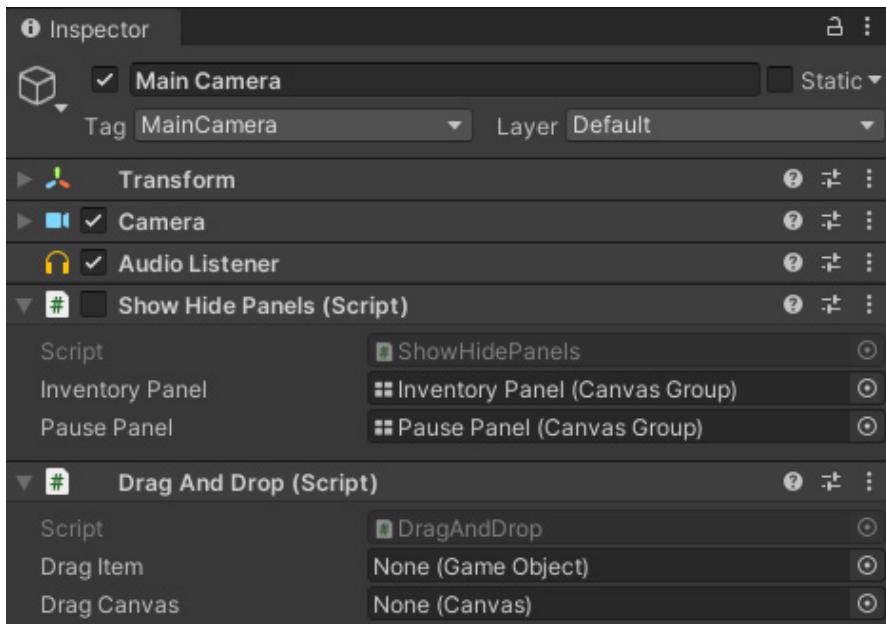


Figure 8.20: The components of the Main Camera

I've chosen to create a script that attaches to the `Main Camera` rather than the individual inventory items to reduce the need to duplicate this script.

5. Now, create a new UI Canvas by selecting `+ | UI | Canvas` from the Hierarchy menu. Name the new Canvas `Drag Canvas`.
6. Select `HUD Canvas` and copy its **Canvas Scaler** component by selecting the settings three dots (aka “the kabob”) in its top-right corner and selecting **Copy Component**.
7. Reselect `Drag Canvas` and paste the copied **Canvas Scaler** properties to its **Canvas Scaler** component by selecting the three dots in its top-right corner and selecting **Paste Component Values**.

Once that is done, it should have the following values:

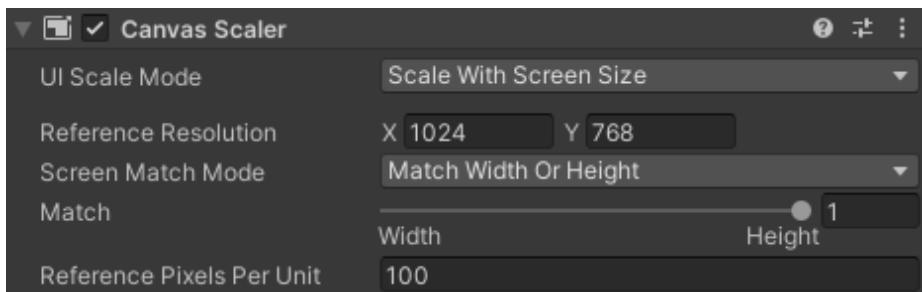


Figure 8.21: The Canvas Scaler component on the Drag Canvas

- Set the **Sort Order** property on the Drag Canvas' Canvas component to 1. This will cause anything that is on the Drag Canvas to render in front of all other Canvases since the other Canvases have a **Sort Order** of 0:

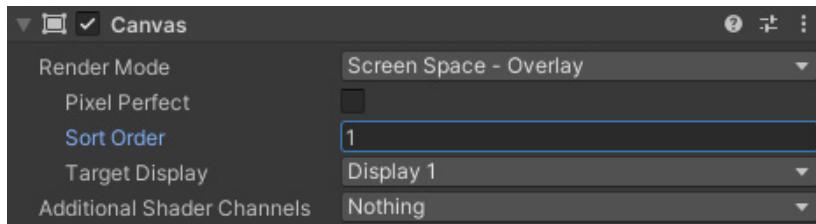


Figure 8.22: Updating the Sort Order on the Drag Canvas' Canvas component

- Drag and drop the Drag Canvas from the Hierarchy into the **Drag Canvas** slot on the DragAndDrop script component on the Main Camera:



Figure 8.23: The Drag and Drop component

- Reopen the DragAndDrop script. Create a new function called `StartDrag()`, as follows:

```
public void StartDrag(GameObject selectedObject){
    dragItem = Instantiate(selectedObject, Input.mousePosition,
selectedObject.transform.rotation) as GameObject;
    dragItem.transform.SetParent(dragCanvas.transform);
    dragItem.GetComponent<Image>().SetNativeSize();
    dragItem.transform.localScale = 1.1f * dragItem.transform.
localScale;
}
```

This function will be called when a drag begins. It accepts a `GameObject` as a parameter and then creates a new instance of it at the position of the mouse. It then moves it so that it is a child of `dragCanvas`. Lastly, it sets the size of the sprite on the `Image` component to native size. This resets the scale of the `Image`'s `RectTransform` to its sprite's original pixel size. (Refer to *Chapter 12* for more on **Set Native Size**). The last line makes the image 10% bigger than its native size.

Note

After we hook up our `BeginDrag` and `Drag` events, if you comment out the line of code that sets the size to native, you'll see that the `Image` does not actually render in the scene, because its scale is «wacky» from the original `GameObject` being within a **Layout Group**.

11. Now, create a new function called `Drag()`, as follows:

```
public void Drag() {
    dragItem.transform.position = Input.mousePosition;
}
```

This function will be called when an object is being dragged. While the object is dragged, it will keep position with the mouse.

12. Return to the Editor. We will just hook the events to the first object in the `Inventory Panel` for now. Select the first `Food` image in the `Inventory Panel`:

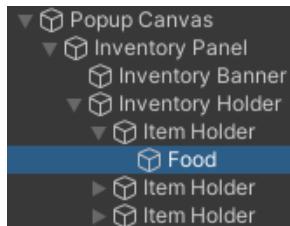


Figure 8.24: Selecting the Food GameObject

13. Add a new Event Trigger component to the `Food` Image by selecting **Add Component | Event | Event Trigger** within its **Inspector**:

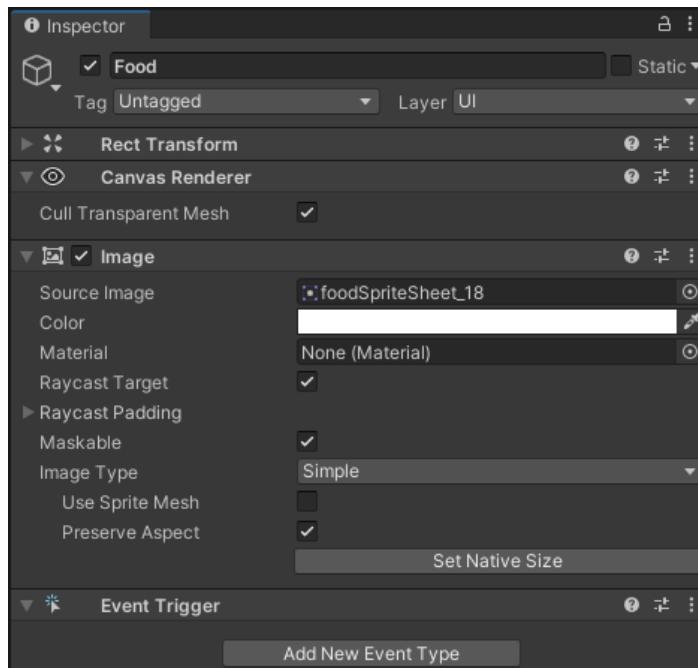


Figure 8.25: The Food GameObject with the Event Trigger component

14. Now, add a **Begin Drag** event type and a **Drag** event type to the **Event Trigger** list by selecting **Add New Event Type | BeginDrag** and **Add New Event Type | Drag**:

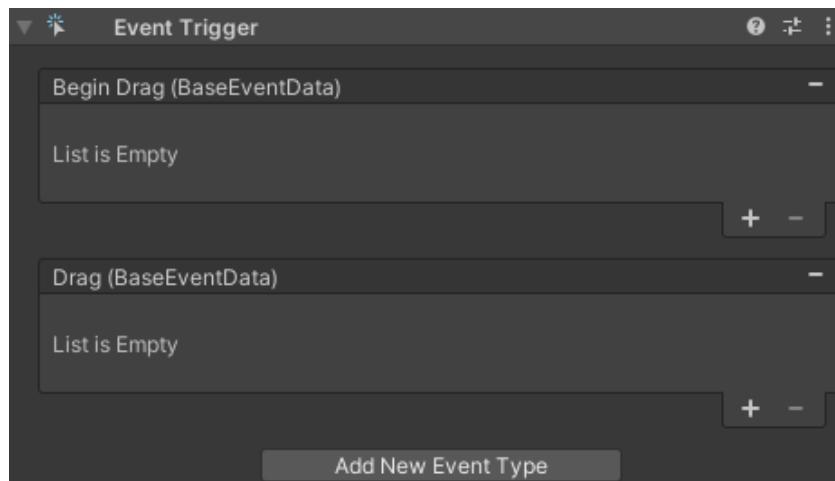


Figure 8.26: The Event Trigger component with two events

15. Now, we will add an action to the **Begin Drag** list by selecting the plus sign at the bottom-right corner of the **Begin Drag** area:

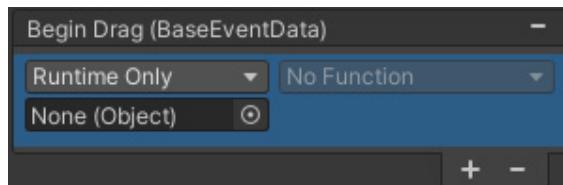


Figure 8.27: Adding a Begin Drag event

16. Drag the **Main Camera** into the object slot:

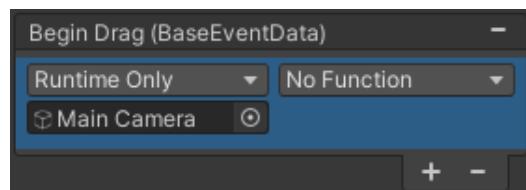


Figure 8.28: Updating the Begin Drag event with the camera

17. The function dropdown list is now intractable. Expand the function dropdown list to see the list of functions, components, and such attached to the **Main Camera**. Find the **DragAndDrop** script and then the **StartDrag (GameObject)** function:

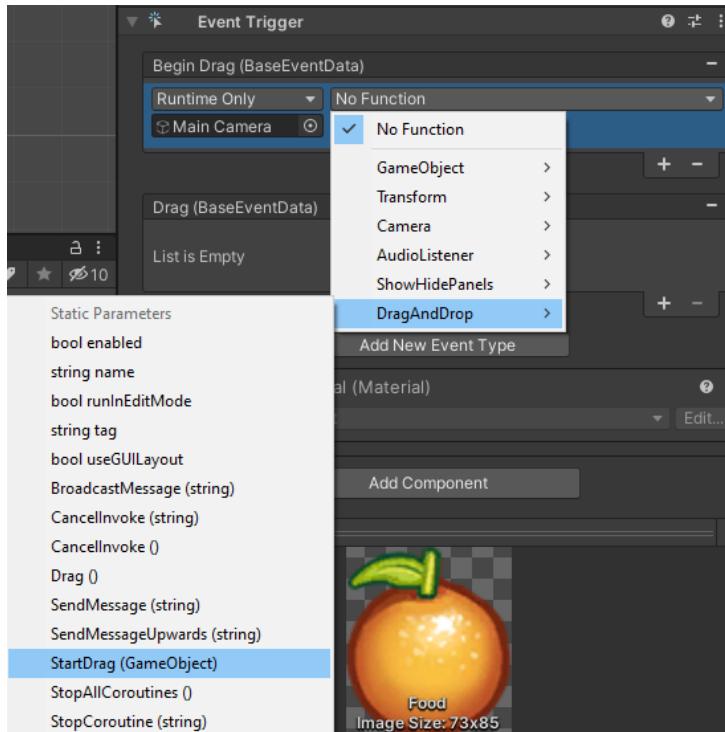


Figure 8.29: Adding the StartDrag method

Once you have done so, you should see the following:

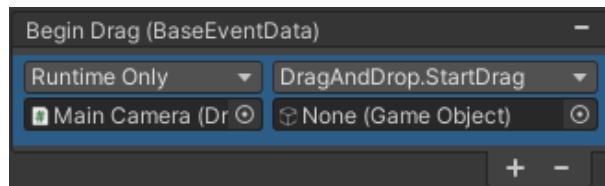


Figure 8.30: Adding the StartDrag method

18. Now, we need to assign the **GameObject** parameter. Drag and drop the **Food** Image that this **Event Trigger** component is attached to into the parameter slot.

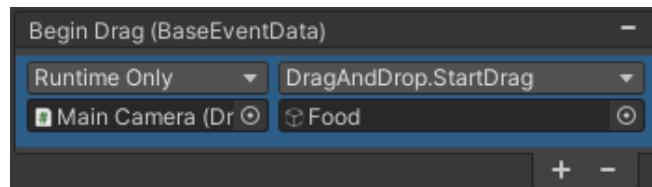


Figure 8.31: Updating the StartDrag method

19. Now, set up the **Drag** event list similarly so that it looks like this:

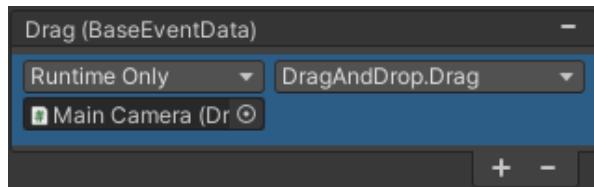


Figure 8.32: Adding the Drag method

20. If you play the game, you should now be able to drag the orange in the first slot out of its slot.



Figure 8.33: Dragging the orange from the inventory

You'll see, in the Hierarchy, there is a new GameObject called **Food (Clone)** that is a child of the **Drag Canvas**. This is the orange that gets created when you begin dragging.

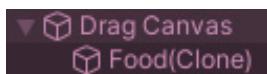


Figure 8.34: The item being dragged in the Drag Canvas

At this point, you can actually make as many of these clones as you want. In a moment, however, we will make it so that there is only one clone in the **Drag Canvas** at a time.

21. Go back to the **DragAndDrop** script and create a new function called **StopDrag()**, as follows:

```
public void StopDrag () {
    Destroy(dragItem);
}
```

This code will destroy the **Food (Clone)** GameObject once it is no longer being dragged.

22. Go back to the Editor and reselect the Food Image in Inventory Panel. Give its **Event Trigger** component an EndDrag event type by selecting **Add New Event Type | EndDrag**. It will automatically assign the Drag () function from the DragAndDrop script to this event since that was the last selected function:

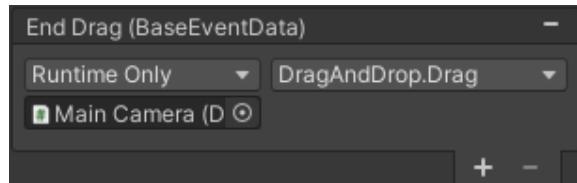


Figure 8.35: The End Drag event with the Drag method

23. Replace the Drag () function in the function dropdown with the StopDrag () function:

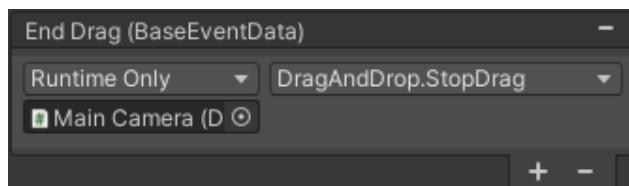


Figure 8.36: The End Drag event with the StopDrag method

24. Play the game, and you will see that the orange can now be dragged out of its slot, and when you release the mouse, it is destroyed. This stops you from being able to drag a bunch of oranges from this slot.
25. Go back to the DragAndDrop script and create a new function called Drop (), as shown in the following:

```
public void Drop(Image dropSlot){
    GameObject droppedItem = dragCanvas.transform.GetChild(0).gameObject;
    dropSlot.sprite = droppedItem.GetComponent<Image>().sprite;
}
```

This function accepts an `Image` as a parameter. This `Image` will be the `Image` component of the slot that will receive a drop. The first line of the function finds the first child of the `dragCanvas` (at position 0) and then assigns its `Image`'s `sprite` to the `sprite` of the `dropSlot`. Since we have set up the `StopDrag()` function to destroy the object being dragged once it stops dragging, we don't have to worry about there being more than one child of the `Drag Canvas` `GameObject`, making this the easiest way to find the object being dragged.

26. Go back to the Editor and select the second Food Image in the Bottom Right Panel:



Figure 8.37: Selecting the correct Food item

We're using the second Food Image, rather than the first, because the first already has an orange in it, and it will be hard to tell that our script worked in that slot.

27. Add a new **Event Trigger** component to the Food Image by selecting **Add Component | Event | Event Trigger** within its Inspector.
28. Add a Drop event type to the **Event Trigger** component by selecting **Add New Event Type | Drop**:

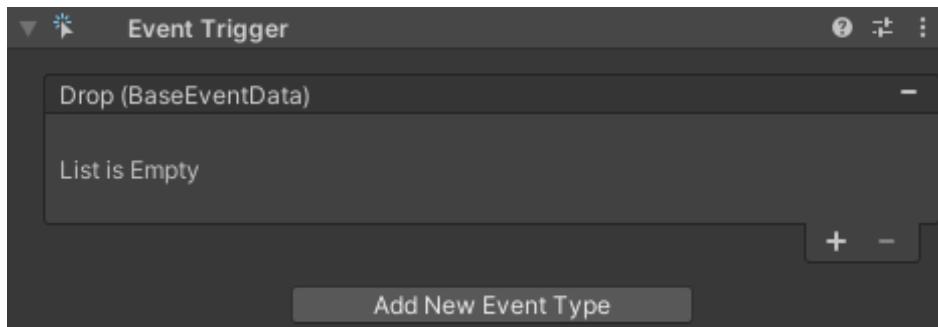


Figure 8.38: The Drop event

29. Add a new action to the list with the + sign.
30. Drag `Main Camera` to the object slot and select the `Drop()` function from the `DragAndDrop` script:

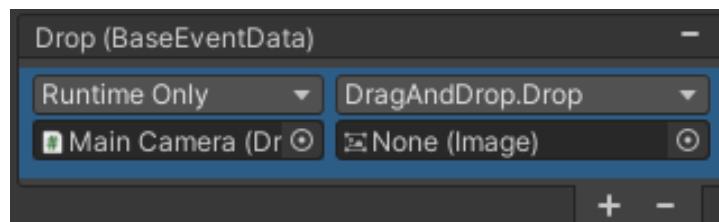


Figure 8.39: The Drop event with its method populated

31. Now, drag the Food Image that this **Event Trigger** component is attached to into the parameter slot:

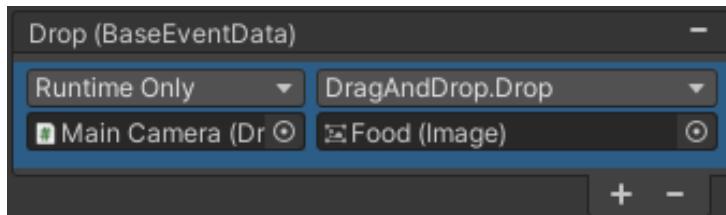


Figure 8.40: The Drop event with the correct parameter

32. Play the game and when you drag the orange into the slot over the banana, the `Drop()` function doesn't appear to trigger. This is because the orange being dragged is blocking the raycasting from reaching the banana, so the banana never thinks anything is dropped on it. This is an easy fix. Add the following line of code to the end of the `StartDrag()` function in the `DragAndDrop` script so that the raycast won't be blocked by the orange anymore:

```
dragItem.GetComponent<Image>().raycastTarget = false;
```

33. Play the game, and you should now be able to drag the orange from the first slot of the Inventory Panel to the second slot of Bottom Right Panel.
34. The functionality of drag and drop is now complete; we just need to add the functionality to the other slots. Let's add the drag events to the other Inventory Panel items first.

Copy the **Event Trigger** component of the Food Item in the Inventory Panel that we hook the events up to, using the three dots in the component's top-right corner.

35. Select all the other Food Images in the Inventory Panel by clicking on them while holding *Ctrl*.
36. Now, with all eight Food Images still selected, click on the three dots on the **Image** component and select **Paste Component as New**. Each of the Food Images should now have the **Event Trigger** component with all the appropriate events.
37. We're not done with these other inventory items yet. We need to select each one and drag it into the parameter of the `BeginDrag` event type for its component. Otherwise, each of these other eight Food items will drag out oranges instead of the appropriate Food item, because the original orange is assigned to that slot. Do so now.
38. Before continuing, play the game and ensure that each inventory item in Inventory Panel drags out the appropriate image.

39. Now, we will copy the drop events from the second Food Image image in the Bottom Right Panel to all the other Food Images within the Panel. Copy the **Event Trigger** component from the second Food Image in the Bottom Right Panel.
40. Select the other four Food Images in the Bottom Right Panel while holding down *Ctrl*.
41. With each of the Food Images shown in the preceding screenshot still selected, paste the component as new in the Inspector.
42. Now, select each of the new Food Images and assign each to the parameter slot within their **Event Trigger** component.
43. Play the game and ensure that the correct image slot is changed when a food item is dropped into it.
44. Now that the drag and drop code is done, re-enable the `ShowHidePanels` script on the Main Camera and re-enable the Pause Panel.

That's it for the drag and drop code. Currently, the `Pause Panel` blocks the raycast on the items within the `Inventory Panel`, so you don't have to worry about disabling these events when the game is paused. However, if you end up changing the layout, you will want to do so by checking whether the `pauseUp` variable in `ShowHidePanels` is `false` before performing the tasks.

If you want to allow the objects to go back and forth (drag from both Panels and drop in both Panels), all you have to do is copy the appropriate component to the opposite Panels!

You might also want to make the repeated UI elements prefabs so that you can save yourself some time during development or instantiate them programmatically.

There are so many more examples I would love to cover in this chapter, but I can't make this chapter take up the entire page count of the book! You'll see more examples of how to use the Event System in the upcoming chapters, so don't worry; this isn't the last code example you will see.

Pan and zoom with mouse and multi-touch input

The last example I want to cover in this chapter is how to pan the camera with a two-finger touch and pinch to zoom. We will also implement a left-click pan and scroll wheel zoom so that you can easily test it on your computer (when you don't have multi-touch input). The following image shows what we will be implementing.



Figure 8.41: Demonstration of the pan and zoom code working

To implement a pan and zoom on the camera, complete the following steps:

1. So that I can actually see the effects of the pan and zoom, I want to populate some items in the scene. Let's start by creating a prefab that we will place multiple times in the background. Create a **Prefabs** folder in your project by right-clicking on the **Assets** folder and selecting **Create | Folder**. Name the new folder **Prefabs**.
2. Drag a sprite to the scene. I chose the first gem in the **foodSpriteSheet** called **food-SpriteSheet_1**.
3. Ensure that the **Transform** component of the sprite is positioned at the origin. If it is not, select the three dots in the top-right corner of the component and then select **Reset**.
4. Rename the sprite **Tile**.
5. Now, drag the sprite from the Hierarchy into your **Prefabs** folder. This will create a prefab called **Tile**.
6. We no longer need the prefab in the scene so go ahead and delete it.
7. Create an empty **GameObject** by selecting **+ | Create Empty** in the Hierarchy. We will use this to hold the code that instantiates our tiles.
8. Rename this empty **GameObject** to **Tile Maker**.
9. In the book's source files, you will find three scripts called **Tile.cs**, **TileMaker.cs**, and **CameraHandle.cs**. Import them into the **Scripts** folder of your project.
10. Attach the **Tile.cs** script to the **Tile** prefab. This script will be used to make any instantiated **Tile** prefab have a random sprite.
11. Since the **Tile.cs** script is not really essential to the example, I won't review the code, but I will point out that the **possibleSprites** list contains all the sprites that the tile can change to. Add all the sprites that look like gems to this list. You should see something like the following

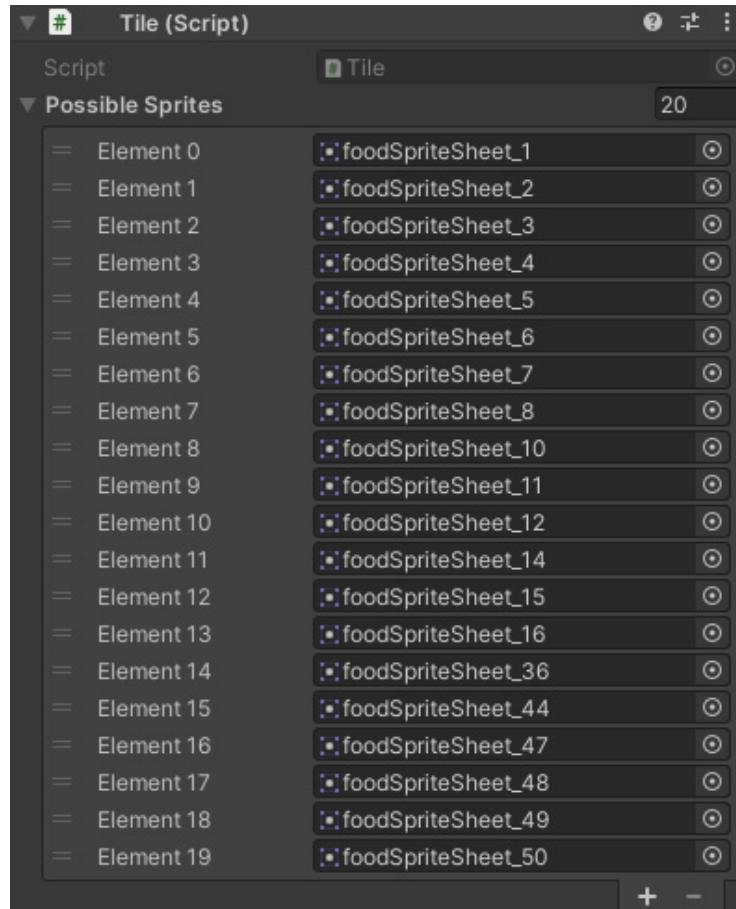
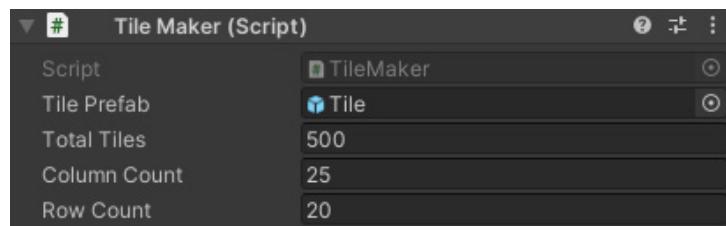


Figure 8.42: The possible sprites for the tile

12. Now attach the `TileMaker.cs` script to the `Tile Maker` GameObject.
13. Drag the `Tile` prefab into the **Tile Prefab** slot and set the other values as follows:

Figure 8.43: The `TileMaker.cs` script component

This will make a total of 500 tiles instantiate in 25 columns and 20 rows.

14. Play the game and you should see a bunch of random gems appear in your scene.



Figure 8.44: The gems in the scene

15. Now, let's hook up the pan and zoom script. Attach the CameraHandler.cs script to the Main Camera.
16. Before we review the code, let's add the correct variables to the component. Adjust the values to the following:

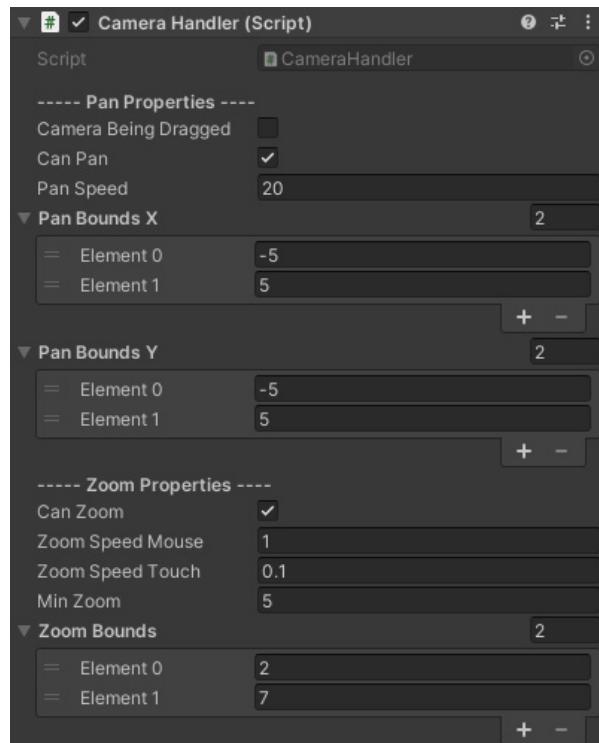


Figure 8.45: The CameraHandler.cs script component

The various properties add bounds to how far the camera can pan, how much it can zoom, and how quickly it pans and zooms.

- Now, let's review the code. The code handles pan and zoom in two ways: one is with mouse input and the other with touch input. You'll see that it also has special conditions for when a touch device is being played remotely via the Unity Editor.

A large portion of this code is vector math, and I will leave that for you to review on your own. The parts of this code I want to focus on are the parts relative to this chapter, specifically the parts involving `Input` and `Touch`. First, let's look at the `HandleMouse()` method. I've highlighted the relevant parts:

```
void HandleMouse() {
    if (Input.GetMouseButtonDown(0)) {
        lastPanPosition = Input.mousePosition;
    } else if (Input.GetMouseButton(0)) {
        PanCamera(Input.mousePosition);
    }

    float scroll = Input.GetAxis("Mouse ScrollWheel");
    ZoomCamera(scroll, zoomSpeedMouse);
}
```

Notice that it uses `Input.GetMouseButtonDown(0)` to see if the left mouse button is currently held down, `Input.GetMouseButton(0)` to see if the mouse button has been clicked, and `Input.mousePosition` to find where the mouse is located.

- Now, let's look at how input is handled with touch in the `HandleTouch()` method. First, it looks to see how many fingers are touching the screen with the following `switch` statement:

```
switch(Input.touchCount) {
```

`Input.touchCount` returns how many *fingers* are currently touching the screen.

- When a single finger is touching the screen, the camera can pan. It first must get the position of the finger. It does so with the following code:

```
Touch touch = Input.GetTouch(0);
if (touch.phase == TouchPhase.Began) {
    lastPanPosition = touch.position;
    panFingerId = touch.fingerId;
} else if (touch.fingerId == panFingerId && touch.phase == TouchPhase.Moved) {
    PanCamera(touch.position);
}
```

Once again, I have highlighted the relevant code.

20. When two fingers are touching the screen, it will get the position of the fingers with the following:

```
Vector2[] newPositions = new Vector2[] {Input.GetTouch(0).position, Input.GetTouch(1).position};
```

It stores the position of the first finger and the second finger in a Vector2 array. It then uses some fancy vector math to see whether the fingers are getting closer to each other or further away from each other, creating a pinch-to-zoom effect.

21. The next piece of code that I want to focus on is the following line within the PanCamera() method:

```
Vector3 offset = theCamera.ScreenToViewportPoint(lastPanPosition - newPanPosition);
```

This line of code is crucial as it converts the screen coordinates of your mouse or finger to that of the viewport.

22. If you try playing the game, the camera won't actually pan! We need to call the DragCamera() and StopCameraDrag() methods, so it will know when to get the inputs. We'll do this with Event Triggers on the Background Canvas. Add an **Event Trigger** component to the Background Canvas with the following events:

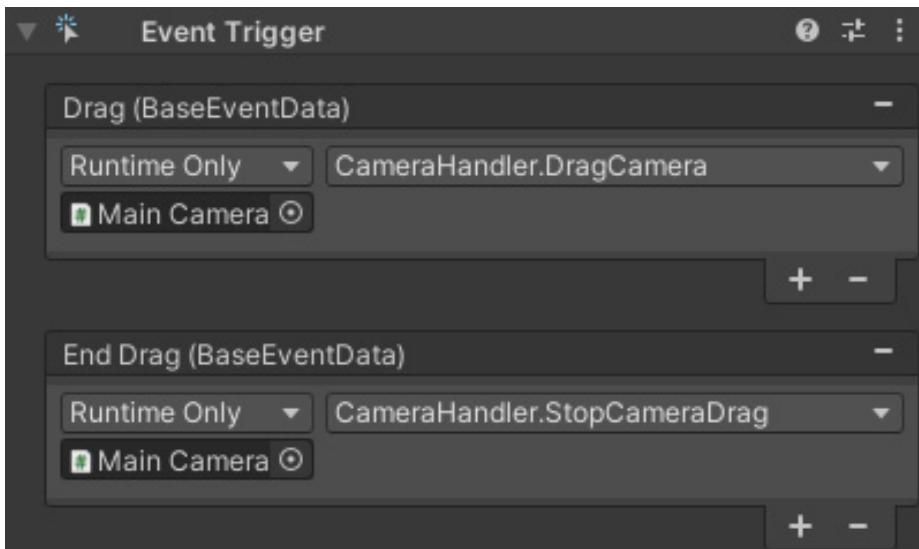


Figure 8.46: The Event Trigger on the Background Canvas

23. Now that you've reviewed the code, play the game again to see the pan and zoom functions in action. If you can, I recommend also plugging a mobile device into your computer and running the game via Unity Remote. You can view information about how to use Unity Remote here: <https://docs.unity3d.com/Manual/UnityRemote5.html>.

24. We don't want the game to pan and zoom when it's paused and the inventory Panel is up! So, let's update the `ShowHidePanels.cs` script to call the `TurnOffPanAndZoom()` and `TurnOnPanAndZoom()` methods, which toggle the `canPan` and `canZoom` Boolean variables.

Add the following variable to the `ShowHidePanels.cs` class:

```
CameraHandler cameraHandler;
```

25. Now, we need to initialize the `cameraHandler` variable. Add an `Awake()` method with the following code:

```
void Awake() { ¶     cameraHandler = GetComponent<CameraHandler>(); ¶}
```

26. Now, add the following to the `TogglePanel()` method. This will call the `TurnOffPanAndZoom()` method whenever the game is paused or showing the inventory and call `TurnOnPanAndZoom()` whenever neither of the Panels is visible:

```
if (inventoryUp || pauseUp)
{
    cameraHandler.TurnOffPanAndZoom();
}
else
{
    cameraHandler.TurnOnPanAndZoom();
}
```

You should now have a fully functional pan and zoom that are disabled when menus are visible!

This is almost the exact code I used in my game Barkeology, which you can find on the iOS app store: <https://apps.apple.com/us/app/barkeology/id1500348850> So, if you do not have the ability to test your code on your mobile device, but would like to see it in action, you can view it there.

While this marks the end of the chapter, we will continue to work in the Event System throughout this text, so you will see plenty more examples.

Summary

Now that we know how to utilize the Event System and program for UI elements, we can start making interactive and visual UI elements. We can also create UI that has its various properties change when events occur.

We covered a lot in this chapter! We discussed how to access the properties of UI elements and how to work with the Event System. We also discussed how to use the Input Module. Now, you can create UI that responds to user inputs as well as UI that responds to events within your game.

In the next chapter, we will look at the other input system provided by Unity: the New Input System (yeah, that's its actual name).

Part 3:

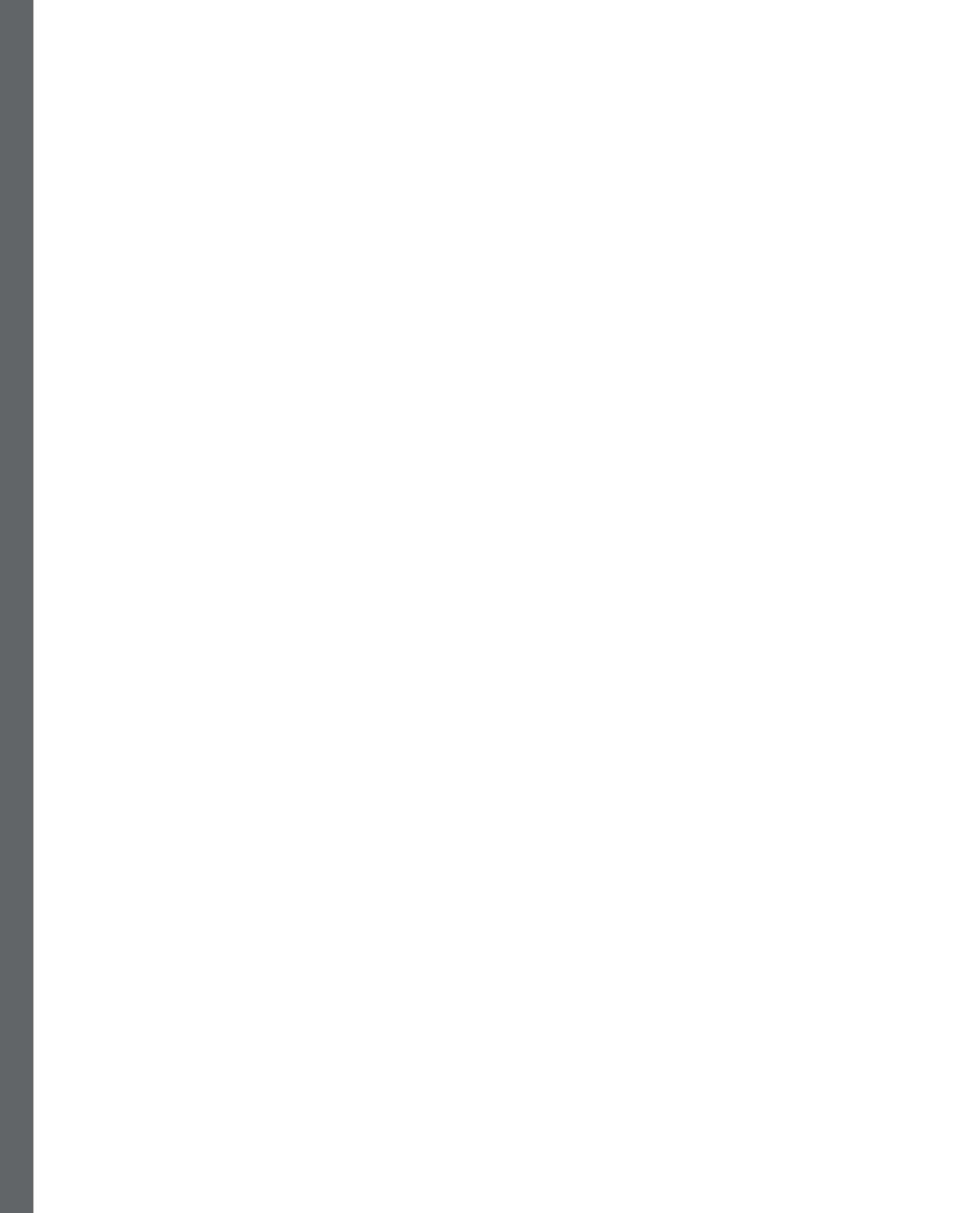
The Interactable

Unity UI Components

In this part, you will explore the individual interactable components provided by the uGUI system. How to lay out and program for buttons is covered. Additionally, the two ways in which you can display text, UI Text and Text-TextMeshPro are covered. You will learn how to work with UI Images and add various effects to them. How to use masks, scrollbars, and scroll views is discussed so that you can make expandable UI menus. And lastly, you'll learn how to use all the other Interactable UI Components provided by the Unity UI system.

This part has the following chapters:

- *Chapter 9, The UI Button Component*
- *Chapter 10, UI Text and Text-TextMeshPro*
- *Chapter 11, UI Images and Effects*
- *Chapter 12, Using Masks, Scrollbars, and Scroll Views*
- *Chapter 13, Other Interactable UI Components*



9

The UI Button Component

Buttons provided by Unity's UI system are graphical objects that preutilize the Event System we covered in the last chapter. When a Button is placed in a scene, it automatically has components added to it that allow the player to interact with it. This makes sense because the whole point of a Button is to interact with it. Let's explore how to add and utilize buttons in our games.

In this chapter, we will discuss the following topics:

- Creating UI Buttons and setting their properties
- How to set button transitions that make the button change appearance when it is highlighted, pressed, or disabled
- How to use invisible button zones to allow large tapping areas
- Navigating button selection on screen with the keyboard or joystick
- How to create an onscreen button that looks like it is physically being pressed
- Loading scenes with a button press
- Creating Button Transition Animations

Note

All the examples shown in the sections before the *Examples* section can be found within the Unity project provided in the code bundle. They can be found within the scene labeled Chapter9.

Each example image has a caption stating the example number within the scene.

In the scene, each example is on its own Canvas, and some of the Canvases are deactivated. To view an example on a deactivated Canvas, simply select the checkbox next to the Canvas' name in the Inspector. Each Canvas is also given its own Event System. This will cause errors if you have more than one Canvas activated at a time.

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2009>

UI Button

Buttons are UI objects that expect a click from the player. You can create a Button by selecting + | **UI | Button**. When you make a button, a **Button** object with a **Text** child will be placed in the scene. As with all other UI objects, if no Canvas or Event System is in the scene when you create the Button, a Canvas and Event System will be created for you, with the Canvas being a parent of your new Button:

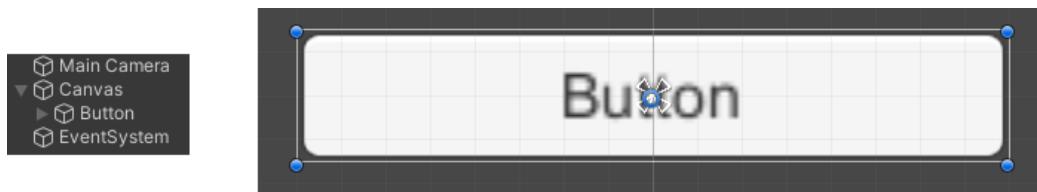


Figure 9.1: Adding a new UI Button to the scene

You can delete the child **Text** object if you do not want to have text displaying on your **Button**.

The **Button** object has three main components: **RectTransform** (like all other UI graphical objects), an **Image** component, and a **Button** component:

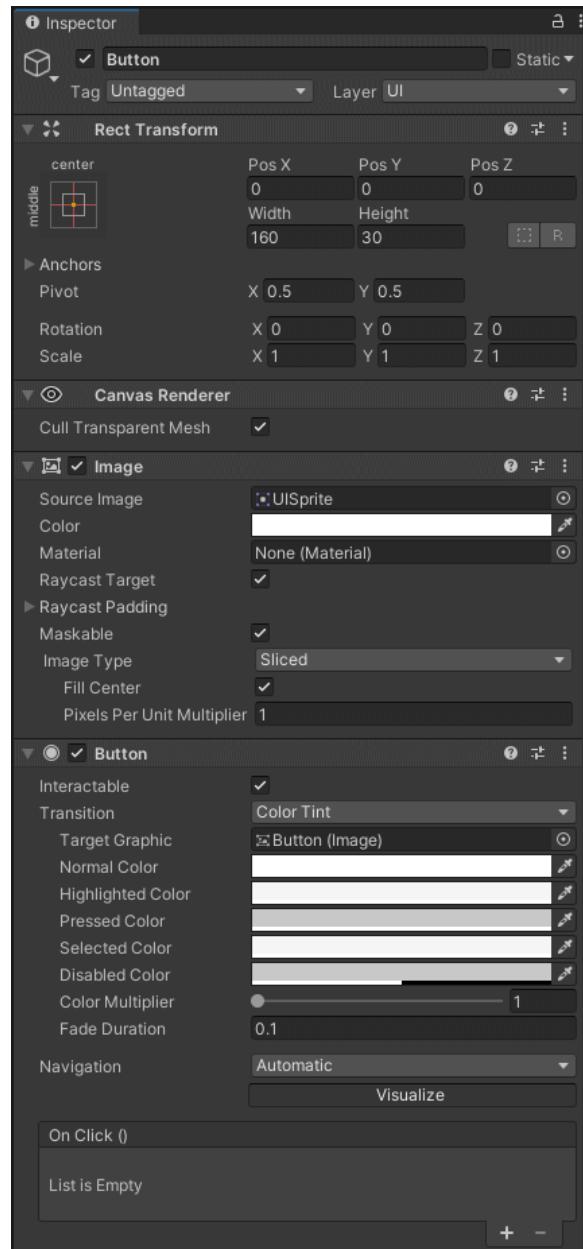


Figure 9.2: The components of a new UI Button

We'll discuss the **Image** component more thoroughly in the next chapter, but for now, just know that the **Image** component determines the look of the Button in its standard state.

The Button component

The **Button** component provides all the properties that allow the player to interact with the button and determine what the Button will do when the player attempts to interact with it.

The first property of the **Button** component is the **Interactable** property. This property determines whether the Button can or cannot be interacted with by accepting input from the player. This is turned on by default but can be turned off if you want to disable the button.

You'll see that the **Button** component already has an **On Click** Event attached to it. The **On Click** Event triggers when the player clicks and releases the mouse while hovering over the button. If the player clicks on the Button, moves the mouse outside of the Button's Rect Transform, and then releases the mouse, the Button's **On Click Event** will not register. You set the **On Click** Event the same way we set Events in *Chapter 8*.

Transitions

The second property of the **Button** component is the **Transition** property. The **Transition** property determines the way the button will visually react when the button is in different states. These different states are *not highlighted* (or normal), *highlighted*, *pressed*, *selected*, or *disabled*. These transitions are performed automatically and do not require coding.

There are four different types of transitions you can assign: **None**, **Color Tint**, **Sprite Swap**, and **Animation**.

None

Selecting **None** for the **Transition** type would mean that the button will not visually change for the different states.

Color Tint

The **Color Tint** transition type will make the button change color based on its state. You assign **Normal Color**, **Highlighted Color**, **Pressed Color**, **Selected Color**, and **Disabled Color**.

In the following example, you can see that the button changes to green when the mouse is hovering over it (hence, highlighting it) and turns red as the mouse is being pressed down on it:

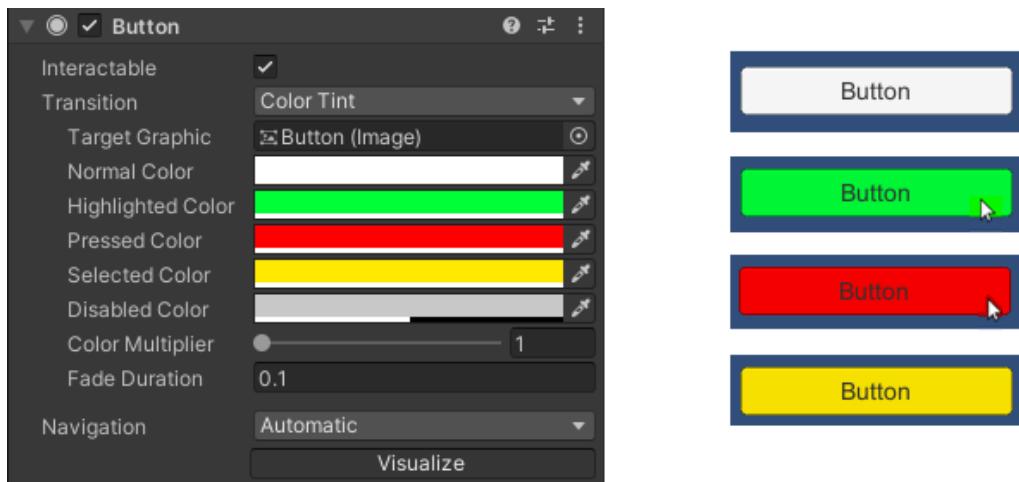


Figure 9.3: Color Swap Example in the Chapter9 scene

If you view the preceding example, you'll notice that the button turns yellow after it has been clicked. This is because it is *selected*. To return it to the normal color, click on any area outside of the button.

For a Button to enter the state that will give it its **Disabled Color**, the Button's **Interactable** property must be disabled. In the following screenshot, you will see how the button changes when the **Interactable** property is toggled on and off:

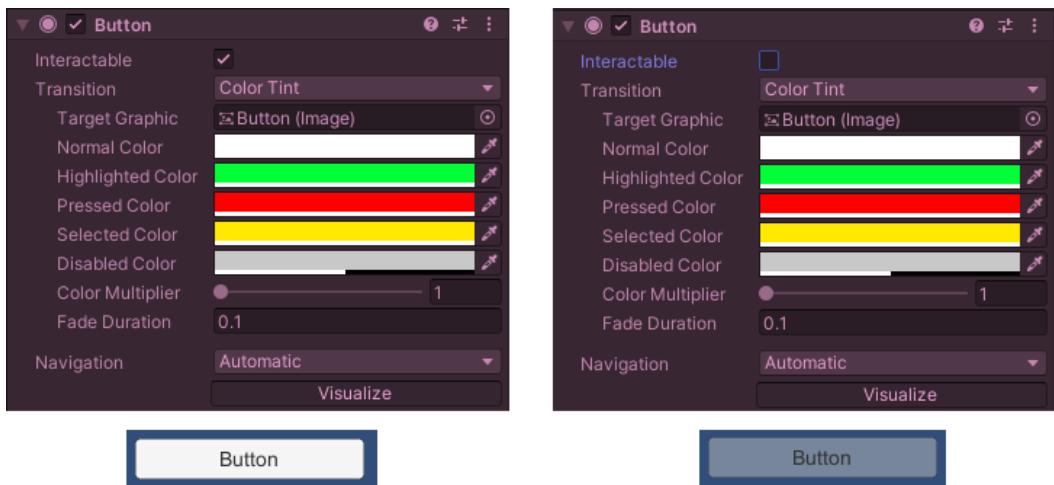


Figure 9.4: Disabled Button Example in the Chapter9 scene

You also can select the **Target Graphic**. This is the graphic that will receive the transitions. By default, it is assigned to the **Button** itself, so the **Button**'s image will change color when it is highlighted, pressed, or disabled. However, you can choose to make a secondary graphic the **Target Graphic**. This means the item assigned to the **Target Graphic** will change color based on the interaction of the button. In the following example, a secondary image is assigned as the **Target Graphic**. You'll see the button does not undergo transitions; instead, the star image undergoes transitions:

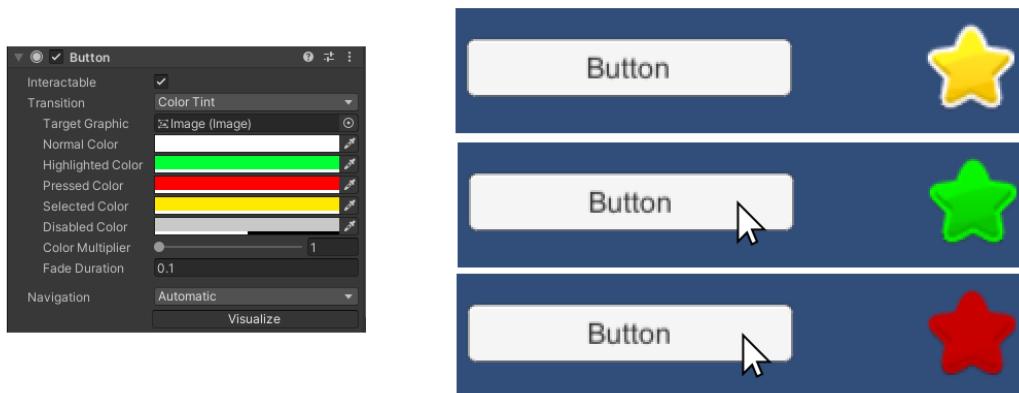


Figure 9.5: Target Graphic Example in the Chapter9 scene

It's important to note that these colors will tint the **Target Graphic**'s image. So, it essentially puts a color overlay on top of the **Target Graphic**. If the image of the **Target Graphic** is black, these tints will not appear to have any effect on the image. Note that the default **Normal Color** is white. Putting a white tint on an image does not change the color of the image.

The **Color Multiplier** property allows you to brighten up the colors or increase the alpha of the graphic. So, if the graphic has an alpha value that is less than 1, this property will increase the alpha of the graphic. This applies to all (including the normal) states.

The **Fade Duration** property is the time (in seconds) it takes to fade between the state colors.

Sprite Swap

The **Sprite Swap** transition type will make the button change to different images for different states.

You'll note that there is no property to assign a sprite for the normal state. This is because the normal state will just use the sprite assigned to the **Image** component.

The sprite sheet we imported in *Chapter 6*, has four button images that will be helpful in demonstrating the Sprite Swap Transition: the images labeled `uiElements_39`, `uiElements_40`, `uiElements_41`, and `uiElements_42` (as shown in the following screenshot):

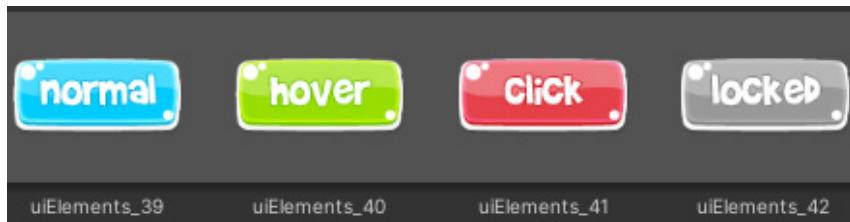


Figure 9.6: The sprites we will use to demonstrate button Sprite Swap

To have the button take on these images at the appropriate states, we simply need to assign uiElements_39 to the **Source Image** on the **Image** component, uiElements_40 to the **Highlighted Sprite**, uiElements_41 to the **Pressed Sprite**, uiElements_39 to the **Selected Sprite**, and uiElements_42 to the **Disabled Sprite**. We also need to delete the child **Text** object from the button:

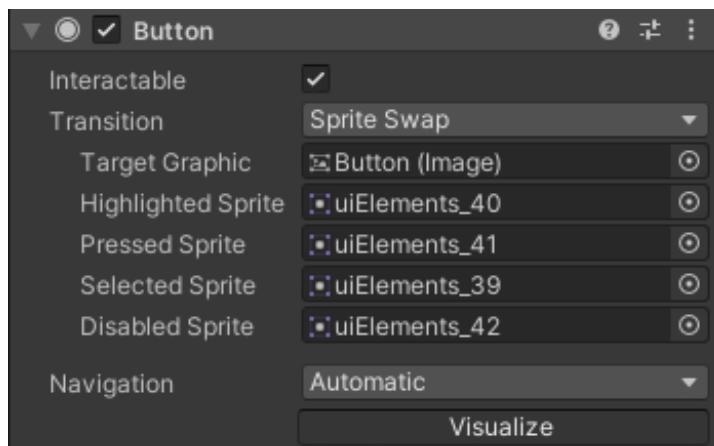


Figure 9.7: Sprite Swap Example in the Chapter9 scene

Remember, you can view the **Disabled Sprite** by deselecting **Interactable**.

A nice sprite swap animation that I always find appealing is applying an image of a button that appears down-pressed to the pressed image. For example, I took the button on the left and slightly edited it to create the button on the right. The change is slight, but I changed it by slightly moving down the top part of the button to make it look pressed:



Figure 9.8: The indented button animation sheet

It doesn't look like much of a difference when viewed side by side. However, when the left-hand image is used for **Source Image**, **Highlighted Sprite**, **Selected Sprite**, and **Disabled Sprite**, and the right-hand image is used for **Pressed Sprite**, the button transitions to show a very nice button-pressing animation:

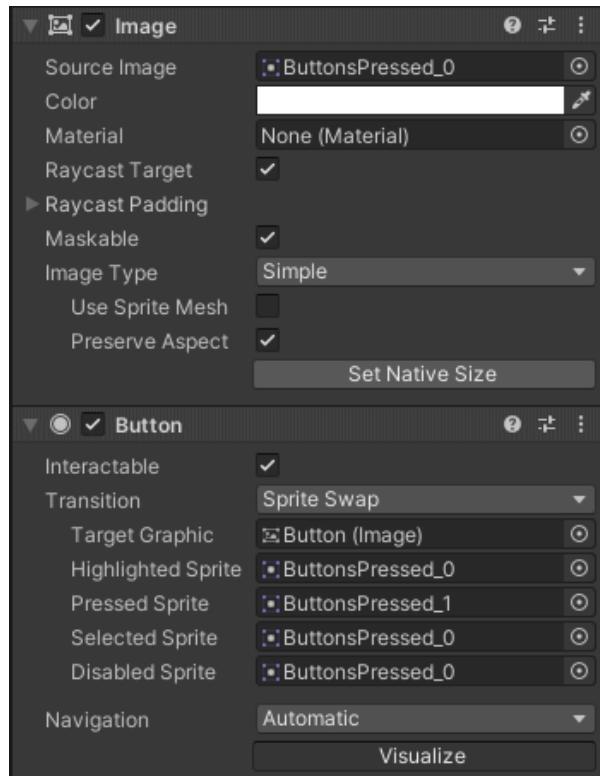


Figure 9.9: Pressed Button Example in the Chapter9 scene

To see this in action, view `Pressed Button Example` Canvas in `Chapter9Scene`; it really is quite satisfying to click.

In *Chapter 11*, we will explore how to create an image swap without the button transition property, for something like a mute/unmute button.

Animation

The **Animation** transition allows the button to animate in its various states.

Animation transition types require an **Animator** component attached to the Button. It can add a preexisting set of animations to a Button by dragging it onto the Button's Inspector, or you can make a whole new Animator Controller by selecting **Auto Generate Animation**. If you use a preexisting Animator Controller, you can simply assign the Animations to the individual states. However, if you

generate a new Animator Controller, you can select the state from the list of Clips in the **Animation** window and edit them from that window. An example of making a Button with animated transitions is provided in the *Examples* section of this chapter.

Navigation

Buttons have a **Navigation** property that determines the order in which they will be highlighted via keyboard or controller inputs:

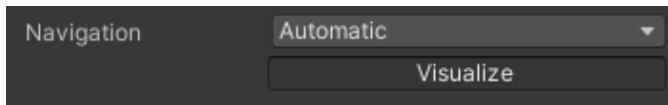


Figure 9.10: The Navigation property on the Button component

Each Button within the scene must have this property set if you want to navigate to all the Buttons. If you recall from *Chapter 8*, we discussed the **First Selected** property of the **Event System** component. If you have a Button assigned to **First Selected**, that Button will be highlighted when you load the scene. If you then navigate through the Buttons with the keyboard, the navigation will begin at the Button with the **First Selected** property. However, if you do not have a Button assigned as **First Selected**, navigation will not start until a button has been selected with the mouse. The next button selected is determined by the navigation option you have selected for the buttons.

There are five **Navigation** options: **None**, **Horizontal**, **Vertical**, **Automatic**, and **Explicit**.

Selecting **None** will disable all keyboard navigation to the specified Button. Remember that this is for the individual button, so if you want to disable all keyboard navigation, you must select **None** for all Buttons.

Horizontal and **Vertical** are pretty self-explanatory. If a Button has its **Navigation** property set to **Horizontal**, when it is selected, the next button selected will be chosen horizontally, meaning with the right and left arrows. **Vertical** works similarly; this represents the navigation *away* from that button, not *to* that button. So, if a button that has **Horizontal** set has its navigation property, you can still access that button from another with a vertical button.

Automatic will allow the **Button** to navigate both Horizontally and Vertically, as determined automatically by its position relative to the other buttons.

The **Visualize** button allows you to see a visual representation of the navigation setup. Each Button will be connected, with arrows demonstrating which Button will be selected after it. Each arrow begins on the side of the button to symbolize the directional arrow pressed, and it points at the next button that will be highlighted if that arrow is pressed. For example, if an arrow begins on a Button's right, that arrow symbolizes what Button will be selected next if the player presses right on the keyboard. The following example shows the visualization of five Buttons, all with their **Navigation** property set to **Automatic**:

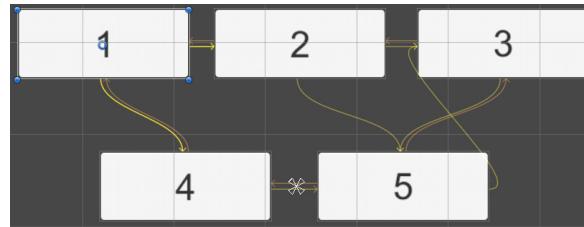


Figure 9.11: Navigation Example in the Chapter9 scene

In the preceding example, each Button has a **Color Tint Transition** property with the **Highlighted Color** and **Selected Color** assigned to green. The Button labeled **1** has been assigned as **First Selected** in the **Event System**:

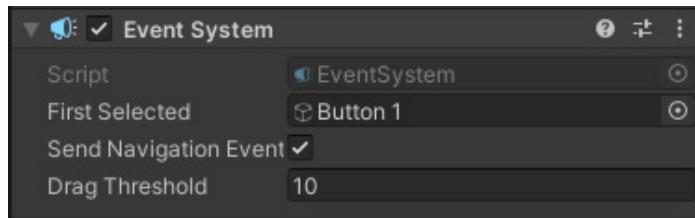


Figure 9.12: Event System with Button 1 assigned to First Selected

Therefore, when the scene begins playing, it will automatically be highlighted. Based on the visualized graph, if the right arrow key is selected on the keyboard after the scene loads, the **Button** labeled **2** will be selected:



Figure 9.13: Navigation Example in the Chapter9 scene

The last **Navigation** type is **Explicit** and allows for significantly better control. With this, you can explicitly define which button will be accessed with each individual keyboard press.

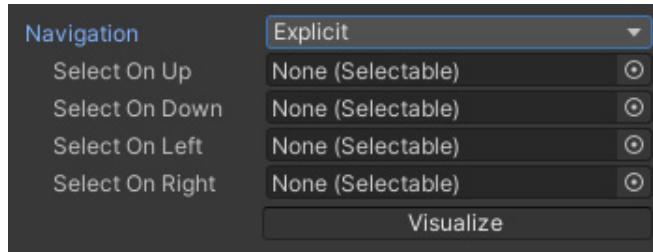


Figure 9.14: The properties of the Explicit Navigation type

Let's say that you wanted the player to cycle through the buttons in the order 1-2-3-4-5 and then loop back to 1. You want this to happen with either the up button or the right button. None of the previously mentioned navigation methods will allow that. However, you can achieve that with the **Explicit** Navigation type. The first step-by-step example of this chapter covers how to create an explicit button navigation map.

Invisible button zones

In *Chapter 2*, we discussed tapping zones on the screen. Often, in mobile games, tapping anywhere on the screen will cause an event. For example, many times when you select outside of a pop-up window, it will close. Other examples are when you can tap on the left or right side of the screen to move a character back and forth. Tapping on areas of the screen may not seem like a button implementation, but it actually is! The buttons are just invisible.

Let's explore the first scenario. If you review the `Close Panel Example` GameObject of the `Chapter9` scene, you will see a Panel that appears when an info button is pressed, closes when the close button is pressed, and also closes when the area outside of the Panel is pressed. This is accomplished by putting a large, invisible button behind the Panel. For it to work appropriately, it needs to be in front of the info button (blocking raycast to it).

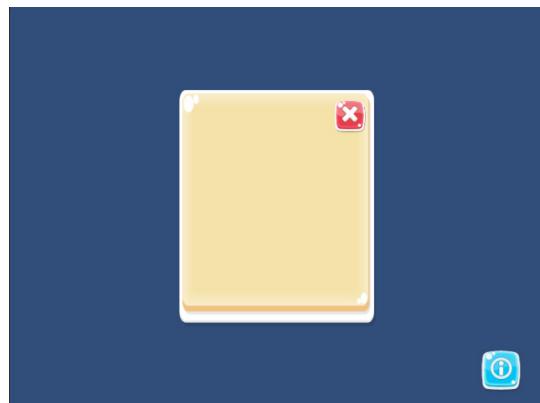


Figure 9.15: Close Panel Example in the Chapter9 scene

It's a good design choice to include the close button, even if the background area will dismiss the Panel. Many people do not intuitively consider tapping outside a Panel to be an action that will close it and will spend time looking for the close button if you do not provide it.

Now let's explore the second scenario, where the two sides of the screen cause different actions. In the `Tap Zone Example GameObject`, tapping the left side moves the candy cane to the left, and tapping the right side moves the candy cane to the right. Once again, large invisible buttons are being utilized.

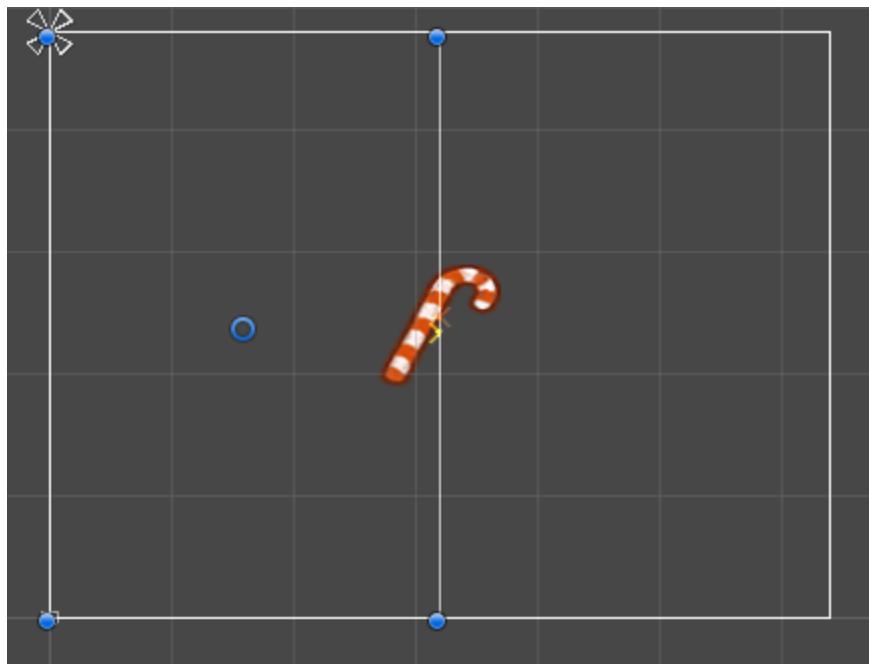


Figure 9.16: Tap Zone Example in the Chapter9 scene

When using these large invisible buttons, it is very important that you consider how raycast will be affected by them, even though they are invisible. They will block things behind them!

Examples

For the first three examples in this chapter, we will momentarily step away from the scene we have been working on to build a new scene that will allow us to experiment with button navigation and scene loading. We'll then pick up where we left off with our scene from *Chapter 8*, to add some buttons to our scene.

Navigating through Buttons and using First Selected

We'll build out a faux start screen that appears as follows:

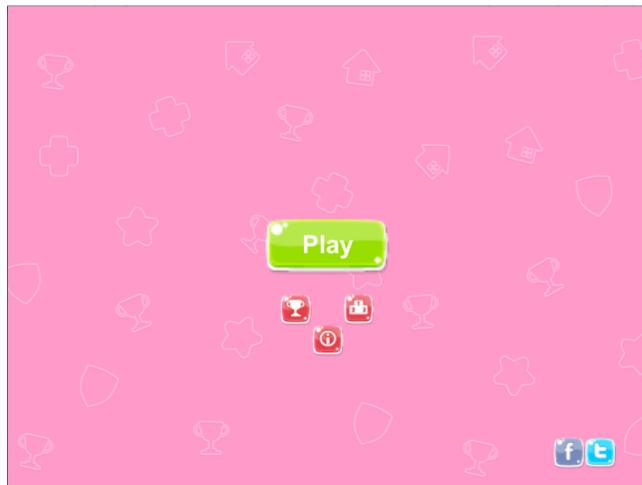


Figure 9.17: The start screen scene we will build

Most of these buttons will be dummy buttons, but we will set up the **Play** button in the next example to load the scene we have been working on.

To give us the ability to experiment with button navigation, we'll assign an **Explicit** navigation scheme to it so that we can cycle through the buttons with the following pattern:

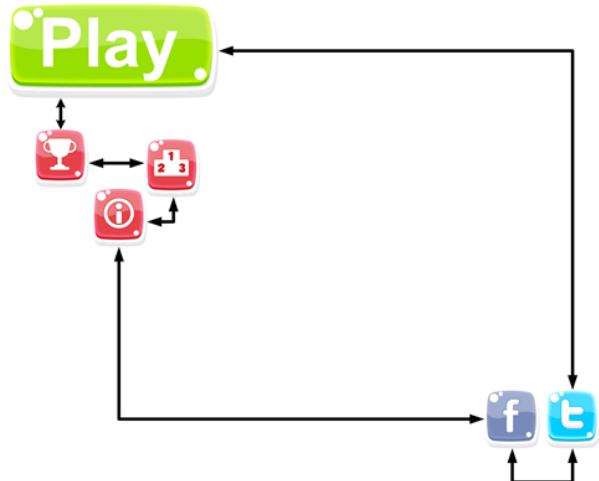


Figure 9.18: The button navigation map

First, the **Play** button will be selected. Pressing the *down* key on the keyboard continuously will result in the following selection path:

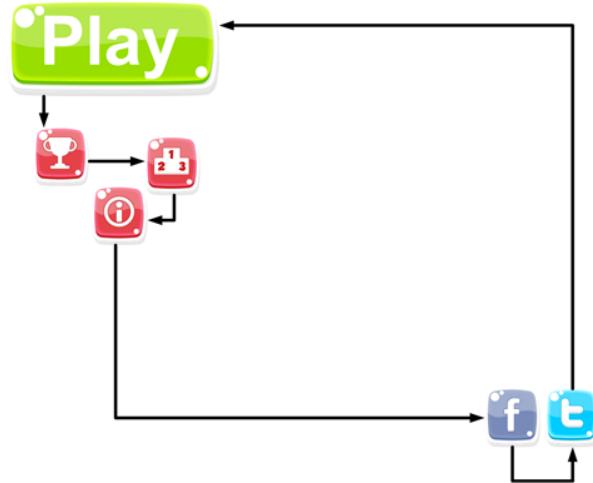


Figure 9.19: The button navigation flow with the down arrow

Pressing the *up* button continuously will result in the following path:

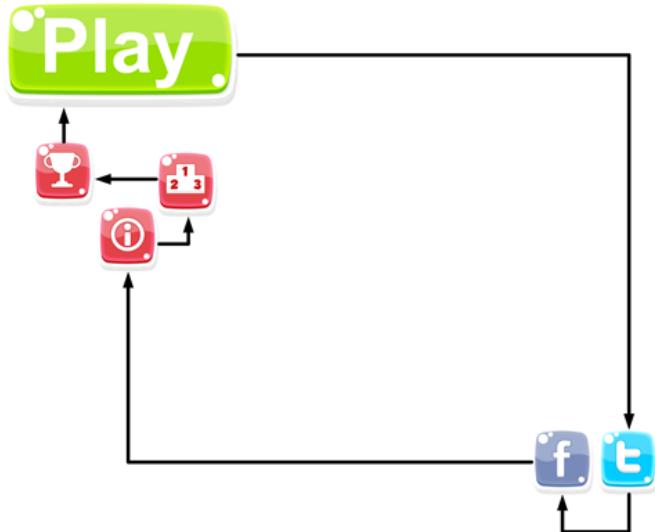


Figure 9.20: The button navigation flow with the up arrow

Next, we'll learn how to lay out the buttons.

Laying out the Buttons

Let's start by creating a new scene and laying out the buttons.

To make a faux start screen as shown in *Figure 9.17*, complete the following steps:

1. Create a new empty scene and name it Chapter9-Examples-StartScreen. Open the new scene.
2. To give this scene the same background as the scenes we've made in the last few chapters, we can create another **Background Canvas**, but it will be easier to just copy it from one of the other scenes. We'll do this by having our new scene and one of our old scenes open in the Hierarchy at the same time.

From the **Project** folder view, drag the Chapter8-Examples scene to the **Hierarchy**. You should now see the following:

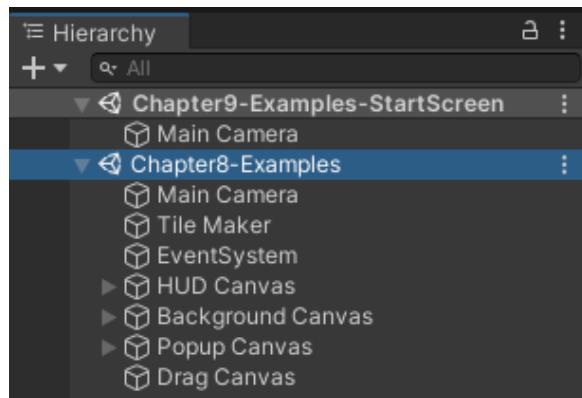


Figure 9.21: Loading the two scenes simultaneously

3. Select **Background Canvas** and press *Ctrl + D* to duplicate it. Now, drag the duplicate, labeled **Background Canvas (1)**, to the Chapter9-Examples-StartScreen scene:

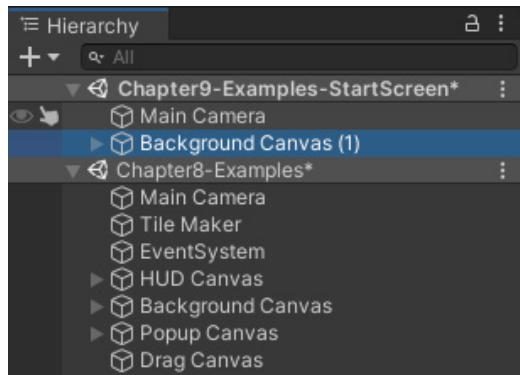


Figure 9.22: Duplicating Background Canvas

4. We can now close the Chapter8-Examples scene, as we no longer need it. To do so, select the three dots on the right of the Chapter8-Examples scene, and select **Remove Scene**. Just in case you accidentally deleted something, select **Don't Save** when prompted:

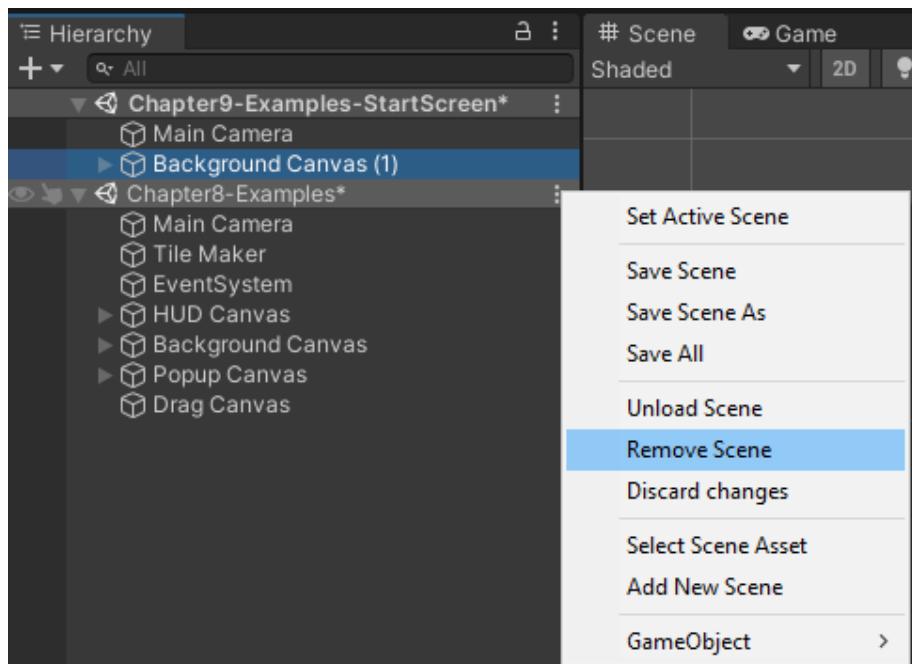


Figure 9.23: Removing the Scene

You should now have only Chapter9-Examples-StartScreen in the **Hierarchy**, and the **Background Canvas** should be visible in the scene.

5. Rename **Background Canvas (1)** to **Background Canvas** and save the scene.

Note

You'll note that by doing this, we have a **Canvas** in the scene without an **Event System**. That's okay, though; once we add new UI elements, an **Event System** will be added in for us.

6. Before we can proceed, we should take care of a warning message that pops up in the Console. Since we copied **Background Canvas** from another scene, it is trying to access the camera from the other scene and can't find it.

A warning message will also appear on the **Canvas** component of the **Background Canvas**:

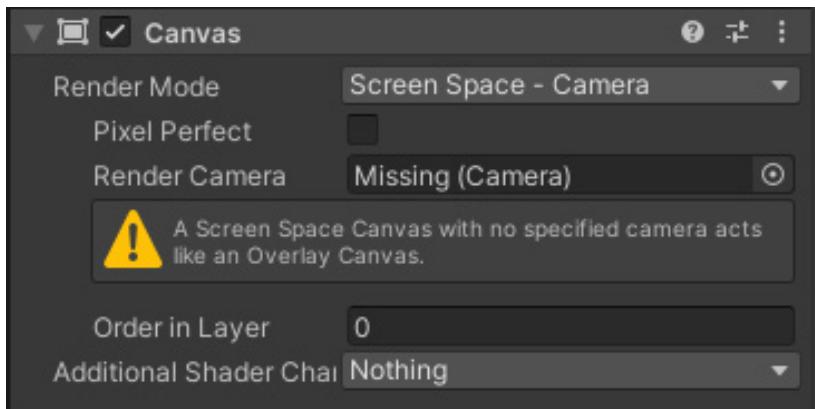


Figure 9.24: The Canvas component error message

To fix this, simply drag the **Main Camera** from the current scene to the **Render Camera** slot and set the **Sorting Layer** to **Background**.

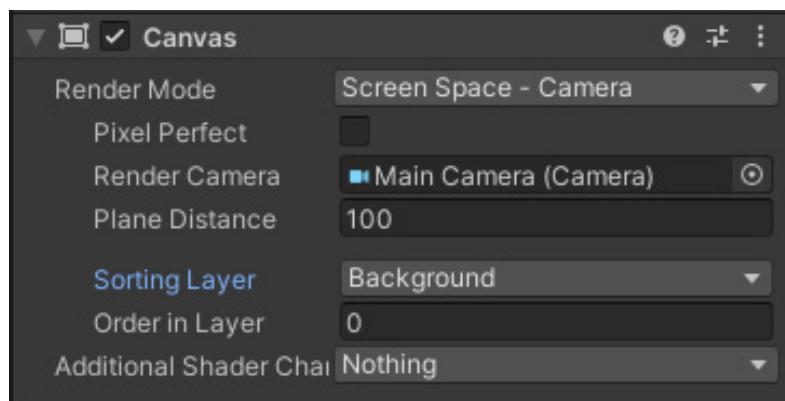


Figure 9.25: The Canvas component with the Main Camera assigned

7. Now, let's add the **Play** button. Create a new UI Canvas to place our Buttons on and name it **Button Canvas**. Note that once you create the **Button Canvas**, an **Event System** **GameObject** will be created for you.
8. Create a new UI Button (+ | UI | **Button**) as a child of **Button Canvas** and give it the following **Rect Transform** and **Image** component properties:

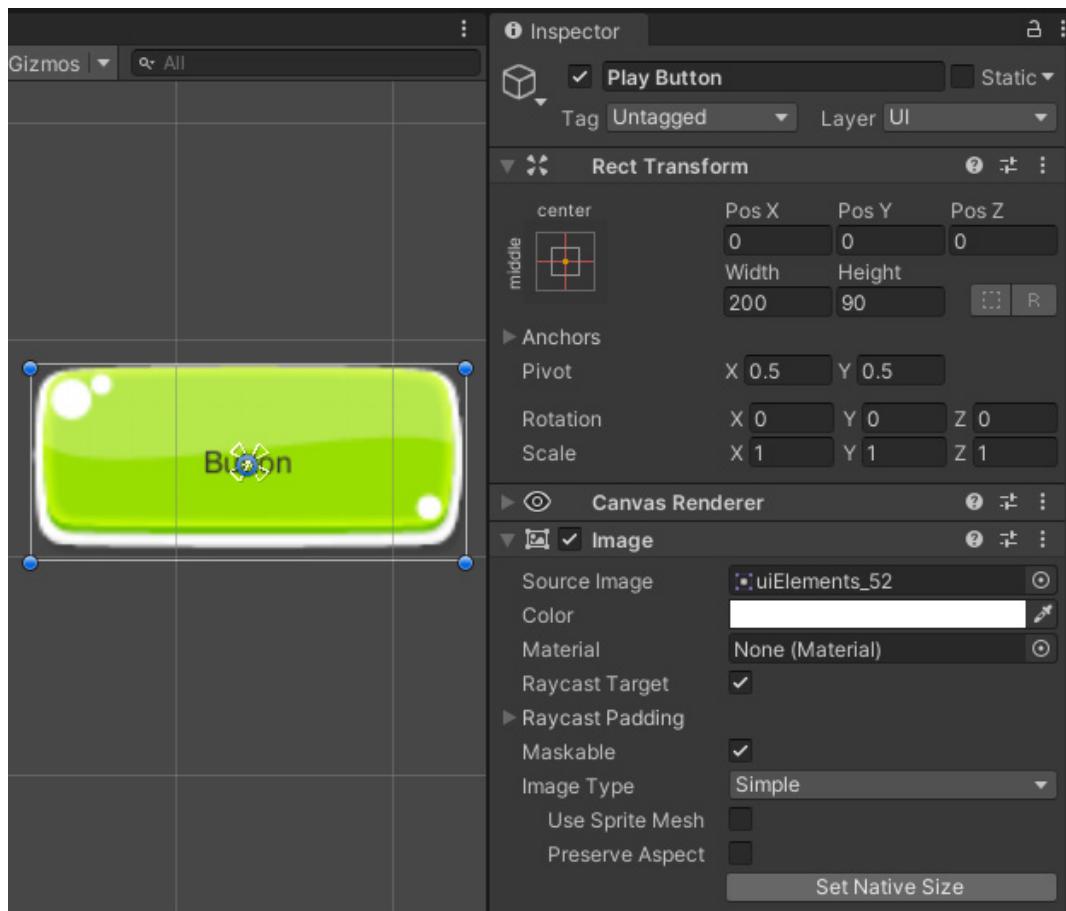


Figure 9.26: The properties on the Play Button

9. Now, select the child **Text** object of the **Play** Button and set its **Rect Transform** and **Text** component properties as such:

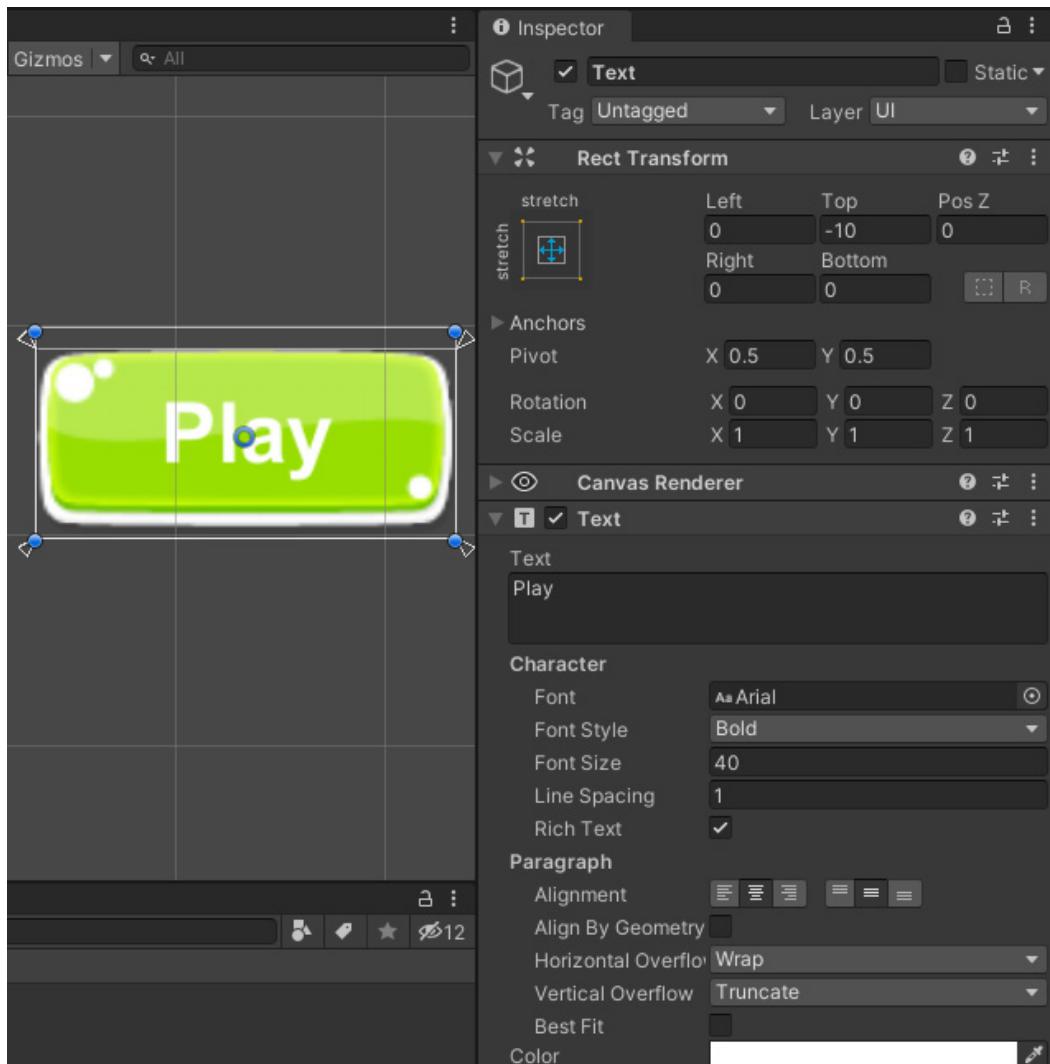


Figure 9.27: The properties on the Play Button's text

10. Now, let's create an Achievement Button, a Leaderboard Button, and an Info Button so they appear as follows:

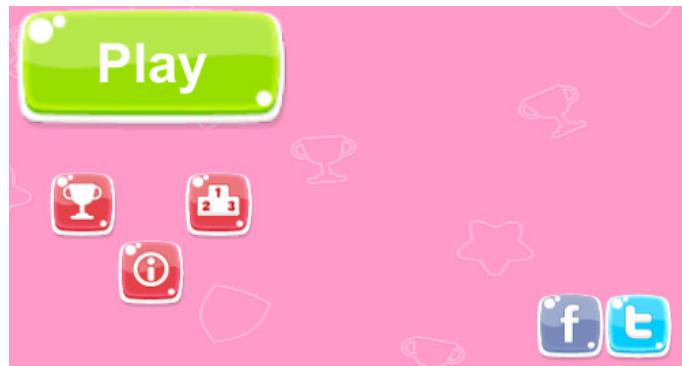


Figure 9.28: The layout of the start screen buttons

To achieve the preceding layout, use the following properties and make sure to remove their child Text objects:

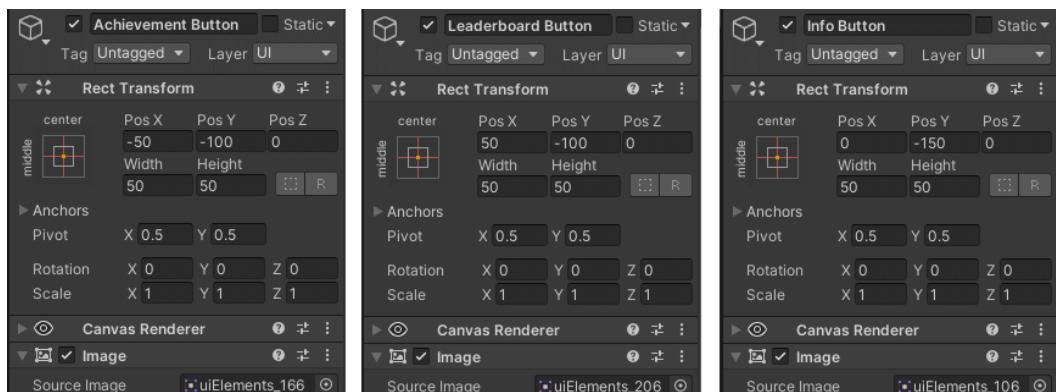


Figure 9.29: The properties of the three buttons

11. Now all that is left to do to set up our scene's layout is to create the Facebook Button and Twitter Button in the bottom-right corner of the scene. (Ignore the fact that the logos are extremely outdated.) To achieve the layout in *Figure 9.26*, create two buttons with the following properties:

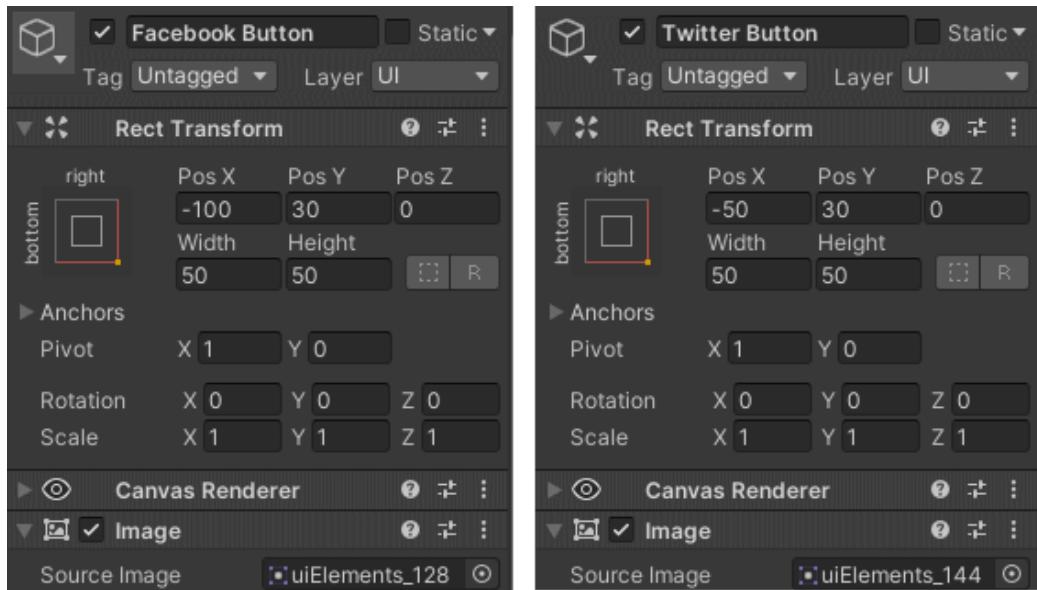


Figure 9.30: The properties on the two buttons

Your scene should be correctly laid out now, so let's work on setting up the navigation.

Setting the explicit navigation and First Selected

If you select the **Visualize** button on any of your Button's **Button** components, you should see the following navigation path:



Figure 9.31: The automatic button navigation visualization

This navigation setup allows significantly more navigation range than the setup I described at the beginning of this example. This is because each button has its navigation set to **Automatic** by default, and **Automatic** allows multidirectional navigation. Let's make our navigation a bit more linear and make our Play Button the **First Selected** Button in our **Event System**.

To set up the navigation described at the beginning of this example, complete the following steps:

1. Select the **Event System** in the Hierarchy and assign the Play Button to the **First Selected** slot by drag and drop:

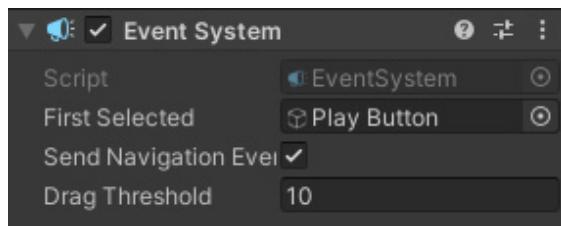


Figure 9.32: The Event System with the Play Button as First Selected

Now, when we start cycling through our buttons, our navigation will begin at the Play Button. Also, if we were to hit the *Enter* key when this scene loads, the Play Button will automatically be executed.

2. Now, let's set all the Buttons to have an **Explicit Navigation** type. Select all six of the buttons in the **Hierarchy** list.
3. With all selected, change the **Navigation** type to **Explicit** from the dropdown menu. Now, each Button should have the following settings in its **Button** component.
4. To make it easier to see our navigation, let's change the **Selected Colors** property on each of our Buttons. With all the Buttons still selected, set **Selected Color** to dark red. It's not very attractive, but it will make it easier for us to see whether our buttons are being selected. The **Button** component on all of your buttons should now look as follows:

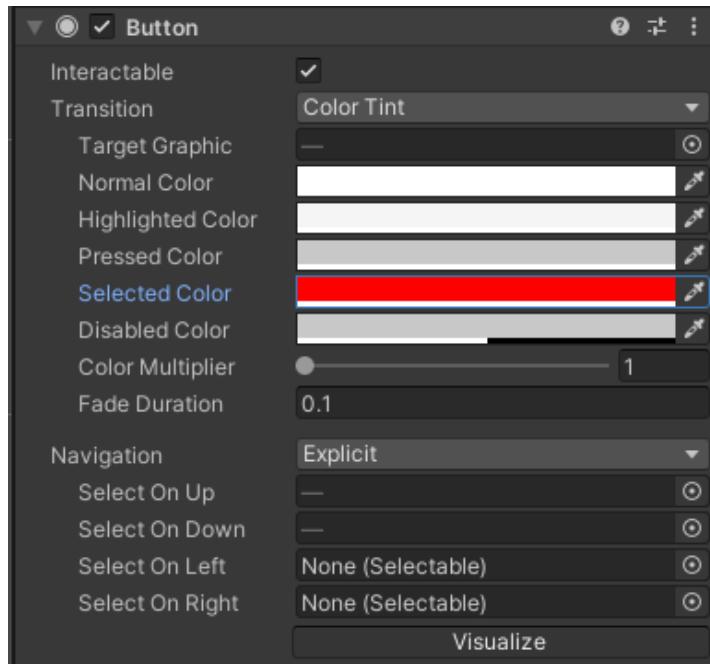


Figure 9.33: The Button component with Color Tint transitions

If you play the game, you should see the `Play` Button colored red, symbolizing that it is selected (since we set it as **First Selected** in Step 1):

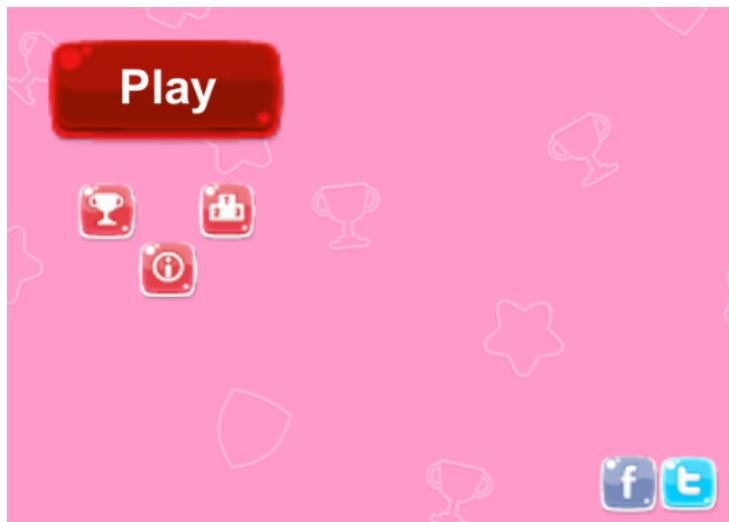


Figure 9.34: Play Button selected and colored red

- Now, we can explicitly (hence the name) set the Buttons that each individual Button is to navigate to by dragging and dropping them into the appropriate slots. Let's set the **Navigation** for the Play Button first since it will be the first button selected.

According to *Figures 9.18 through 9.20*, the Play Button should navigate to Twitter Button when the *up* key is pressed and Achievements Button when the *down* key is pressed. So, drag and drop those Buttons into the slots labeled **Select On Up** and **Select On Down**, respectively, from the **Hierarchy**.

If you have the **Visualize** button selected, as you are dragging the Buttons into their slots, you should see the navigation visualization starting to build out.



Figure 9.35: The Navigation properties of the Play Button

Note

Remember that if an arrow begins on top of a Button, that arrow symbolizes where navigation will go if the *up* key is pressed, and if the arrow begins on the bottom of a Button, it symbolizes where navigation will go if the *down* key is pressed.

- Play the game to check and see whether it works. Press the *up* key and you should see the Twitter Button turn red, indicating it is selected.

To see the Achievement Button become selected, you actually have to stop playing the game and replay, because we have not set up the navigation for the Twitter Button to go back to the Play Button. So, stop the game, press Play again, then press the *down* key, and you should see the Achievement Button highlight red.

Note

Instead of restarting the game, you can also highlight the Play Button with your mouse so that you can then navigate to the Achievement Button.

- Once you set up the navigation for one Button, the rest aren't too difficult, albeit tedious. Use the following chart to help you set up the rest of the Buttons:

Button	Select On Up	Select On Down
Play Button	Twitter Button	Achievement Button
Achievement Button	Play Button	Leaderboard Button
Leaderboard Button	Achievement Button	Info Button
Info Button	LeaderBoard Button	Facebook Button
Facebook Button	Info Button	Twitter Button
Twitter Button	Facebook Button	Play Button

Table 9.1: The Select On Up and Select On Down assignments for each Button

When you are done, your navigation visualization should look like the following:

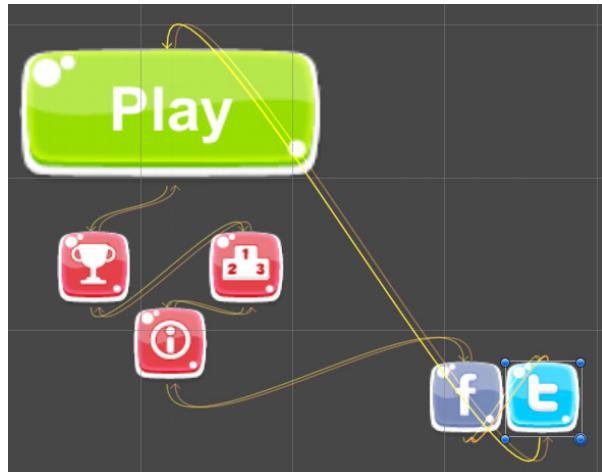


Figure 9.36: The finalized Navigation flow visualization

If you play your game, you should be able to easily cycle through the Buttons using the arrow keys.

I recommend setting all the buttons to have **Horizontal** or **Vertical Navigation** patterns and seeing how they differ from what we have created so that you can see that this pattern is not attainable with a predefined pattern applied to all the Buttons.

Loading scenes with Button presses

Now that we have our start screen laid out, let's hook up the Play Button to play our game. First, duplicate the scene you created in *Chapter 8*, called *Chapter8-Examples*, using *Ctrl + D*. The new scene should be called *Chapter9-Examples*. Our goal is to have the Play Button in *Chapter9-Examples-StartScreen* load the *Chapter9-Examples* scene.

To make the Play Button in the *Chapter9-Examples-StartScreen* scene, load up the *Chapter9-Examples* scene and complete the following steps:

1. To transition between scenes within Unity, you must first ensure that they are each listed within the **Scenes In Build** list in the **Build Settings**. Select **File | Build Settings** (or *Ctrl + Shift + B*). The following should be visible:

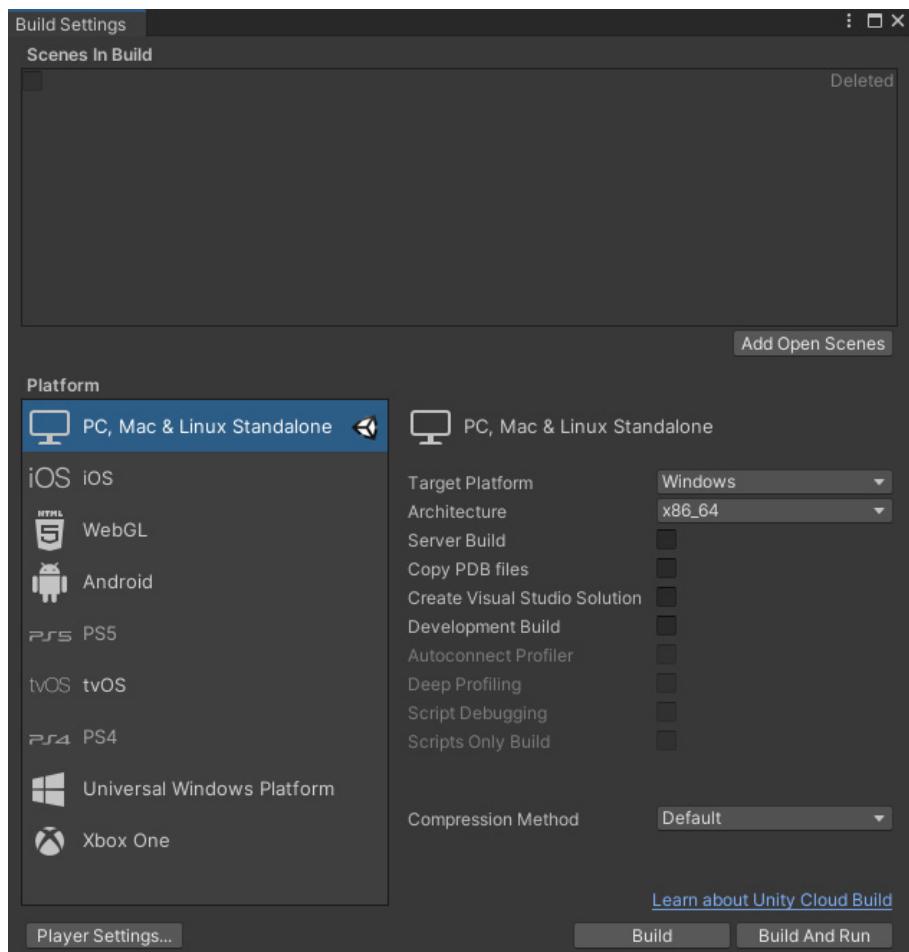


Figure 9.37: The Build Settings with no scenes in the build

- From the **Project** folder, drag and drop the Chapter9-Examples-StartScreen and Chapter9-Examples scenes into the list:

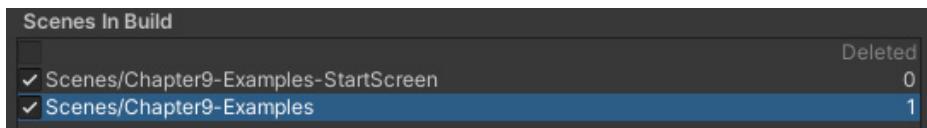


Figure 9.38: Adding the scenes to the build

The order in which scenes appear in this list does not matter, except for the first scene (the one listed as scene **0**). The first scene in the list should be the scene you want to load when the game loads. Therefore, it makes sense for us to put Chapter9-Examples-StartScreen in position **0**.

- Now that our scenes are in the **Build Settings**, we can write a script that will allow us to navigate between them. Right-click on the **Scripts** folder in your **Project** view and select **Create | C# Script**. Name the new script **LevelLoader**. Open the new **LevelLoader** script and replace the code with the following:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class LevelLoader : MonoBehaviour {

    public string sceneToLoad = "";

    public void LoadTheLevel() {
        SceneManager.LoadScene(sceneToLoad);
    }
}
```

The preceding code contains a single function—**LoadTheLevel()**. This function calls the **LoadScene** method in the **SceneManager** class. We will load the **sceneToLoad**, which is a string we will specify in the **Inspector**. Note that the **UnityEngine.SceneManagement** namespace must be included with the following line at the top of the script:

```
using UnityEngine.SceneManagement;
```

- If you still do not have the Chapter9-Examples-StartScreen scene open, open it again. Select the **Play** Button and drag and drop the **LevelLoader** script onto its **Inspector**.
- Now, within the **Scene To Load** slot, type Chapter9-Examples. You don't need to put it in quotes; since the **sceneToLoad** variable is a string, Chapter9-Examples is assumed to be a string without you needing to put it in quotes:



Figure 9.39: The LevelLoader.cs script component

- Now all that is left is to hook up the Button's click event. Select the + sign at the bottom of the **OnClick()** Event list of the **Button** component to add a new event. The script we want to access, `LevelLoader.cs`, is on the `Play` Button, so drag the `Play` Button into the object slot. Now, from the function dropdown menu, select `LevelLoader | LoadTheLevel`.

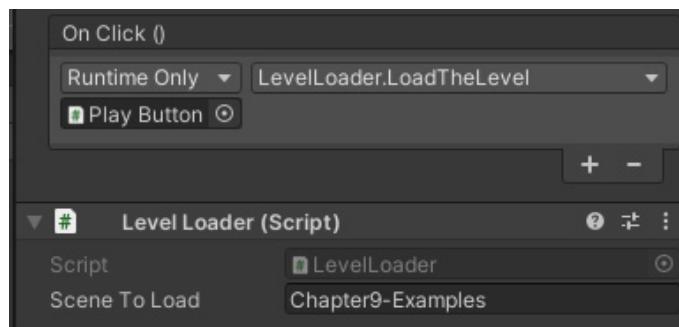


Figure 9.40: The OnClick event hooked up

That's it! Your `Play` Button should now navigate to the `Chapter9-Examples` scene when clicked on or when you press *Enter* when it is highlighted (as it is at the beginning or with your keyboard navigation).

Button Animation Transitions

Often buttons are animated as a way to draw your attention to them. Let's give the `Play` Button an Animation Transition so that when it is in its normal state it will pulsate to draw the attention of the player to it.

To add button Animation Transition on the `Play` Button, complete the following steps:

- Select the `Play` Button and change its **Transition** type to **Animation**.
- We do not have an Animator Controller prebuilt, so select **Auto Generate Animation** to create a new one.
- A window will pop up asking you to save the newly created Animator Controller. Create a new folder in the `Assets` folder, called `Animations`, and save the new Animator Controller as `Play Button` to the folder.

You should now see the new Animator Controller in the new Animations folder:

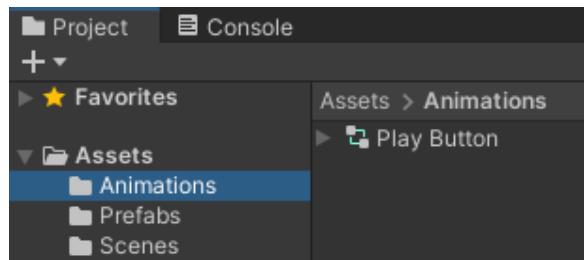


Figure 9.41: The Play Button animator in the Project

You will also see the new **Animator** component on the Play Button:

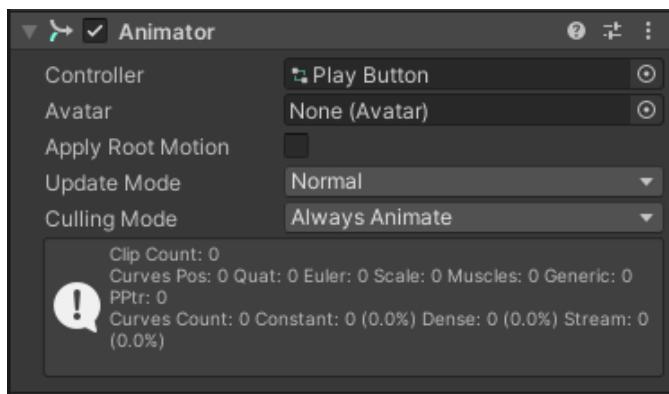


Figure 9.42: The Play Button Animator component

4. Open the **Animation** window by selecting **Window | Animation**. If you want to dock this new window, dock it somewhere so that you can still see the Scene and Game views when it is up.
5. Select the Play Button and open the **Animation** window. The various animation clips associated with the transition states will be visible in the animation clip dropdown menu:

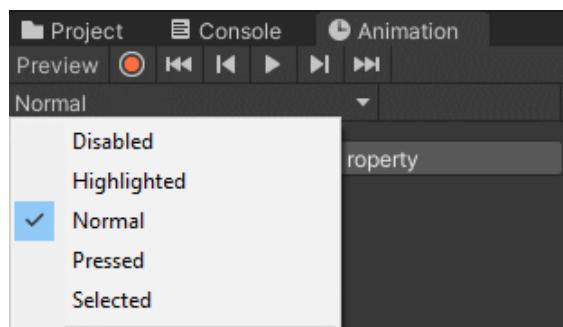


Figure 9.43: The various animations on the Play Button

We want to edit the animation for the **Normal** transition state, so make sure it is selected (**Normal** is the animation selected by default).

- To make the button look like its pulsating, we want to affect its scale. Select **Add Property** | **Rect Transform** and then hit the + icon next to **Scale**:

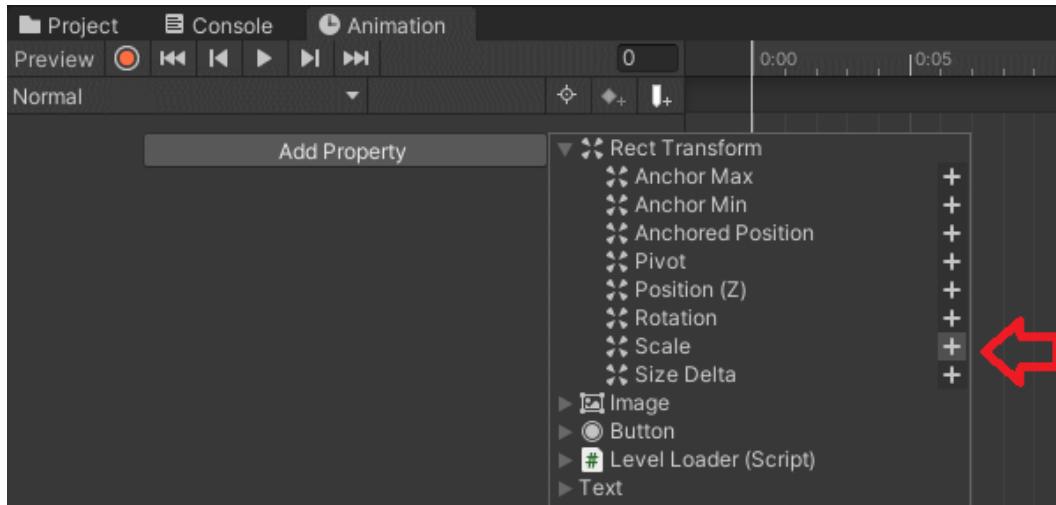


Figure 9.44: Adding the Scale property to the Play Button

The **Scale** property should now be showing up in the **Animation** timeline:

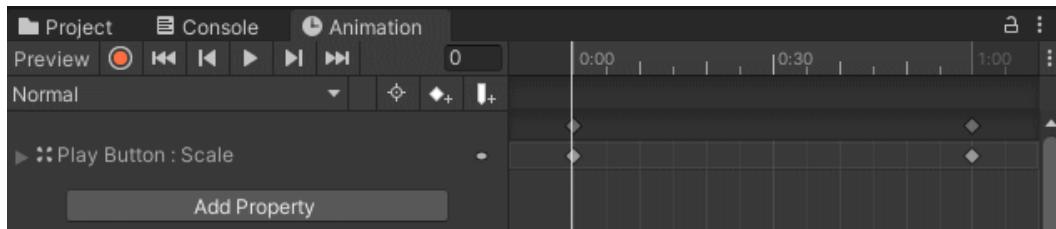


Figure 9.45: The Play Button Scale timeline

- To achieve the pulsation with scaling, we want the button to start at its normal size, get big, and then go back to its normal size. The diamonds that appear on the timeline are known as keyframes. To do what I just described, we need one more keyframe, right in the center of the timeline. Click on the top of the timeline (where the numbers appear) to move the timeline to the **0:30** mark. Then, select the **Add keyframe** button to add a new keyframe:

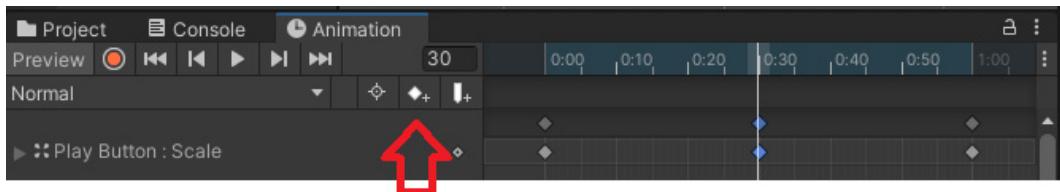


Figure 9.46: Adding an extra keyframe to the timeline

8. Expand the **Scale** property by selecting the arrow to its left. You will see that if you select any of the keyframes, the number **1** appears next to all three scaling directions. This indicates that the scale at that frame is 100% (or its normal scale):

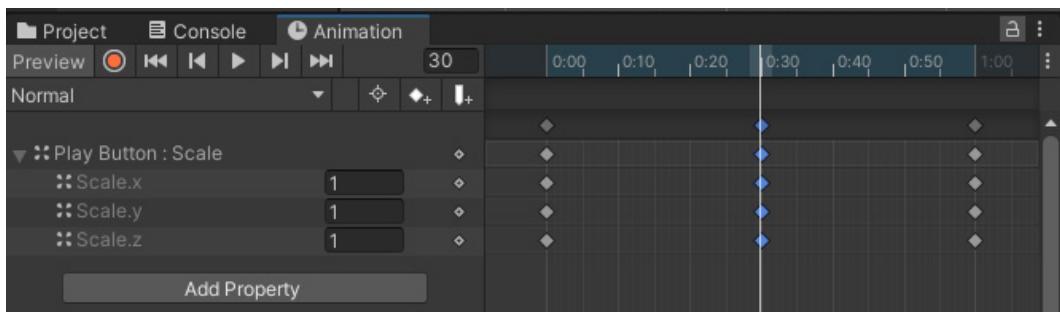


Figure 9.47: Expanding the Scale property of the Play Button

9. Select the keyframe in the center. Change the number **1** on **Scale.x** and **Scale.y** to **1.2** by clicking on the number **1** and typing the new value. You can press the Play button in the animation window to preview your animation. You will see that the button now pulsates in the scene.

Your button should now pulsate when you play the game (and will not turn red any longer). Note that it does not pulsate immediately, because the button will only pulsate when it is in its normal state. Since we have it set to **First Selected**, it is selected on start. To remove the selection, simply click anywhere in your scene outside of the button or navigate away using your keyboard. Once the button is no longer selected, it should begin pulsating.

10. Let's bring back the red selection since it made it easy to tell when the button was selected (even if it was unattractive). From the animation list dropdown menu, select the **Selected** animation.

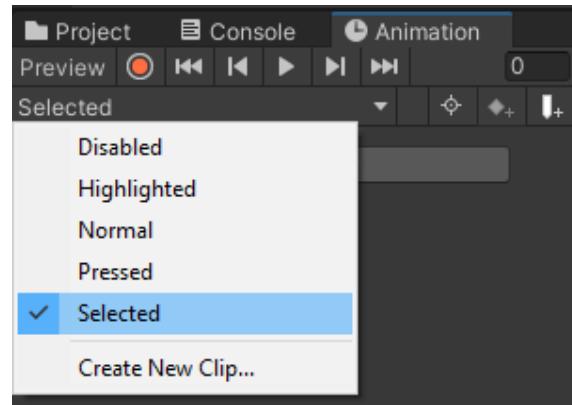


Figure 9.48: Selecting the Selected animation for the Play Button

11. Select the **Add Property** button, then select **Image | Color**.

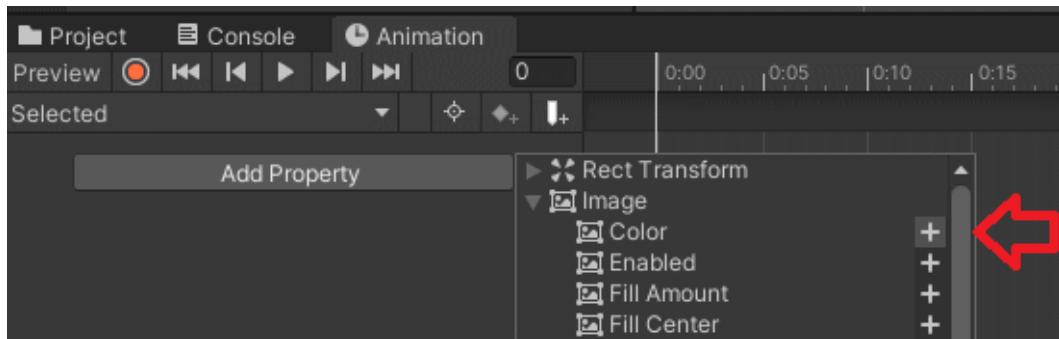


Figure 9.49: Adding the Color property to the Play Button timeline

12. Delete the second keyframe by selecting it and hitting the *Delete* key.

-
13. Change the **r**, **g**, **b**, and **a** properties to 1, 0, 0, and 1, respectively, on the remaining keyframe.

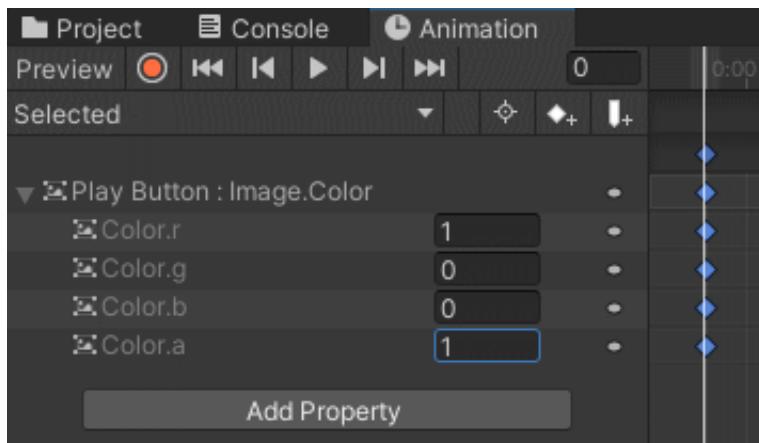


Figure 9.50: Adjusting the Color property of the Play Button

Now when you play the game, the `Play` Button should pulsate when it is not selected and change to red when it is.

That marks the end of the examples concerning Buttons, but we will continue to use them in future chapters.

Summary

Once you learn how to work with the Event System, working with buttons is an easy extension. Buttons are the most common interactive UI element, so having a good grasp on them is essential to effective UI development. Setting them up so that they function when clicked on is only half the process, though. You want to also ensure that you have your button navigation set up properly if you will be developing for PC, Mac, or console.

We're not done with Buttons! We'll be working with them throughout this book. Once we explore the Image component more thoroughly, we will cover more interesting button implementations and transitions.

In the next chapter, we'll cover the UI Text component!

10

UI Text and TextMeshPro

We've spent some time with UI Text objects already as they are the one of most basic graphical UI elements. We discussed them briefly in *Chapter 6* because it was pretty hard to start laying out UI without having anything to display visually. Text objects are also always children of new Buttons, which we discussed in the previous chapter. However, we haven't explored the properties of Text objects or how to work with them in code.

In this chapter, we will explore **UI Text** objects more thoroughly. We will also discuss **Text-TextMeshPro** objects and how they allow for even more control of the text in our game.

In this chapter, we will discuss the following topics:

- Using UI Text objects and setting their properties
- Using TextMeshPro -Text objects and setting their properties
- Working with fonts and font assets
- Using markup format with UI Text objects and style sheets with TextMeshPro objects
- Making Text that animates as if it's being typed out
- Developing a system that allows for easy Text translation
- Creating a custom font to be used with UI Text objects
- Creating Text that wraps along a curve and renders with a gradient

Note

All the examples shown in this section can be found within the Unity project provided in this book's code bundle. They can be found within the Chapter10 scene.

Each example image has a caption stating the example number within the scene.

In the scene, each example is on its own Canvas, and some of the Canvases have been deactivated. To view an example on a deactivated Canvas, simply select the checkbox next to the Canvas' name in the **Inspector**. Each Canvas has also been given its own Event System. This will cause errors if you have more than one Canvas activated at a time.

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2010>

UIText GameObject

You can create a new UIText object using + | UI | Text:

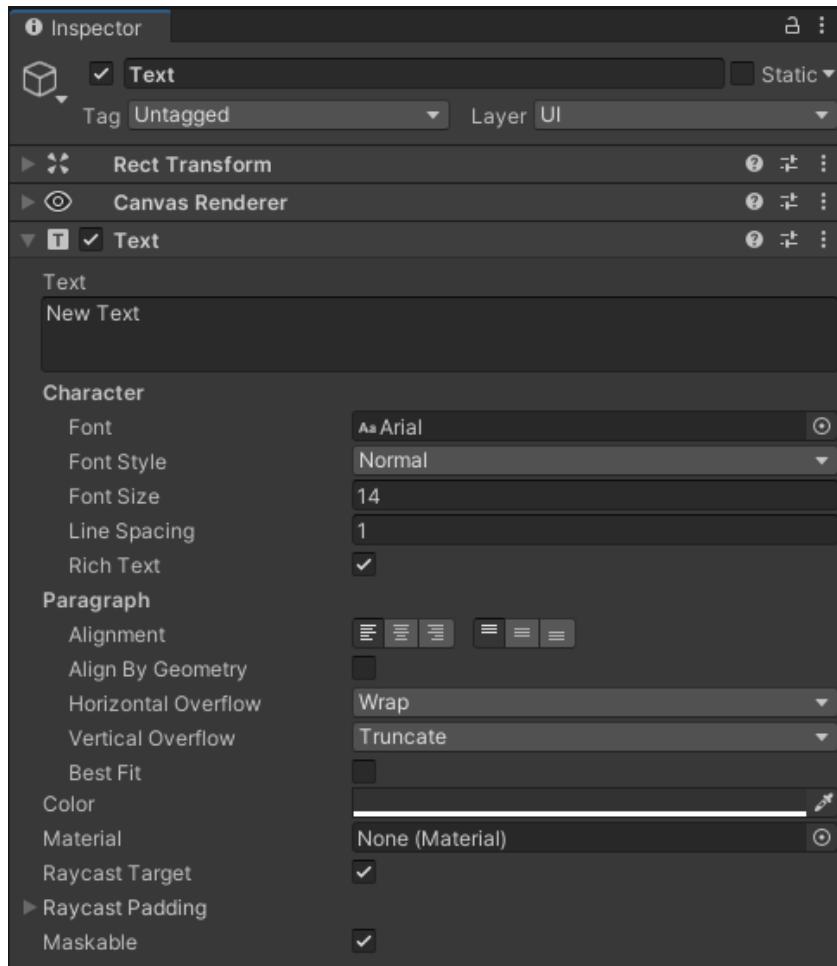


Figure 10.1: The UIText GameObject Inspector

The UIText GameObject contains the **Rect Transform** and **Canvas Renderer** components, as well as the **Text** component.

The UI Text component gives the object it is attached to a non-interactive text display. This component does not allow you to create all types of text you may be interested in, but it does allow most basic text displays.

The Text and Character properties

The **Text** property changes the text that will be displayed. Whatever is typed within this box will be displayed within the Text object.

Below the **Text** property is a group of **Character** properties. These properties allow you to change the properties of the individual characters within the **Text** property's field.

The **Font** property determines which font is used for the entire block of text. By default, the **Font** is set to **Arial**. To use any other font, you must import the font into your **Asset** folder. Refer to the *Working with fonts* section to learn how to bring in additional fonts.

Font Style provides a dropdown list of available font styles that come with the provided font. The possible styles are **Normal**, **Bold**, **Italic**, and **Bold And Italic**:

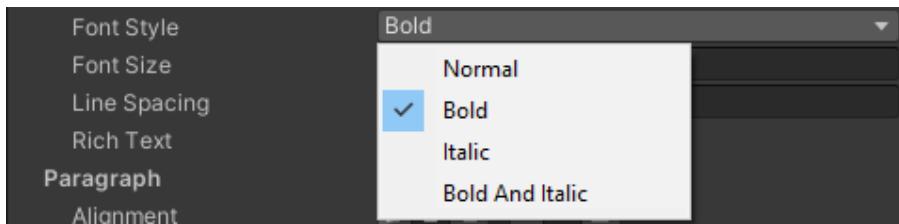


Figure 10.2: The UI Text component's Font Style options

Note

It's important to note that not all fonts will support all the listed font styles.

Font Size determines the size of the text, whereas **Line Spacing** represents the vertical spacing between each line of text.

If the **Rich Text** property is selected, you can include markup tags within the **Text** property field and they will appear with **Rich Text** styling rather than as typed. If this property is not selected, the text will display exactly as typed. Refer to the *Markup format* section for more information concerning writing with **Rich Text**.

The Paragraph properties

The next set of properties – the **Paragraph** properties (*Figure 10.1*) – allow you to determine how the text will display within (or outside of) the Rect Transform's bounds.

The **Alignment** property determines where the text will align based on the Rect Transform bounds. You can choose both horizontal and vertical alignment options. The buttons represent the position relative to the Rect Transform bounds, so the left horizontal alignment will have the text pushed up to the edge of the Rect Transform's left bound:



Figure 10.3: The UI Text component's Alignment options

The **Align by Geometry** property aligns the text as if the glyphs or characters are cropped down to their opaque area rather than the area they cover. This cropping is based on their character map. This can give a tighter alignment but might also cause things to overlap.

The **Horizontal Overflow** property determines what happens to text if it is too wide for the Rect Transform area. There are two options: **Wrap** and **Overflow**. **Wrap** will cause the text to continue on the next line, whereas **Overflow** will cause the text to expand past the rectangular area:

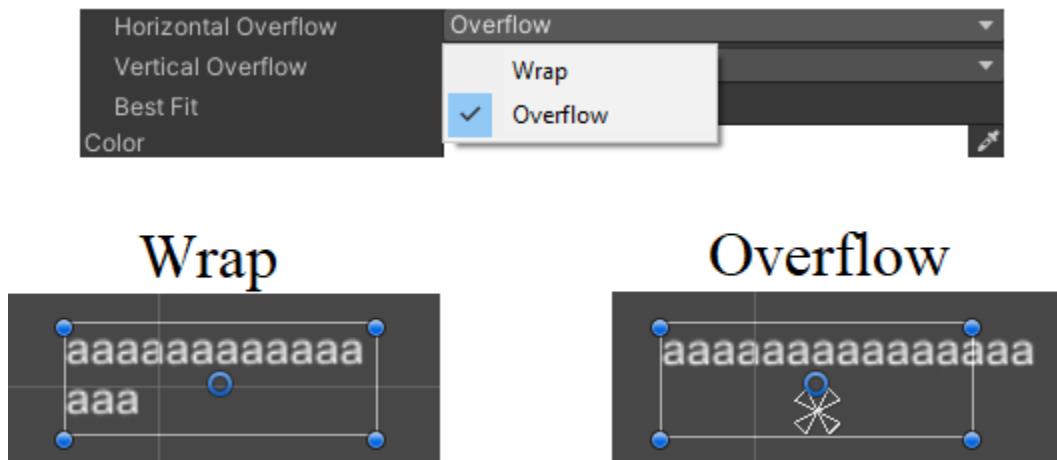


Figure 10.4: Horizontal Overflow Example in the Chapter10 scene

The **Vertical Overflow** property determines what happens to text if it is too long for the Rect Transform area. There are two options: **Truncate** and **Overflow**. **Truncate** will cut off all text outside of the rectangular area, whereas **Overflow** will cause the text to expand past the rectangular area. In the following figure, both Text boxes have the same text, but **Truncate** removes the last two lines of text due to them being outside of the rectangular area, while **Overflow** allows it to go outside the box:

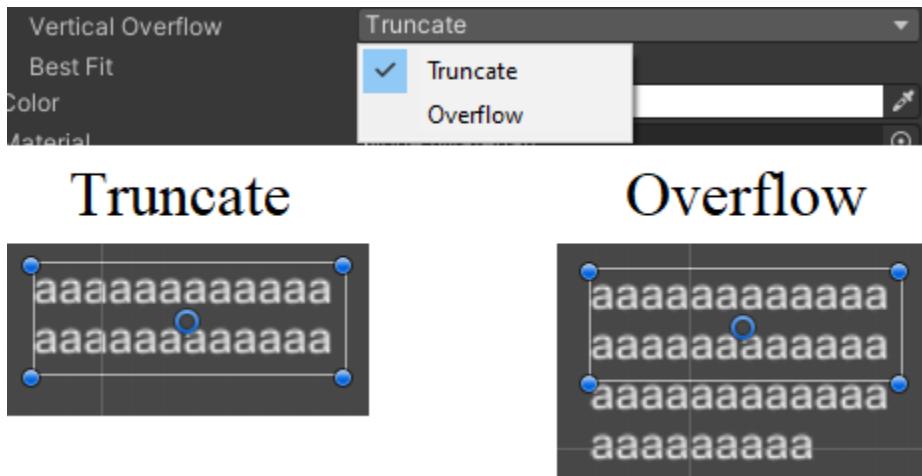


Figure 10.5: Vertical Overflow Example in the Chapter10 scene

The **Best Fit** property attempts to resize the text so that all of it fits within the rectangular area. When you select the **Best Fit** property, two new properties will become available: **Min Size** and **Max Size**. These properties allow you to specify the range the font size can maintain.

Keep in mind that depending on the text you have written, the **Horizontal Overflow** property may cause this to work slightly differently than you'd expect.



Figure 10.6: Best Fit Example in the Chapter10 scene

For example, the two Text boxes in *Figure 10.6* have **Best Fit** selected, but the first has **Horizontal Overflow** set to **Wrap**, while the second has it set to **Overflow**.

The Color and Material properties

The **Color** and **Material** properties allow you to adjust the appearance of the Text's font. The **Color** property will set the base rendering color of the Text and is the quickest way to change the font's color. By default, this property is set to a very dark (not fully black) gray. The **Material** property allows you to assign a material to your font. This gives you more control over the look of the font and also allows you to apply specific shaders. By default, this property is set to **None**.

The Raycast and Maskable properties

The **Raycast Target** property determines whether the object's Rect Transform area will block raycasts or not. If this property is selected, clicks will not register on UI objects behind it. If it is not selected, items behind the object can be clicked. If you'd like for the Text to block raycasts, but not over its entire area, you can adjust the area with the various **Raycast Padding** properties.

The last property, **Maskable**, determines if the Text can be masked. We will discuss this in *Chapter 12*.

Text-TextMeshPro

There is a bit of a limitation to what you can do with UI Text. If you find yourself wanting to accomplish some formatting with your text that you are unable to do with UI Text, you may be able to accomplish it with a TextMeshPro GameObject. For example, if you want to use underlined text, I recommend using a TextMeshPro GameObject. TextMeshPro assets allow for significantly more text control. Additionally, its rendering allows text to appear crisp at more resolutions and point sizes than what's possible with the standard UI Text.

TextMeshPro used to be a paid asset in the Unity Asset Store, but it was adopted by Unity around March 2017 and is now available for free. However, to use TextMeshPro assets, you have to download the necessary resources. To download the TextMeshPro resources, attempt to add a TextMeshPro - Text to your scene by going to + | UI | Text - TextMeshPro; you will see the following popup:

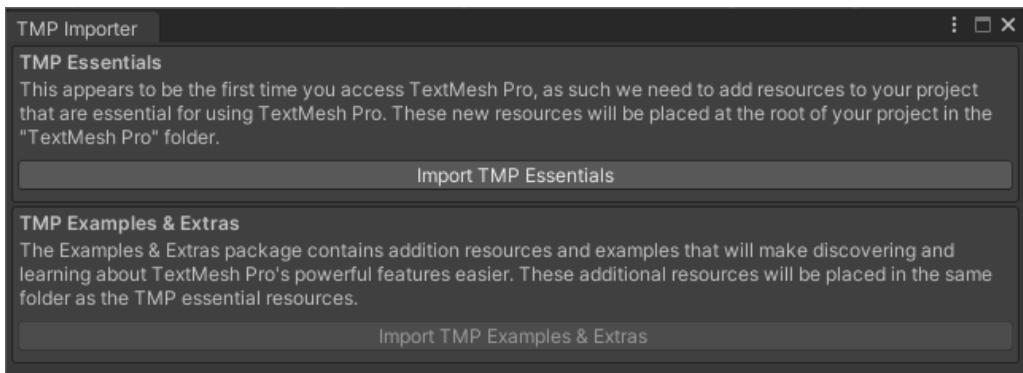


Figure 10.7: The TMP Importer

Select **Import TMP Essentials** to get all the necessary assets. I also recommend selecting **Import TMP Examples & Extras**.

Note

Selecting any of the TextMeshPro GameObjects from the UI menu (Text - TextMeshPro, Button - TextMeshPro, Dropdown - TextMeshPro, or Input Field - TextMeshPro) will bring up the popup from the preceding screenshot and allow you to download the necessary assets.

Once you've downloaded it, you will not have to do so again.

Due to the robustness of TextMeshPro, I sadly can't cover everything you can do with it within this chapter. Instead, I will provide a broad overview of its functionality. Luckily, the TextMeshPro asset comes with many examples and good documentation, which can be found here:

<https://docs.unity3d.com/Packages/com.unity.textmeshpro@4.0/manual/index.html>

When you create a new **TextMeshPro - Text** GameObject, you will see a GameObject with the following component:

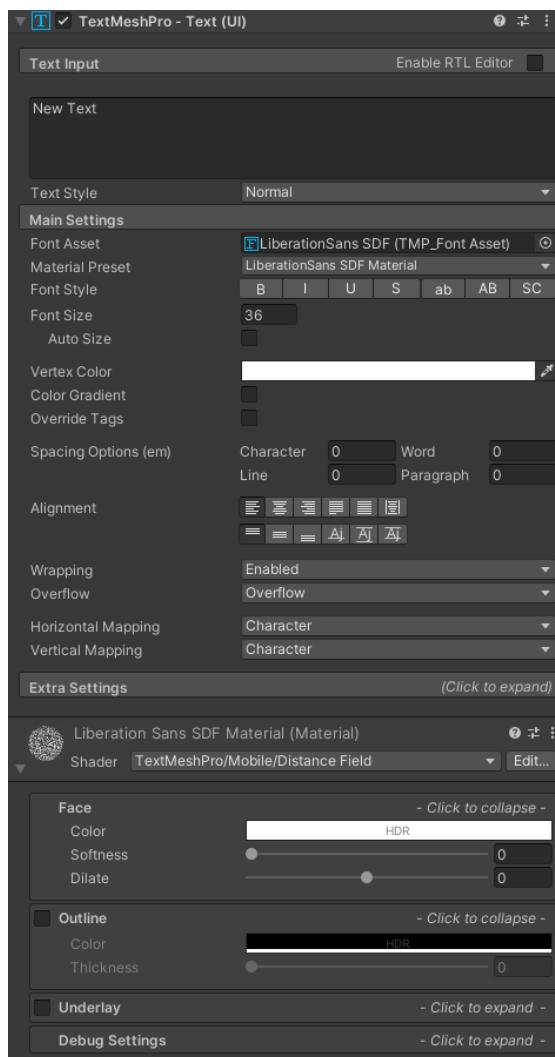


Figure 10.8: The TextMeshPro - Text component

You can also create Text-TextMeshPro GameObjects outside of Unity's UI system via **GameObject | 3D Object | Text-TextMeshPro**. This will render the text independent of the UI and without a Canvas.

The GameObject itself will be named **Text (TMP)** and, for simplicity's sake, is how I will reference it.

As with all other UI objects, a **RectTransform** and a **Canvas Renderer** component are attached to the GameObject as well. The graphic display of the **Text (TMP)** GameObject is controlled by the **TextMeshPro - Text (UI)** component.

Let's investigate the properties of the **TextMeshPro - Text (UI)** component.

Text Input properties

You can enter the text you wish to display within the **Text Input** section:

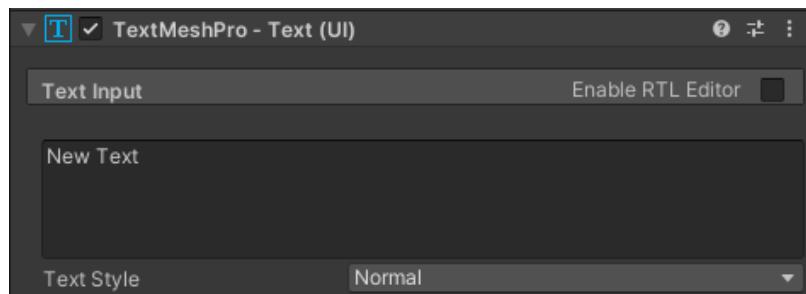


Figure 10.9: The TextMeshPro - Text component's Text Input settings

The **Enable RTL Editor** property allows you to create text that will display from right to left which is necessary for some languages. When you select it, the text will appear in a second area in its right-to-left order:

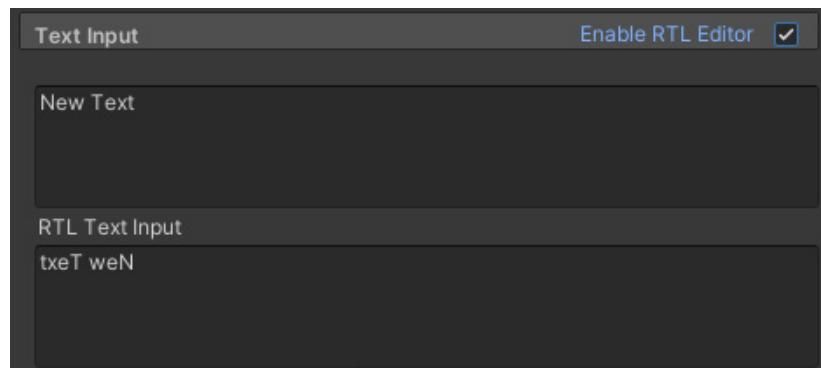


Figure 10.10: The TextMeshPro - Text component's RTL Text Input setting

The **Text Style** setting lets you specify a style for the text. You'll see multiple pre-defined options from the dropdown:

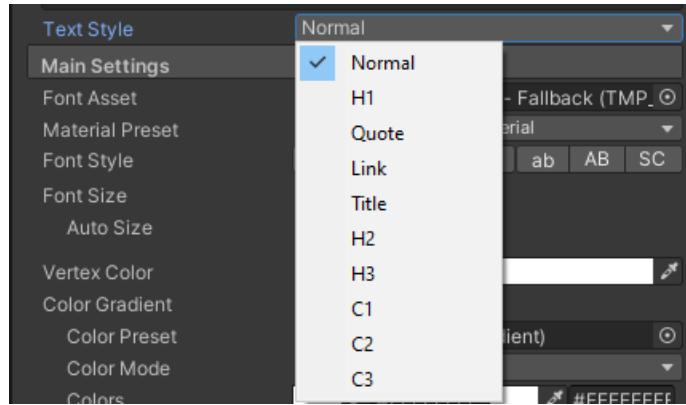


Figure 10.11: The TextMeshPro - Text component's Text Style settings

We'll look at how to use this more thoroughly in the *Style sheets* section of this chapter.

Main Settings

The **Main Settings** section allows you to adjust all the properties of the text:

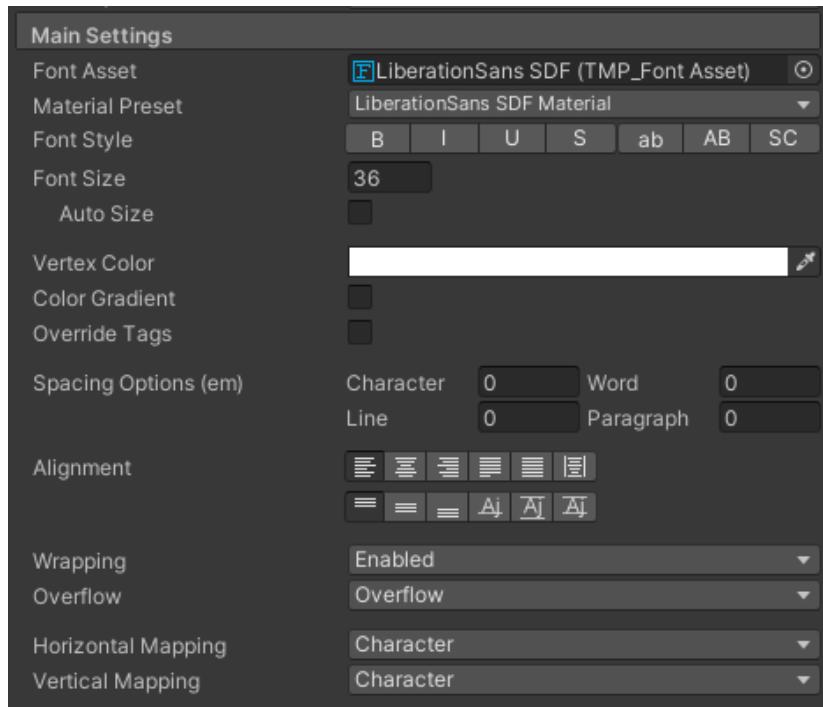


Figure 10.12: The TextMeshPro - Text component's Main Settings section

The **Font Asset** property represents the font that will be used. By default, **Font Asset** is set to `LiberationSans SDF (TMP_Font Asset)`. I want to point out that fonts and font assets are two different things. Fonts are used in UI Text GameObjects, whereas font assets are used in TextMeshPro GameObjects. I'll discuss these differences, as well as how to import new fonts and create new font assets, in the *Working with fonts* section.

The **Font Asset** property needs a material to render. Any material that contains the name of the font asset will appear in the **Material Preset** list. You can create your own material, but when you create a new font asset, it comes with a default material. Whichever material is selected here will also appear at the bottom of the component, below the **Extra Settings**. From this area, you can also select the material's shader:

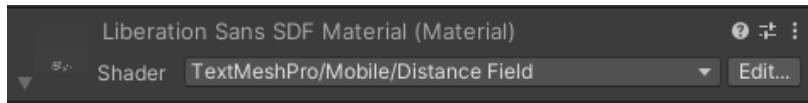


Figure 10.13: The TextMeshPro - Text's material properties

The **Font Style** property allows you to create basic formatting for your text. You can select from **Bold**, **Italic**, **Underline**, **Strikethrough**, **Lowercased**, **Uppercased**, or **Small Caps**. You can choose any combination of the first four settings; however, you can only choose between **Lowercase**, **Uppercase**, or **Small Caps**.

The **Font Size** property works as you would expect, but you also have the option to select **Auto Size**. The **Auto Size** property will attempt to fit the text within the bounding box of the Rect Transform as best as it can based on the properties you specify:

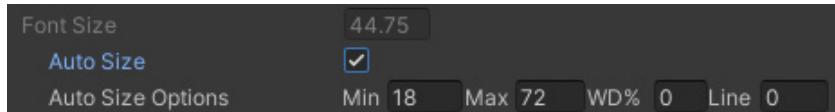


Figure 10.14: The TextMeshPro - Text component's Font Size properties

You can specify the minimum (**Min**) and maximum (**Max**) font size along with the **WD%** and **Line** properties. The **WD%** property allows you to squeeze text horizontally to make the characters taller, whereas the **Line** property allows you to specify line height.

You can change the color of the text using either the **Vertex Color** property or the **Color Gradient** property:

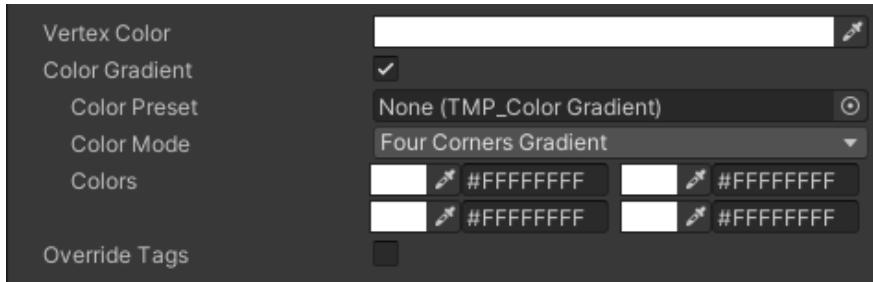


Figure 10.15: The TextMeshPro - Text component's color properties

An example of using the **Color Gradient** property can be found at the end of this chapter in the *Examples* section. You can select **Override Tags** if you want the color settings to override any `<color>` markup tags.

You can set the spacing between **Characters**, **Words**, **Lines**, and **Paragraphs** in the **Spacing Options** area.

You also have significantly more **Alignment** options available to you in TextMeshPro-Text than you do with the standard UI Text.

Just as with UI Text, you can enable or disable **Wrapping**. You have significantly more **Overflow** options, however, as shown in the following screenshot:

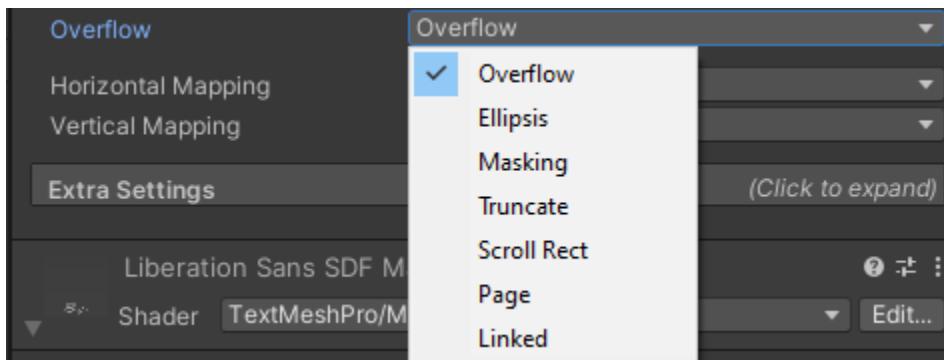


Figure 10.16: The TextMeshPro - Text component's Overflow options

Overflow and **Truncate** work similarly to those on the UI Text objects. Of the other options, the only one I want to mention at this time is the **Ellipsis** option. It will truncate the text to the text box area but add an ellipsis (...):



Figure 10.17: Text Mesh Pro Overflow Example in the Chapter10 scene

The **Horizontal Mapping** and **Vertical Mapping** properties allow you to affect the way a texture is displayed across the text:

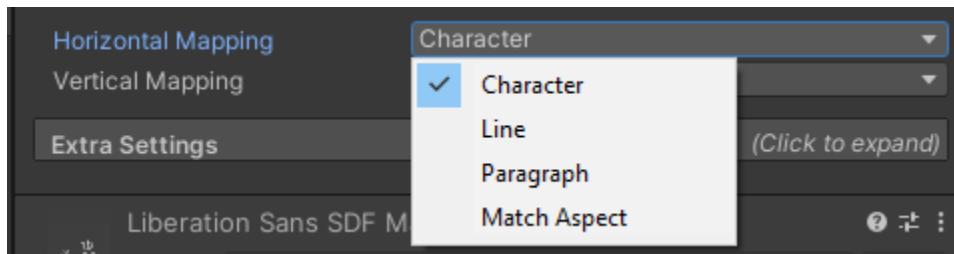


Figure 10.18: The TextMeshPro - Text component's Horizontal Mapping options

Most of the work you do to customize your font will be done in the **Text Input** section and the **Main Settings** section, but let's look at some more nuanced settings that you will adjust with your fonts.

Extra Settings

The **Extra Settings** menu has to be expanded to be visible. It allows you to adjust some less-common settings of the font:

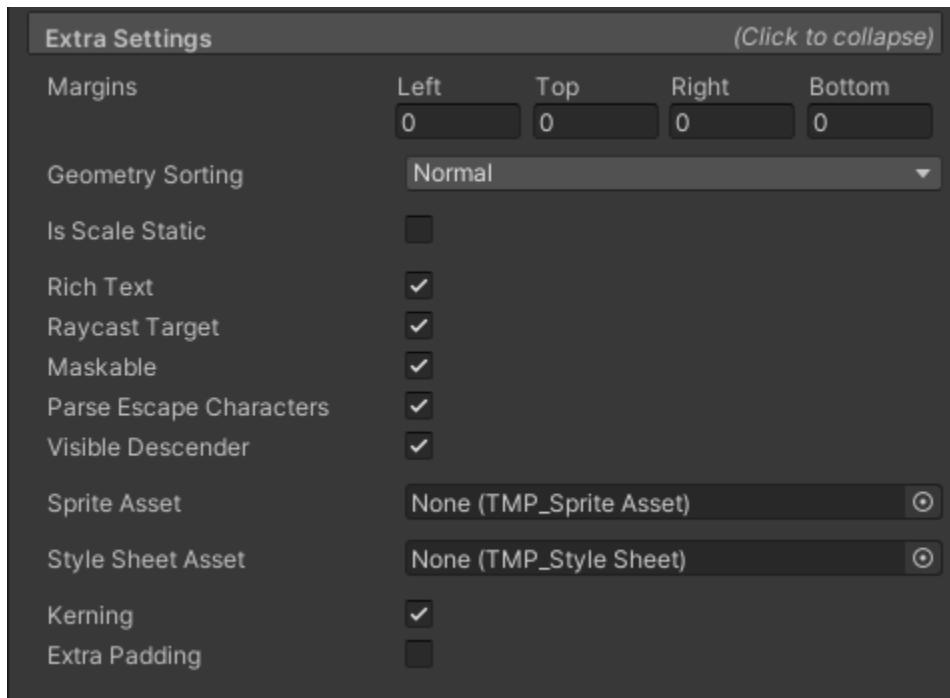


Figure 10.19: The TextMeshPro - Text Extra Settings section

The most notable properties in this menu are the ability to add **Margins** and the ability to enable **Raycast Target**.

Lastly, you can specify whether or not you want to **Enable Kerning** or allow **Extra Padding**. Selecting **Kerning** will use the kerning data provided by the font file. Selecting **Extra Padding** will add a little padding around the glyphs of the sprite on its sprite atlas.

Note

You can learn about any of the properties I glossed over or skipped here: <https://docs.unity3d.com/Packages/com.unity.textmeshpro@4.0/manual/TMPOBJECTUITEXT.html>.

TextMeshPro Project Settings

In addition to being able to adjust the individual settings of each TextMeshPro object you have within your scene via their components, you can also adjust TextMeshPro project-wide settings via the **Project Settings** (**Edit > Project Settings**):

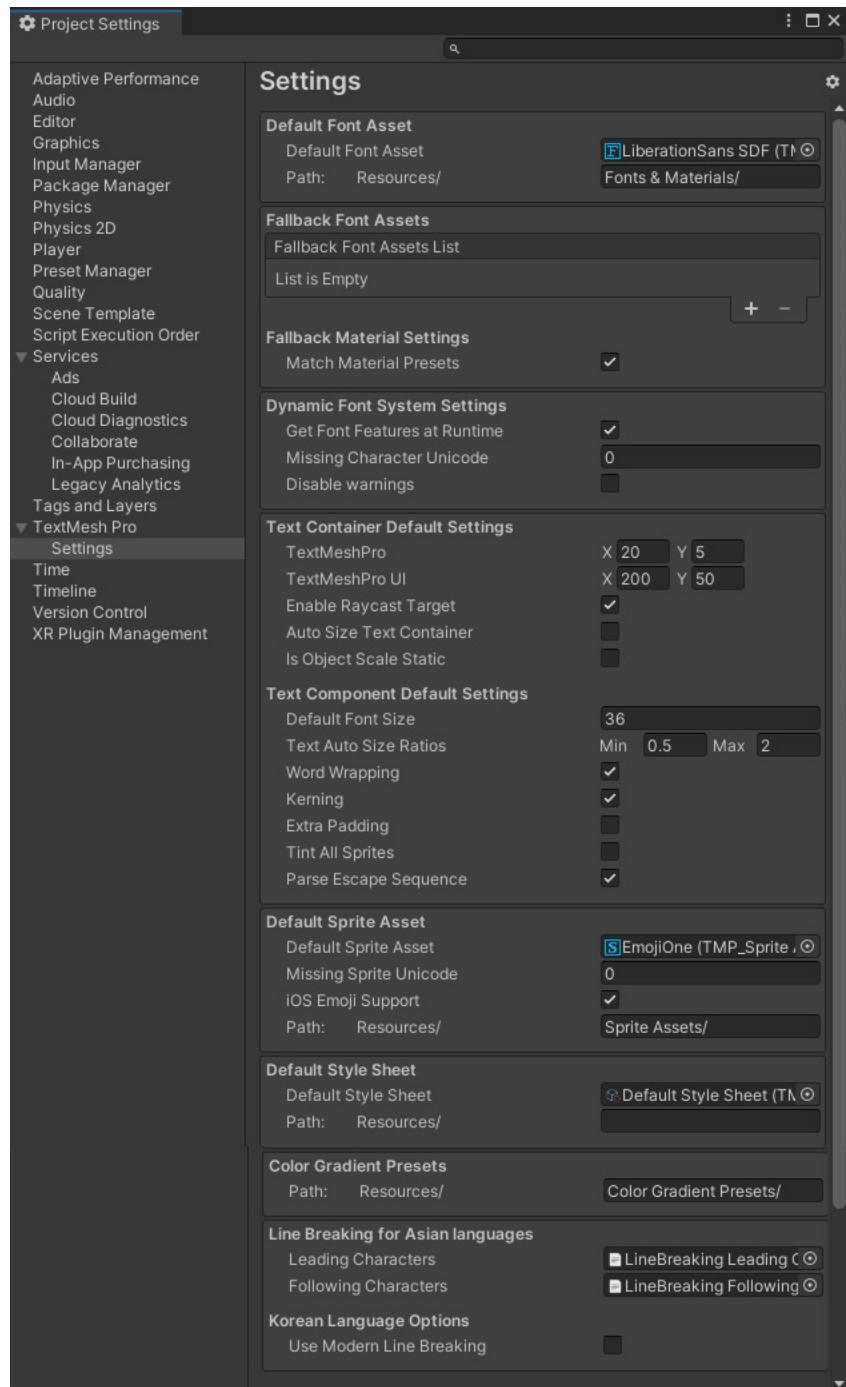


Figure 10.20: The TextMeshPro – Project Settings

These let you set the various defaults for newly created TextMeshPro objects. You can learn more about each property at <https://docs.unity3d.com/Packages/com.unity.textmeshpro@4.0/manual/Settings.html>.

Working with fonts

It's extremely likely that you won't want to use the default **Arial (UI Text)** or **Liberation Sans (TMP Text)** fonts and will want to bring a custom font into your project. Let's explore how you can both find these text resources and use them in your project.

Note

You do not have to install a font onto your computer (to use within programs outside of Unity) to use the font within Unity.

Importing new fonts

The font file formats accepted by Unity are **.tff** (TrueType) and **.otf** (OpenType). You can get these files in multiple places. My favorite places to find fonts are as follows:

- *Google Fonts*: <https://fonts.google.com/>
- *DaFont*: <http://www.dafont.com/>
- *Font Squirrel*: <https://www.fontsquirrel.com/>

All the fonts on *Google Fonts* are open source and are free for personal or commercial use (at least at the time of writing this book), but the fonts on *Font Squirrel* and *DaFont* have varying licensing options. Ensure that any font you get has a licensing agreement that meets your needs before you use it.

Once you've downloaded the font of your choice, simply drag the font into your project's **Assets** folder. I highly recommend that you create a folder called **Fonts** within your **Assets** folder in which you place all of your font files.

Then, you can adjust the font's import properties in the **Inspector** if you so choose. The following screenshot shows the import settings of the **BungeeShade-Regular** font, which has been downloaded from Google Fonts:

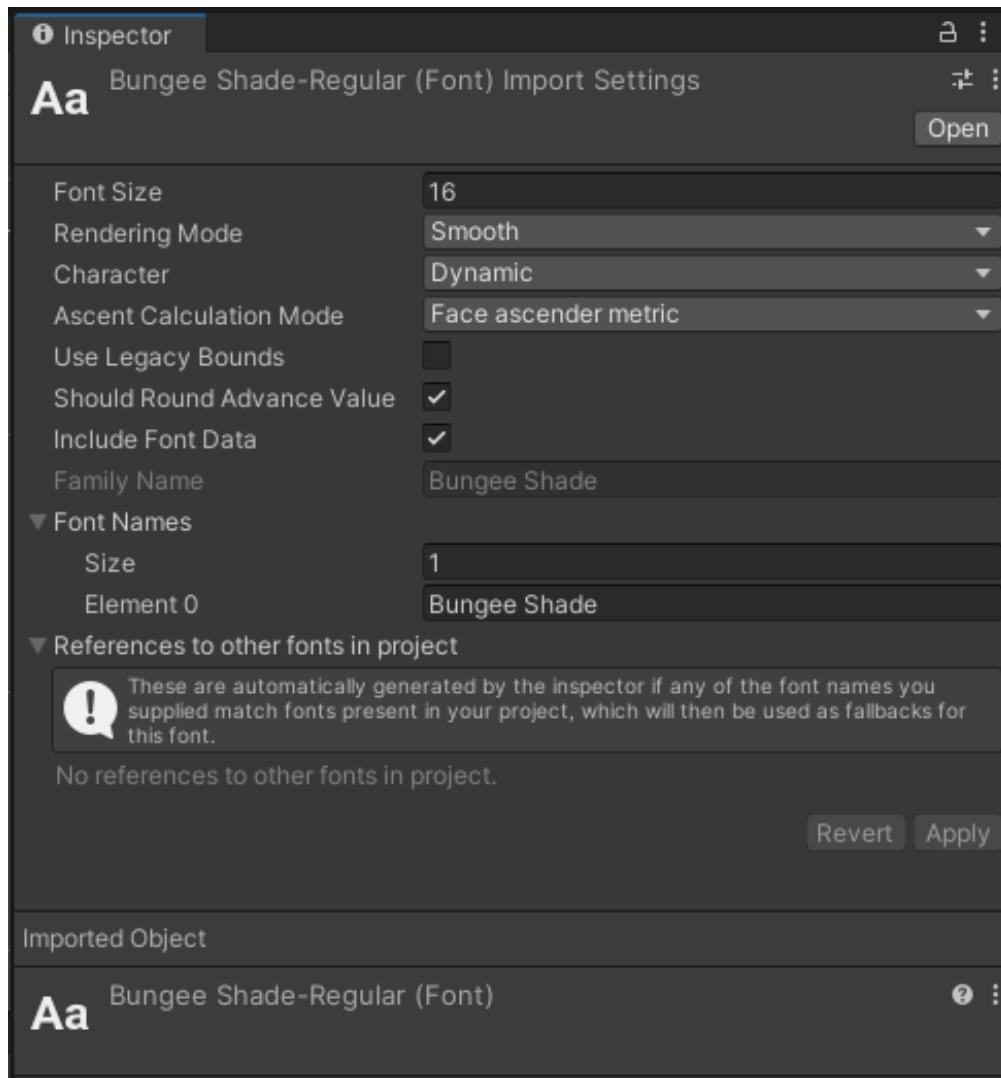


Figure 10.21: The Inspector of the Bungee Shade-Regular font

Here, you can adjust quite a few of the settings concerning how the font will be handled by the engine.

Font Size

The **Font Size** setting determines how large the font will appear on its Unity-created texture atlas. Increasing or decreasing the **Font Size** setting will change the size of the various glyphs on the texture atlas. If your font appears fuzzy in your game, adjusting the **Font Size** setting may improve its appearance.

Rendering Mode

The **Rendering Mode** settings tell Unity how the glyphs will be smoothed. The possible options are **Smooth**, **Hinted Smooth**, **Raster**, and **OS Default**.

The **Smooth** rendering mode is the fastest rendering mode. It uses anti-aliased rendering, which means it will smooth out jagged, pixelated edges. The **Hinted Smooth** rendering mode will also smooth out edges, but it will use the “hints” contained within the font’s data files to determine how to fill in those jagged edges. This is a slower rendering mode than **Smooth** but will likely look crisper and be easier to read than **Smooth**. The **Hinted Raster** rendering mode does not provide anti-aliasing and instead provides aliased or jagged edges. This is the crispest and the quickest of the rendering modes. **OS Default** will default to whatever the operating system’s preferences are set to on Windows or Mac OS. This will select from **Smooth** or **Hinted Smooth**.

Character

The **Character** property determines which character set of the font will be imported into the font texture atlas. There are six options: **Dynamic**, **Unicode**, **ASCII default set**, **ASCII upper case**, **ASCII lower case**, and **Custom set**:

Setting the **Character** property to **Dynamic** (which is the default) will only include the characters that are needed. This reduces the texture size needed for the font and, in turn, the download size of the game.

Unicode is used for languages that have characters that are not supported in an ASCII character set. So, for example, if you want to display text in Japanese, you will want to use **Unicode**.

If you’d like to include **Unicode** characters in your scripts, you need to save them with UTF-16 encoding. This will allow you to type **Unicode** characters directly in your code as strings so that they can display in your Text objects on screen.

The *Noto fonts* provided by *Google Fonts* provide support for many languages and can be very helpful if you want to create a game that is translated into multiple languages. The *Noto fonts* can be found at <https://www.google.com/get/noto/>.

American Standard Code for Information Interchange (ASCII) is a set of characters from the English-language character set. The three variations of **ASCII** character sets allow you to choose between the full set, only uppercase, or only lowercase characters. You can find a list of the **ASCII** characters at <http://ascii.cl/>.

A **Custom set** character will allow you to import your own texture atlas for your own custom font. I find this most commonly used when developers want beautified text with an extremely limited character set, such as numbers only.

Ascent Calculation Mode

The **ascent** of a font is the distance between the font's baseline and the highest glyph point. There is no standard for how this supposed *highest glyph point* is determined, so different modes are available in Unity to choose from, each determining a different *highest glyph point*. The **Ascent Calculation Mode** property determines how the ascent will be calculated. There are three options for how this calculation will be chosen: **Legacy version 2 mode (glyph bounding boxes)**, **Face ascender metric**, and **Face bounding box metric**. The method chosen may affect the vertical alignment of the font.

Legacy version 2 mode (glyph bounding boxes) measures the ascent using the highest point reached by any one of the font's glyphs listed within its character set as the height. This only uses those listed in the character set, and not all glyphs may be included within that set. **Face ascender metric** uses the face ascender value that is defined to measure the ascent, whereas **Face bounding box metric** uses the face bounding box to measure the ascent.

Typography is a lot more complicated than many people realize, and it's too complicated to fully cover in this book – not to mention I am no typography expert. If you'd like to learn more about glyph metrics, a good introduction can be found at <https://www.freetype.org/freetype2/docs/glyphs/glyphs-3.html>.

Dynamic font settings

When you import your font with a dynamic character set, two new settings are made available: **Include Font Data** and **Font Names**.

Include Font Data builds the font file with the game. If this is not selected, the game will assume that the player has the font installed on their machine. If you are using a font you have downloaded from the web, it is a pretty safe bet that the end user will not have the font installed and you should leave **Include Font Data** selected.

Font Names is the list of names of fonts that Unity will fall back on if it cannot find the font. It will need to fall back on this font name if the font doesn't include the requested glyph or the **Incl. Font Data** property was deselected and the user does not have the font installed on their machine. If Unity cannot find the font, it will search the game's project folder or the user's machine for a font matching one of the names listed in **Font Names**. Once the fonts have been typed into **Font Names**, the appropriate fonts will be listed in the **References to other fonts in project** section:

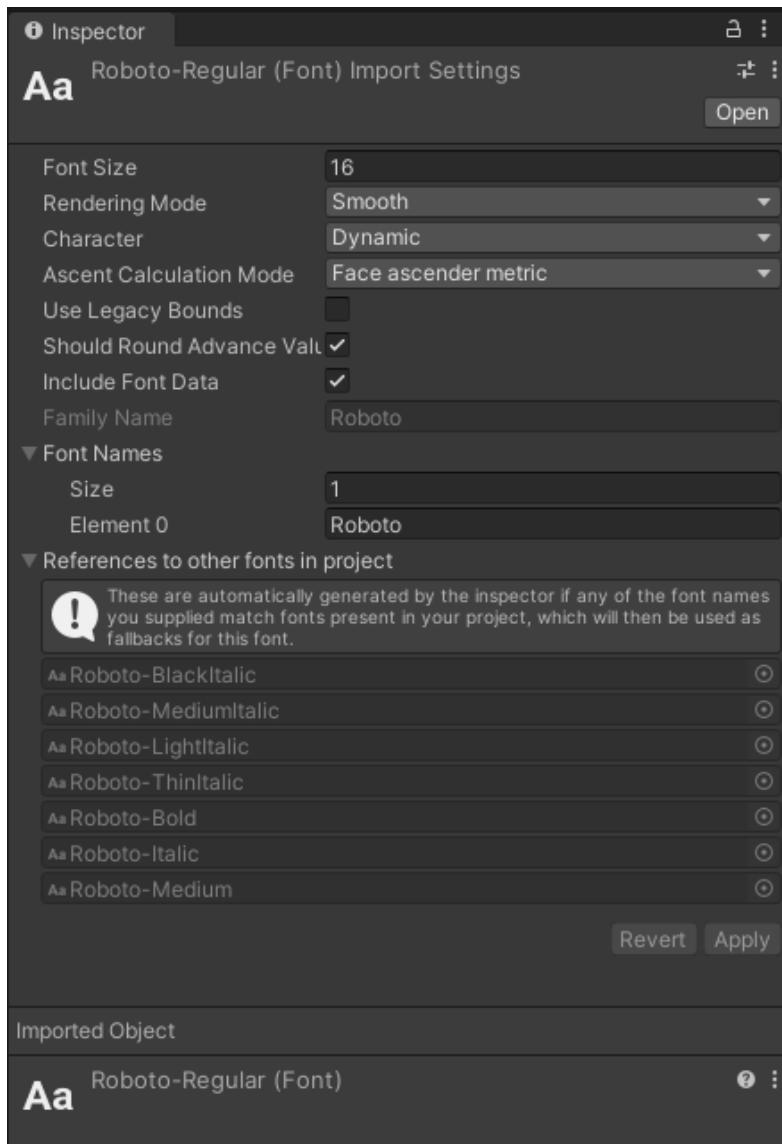


Figure 10.22: The Inspector of the Roboto-Regular font

If Unity cannot find one of the fonts listed, it will use a font provided in a predefined list of fallback fonts hard-coded within Unity.

Some platforms don't have built-in fonts in their system or can't access built-in fonts. These platforms include WebGL and some console systems. When building to these platforms, Unity will always include fonts, regardless of the setting chosen for **Include Font Data**.

Importing font styles

If you bring in a font that has multiple styles (that is, bold, italic, or bold and italic), you must bring in all the font styles for them to appear properly. Bringing the fonts into your project may not be sufficient for the font to recognize which fonts should be used when the **Bold**, **Italic**, and **Bold and Italic** font styles are selected, however. For example, the following screenshot shows the **Roboto-Regular** Google Font with the **Bold and Italic Font Style** on the top line of text and the **Roboto-BoldItalic** Google Font with the **Normal Font Style** on the bottom line:



Figure 10.23: Styles of the Roboto font

If the **Font Style** property were working correctly, the two lines should match. However, as you can see, they do not. To make the fonts appear correctly, select the regular font, retype the **Font Name** property to make all the appropriate ones appear in the font list (as shown in *Figure 10.22*), and hit **Apply**. After doing so, the two fonts should appear the same:



Figure 10.24: Styles of the Roboto font applied correctly

Now that we've reviewed how to import fonts, let's look at how to create custom fonts.

Custom fonts

You can create a custom font by selecting **Create | Custom Font** from the project window. To use a custom font, you will need a font material and font texture. How to do this is covered in the *Examples* section of this chapter. Once you create your custom font, you will be given the following properties to set:

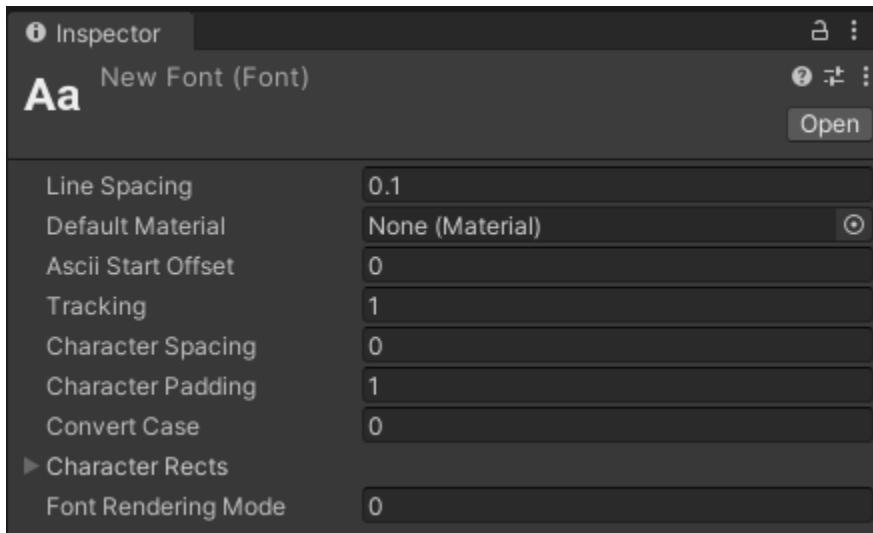


Figure 10.25: A custom font's Inspector

The **Line Spacing** property specifies the distance between each line of text.

The **Ascii Start Offset** property defines the first ASCII character in the font's character set. For example, if you created a font that only included numbers, your character set would start with the number 0, which is ASCII index 48. Therefore, you would set the **ASCII Start Offset** to 48, indicating the first character in this font's character set is the character 0. You can determine the ASCII index number for individual characters at <http://ascii.cl/>.

The **Tracking** property represents the spacing between characters for a full line of text. It allows the spacing between all characters to be uniform.

The **Kerning** property has been replaced with the **Tracking** property. **Tracking** and **Kerning** are both properties related to spacing between characters, but they are different. For more information, check out <http://www.practicalecommerce.com/Typography-101-The-Basics>.

Character Spacing is the amount of space between characters, whereas **Character Padding** is the amount of padding surrounding individual characters before the spacing.

The **Character Rects** property determines how many total characters are in your font. Changing the number from 0 to any positive number will provide a list of **Elements** that can be expanded:

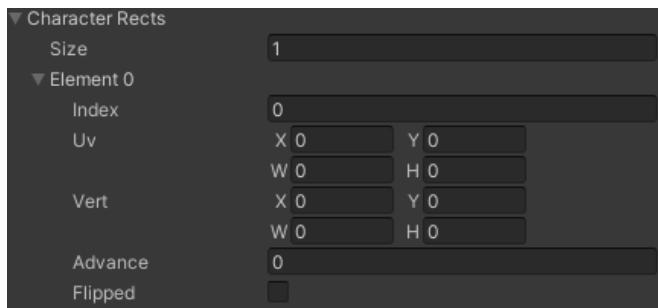


Figure 10.26: The Character Rects of a custom font

Each element represents a character on your character set. The **Index** is the ASCII index of the specified character.

The **UV** width (**W**) and height (**H**) values of your font represent the percentage of the width and height your characters occupy. For example, if you had a font file with five columns of characters and two rows of characters, the **W** would be one-fifth or .2 and the **H** would be one-half or .5. This should be consistent throughout all of your characters. The **X** and **Y** values are determined by multiplying the **W** and **H** values by the column or row that the character is located in.

The **Vert** properties represent the width and height of the character in pixels. The **H** value is always negative. So, if a character's pixel dimension is 50 by 50, the **W** and **H** values would be 50 and -50, respectively. To be perfectly honest, I am not exactly sure why the height property is always negative and I can't seem to find the answer. I suspect that it has something to do with the fact that this uses texture coordinates. The **X** and **Y** values of **Vert** represent a shift in position, where these numbers can be negative or positive.

The **Advance** setting represents the pixel distance between the specific character and the next character.

The **Flipped** setting indicates if the glyph is flipped of how it should be displayed.

Please refer to the *Examples* section for an example of calculating the **UV**, **Vert**, and **Advance** values of a custom font.

At the time of writing, not all the properties for custom fonts are fully defined within Unity's documentation and a few of these properties are a bit ambiguous. For example, **Convert Case** used to be a dropdown menu, and it is unclear to me how it is now used since it only accepts a number input. Perhaps, in the future, these properties will be better defined and the manual will be updated to reflect the new changes made, at <https://docs.unity3d.com/Manual/class-Font.html>.

Font assets

Recall that TextMeshPro objects need to use font assets, not fonts. So, if you've downloaded some fonts for your project, you won't see them as possible font options with a TextMeshPro object. If you've imported the TextMeshPro examples, you may have few options other than Liberation Sans available to you. To use a different font in a TextMeshPro GameObject, you cannot simply drag a new font into the **Font Asset** slot; you must create a Font Asset via the **Font Asset Creator**.

To access the **Font Asset Creator**, select **Window | TextMeshPro - Font Asset Creator**. This will allow you to convert a font file into a font asset that can be used by TextMeshPro:



Figure 10.27: The Font Asset Creator window

There are quite a few settings that can be controlled via the **Font Asset Creator**. You can find a breakdown of these settings at <https://docs.unity3d.com/Packages/com.unity.textmeshpro@4.0/manual/FontAssetsCreator.html?q=font%20asset%20creator>. I will cover the process of creating a font asset in the *Examples* section of this chapter.

Exploring the markup format

HTML-like markup language can be included within the text field of the **Text** or **TextMeshPro - Text (UI)** components to format the text. This markup format is HTML-like in that it uses angle bracket tags around the text that is to be formatted. The tags use the `<tag>the text you wish to format</tag>` format, where you replace tag with the appropriate tag. These tags can be nested, just as they can in HTML.

To use markup format on a **Text** object, you must first select the **Rich Text** property within the **Text** component. By default, it works within a **TextMeshPro** object.

This formatting allows you to change the font style, font color, and font size. The following chart lists the tags necessary to perform the specified formatting:

Format	Tag
Bold	<code>b</code>
Italic	<code>i</code>
Color	<code>color</code>
Size	<code>size</code>

Table 10.1: Formats and their tags

Now, let's look at how to change the style of a font with markup.

Font style

You can change the font style of text using the bold and italic tags.

To add a bold font style to text, add the `` tags around the text you wish to bold. To add an italic font style to text, add the `<i></i>` tags around the text you wish to italicize.

<code>Bold Text</code>	Bold Text
<code><i>Italic Text</i></code>	<i>Italic Text</i>
<code><i>Nested Text</i></code>	<i>Nested Text</i>

Table 10.2: Examples of formatting tags

Now, let's look at how to change the color of a font with markup.

Font color

You can change the color of your font with either the hex value representation of a number or using the color name. To change the color of the text, add `<color=value></color>` around the text you wish to color, where you place either the hex value (following a #) or the color name where the word value appears.

Only a limited set of colors have names that can replace the hex values. The color names that are recognized are black, blue, brown, cyan, darkblue, green, grey, lightblue, lime, magenta, maroon, navy, olive, orange, purple, red, silver, teal, white, and yellow. You can also use aqua in place of cyan and fuchsia in place of magenta.

When using the color tag, any text not surrounded by the color tag will be colored based on the **Color** property selected.

The following table shows how to use the color tag:

<code><color=#ff0000>Red Text</color></code>	Red Text
<code><color=red>Red Text</color></code>	Red Text
<code><color=red>Red</color> Text</code>	Red Text

Table 10.3: Color formatting tag examples

Now, let's look at how to change the size of a font with markup.

Font size

To change the font size, add the `<size=#></size>` tags around the text you wish to resize. Any text not within the tag will be sized based on the **Font Size** property setting. The following example shows how to use the `size` tag:

<code><size=30>Small</size> Normal <size=100>Big</size></code>	Small Normal Big
--	------------------

Table 10.4: Examples of multiple font size tags

So far, we've looked at the ways we can format fonts with markup. Now, let's look at how to format a font using a style sheet.

Using style sheets

In addition to the markup tags described in the preceding section, TextMeshPro objects can also use style sheet tags. As you may recall from *Figure 10.9*, the **TextMeshPro - Text (UI)** component has a property labeled **Text Style** with ten options in its dropdown. From there, you can select any one of the styles shown in the following figure:



Figure 10.28: The various default styles from the default style sheet

All of these styles are pre-defined by the **Default Style Sheet** option that's assigned for the **TextMeshPro Settings** within the **Project Settings** (as we discussed in the *TextMeshPro Project Settings* section).

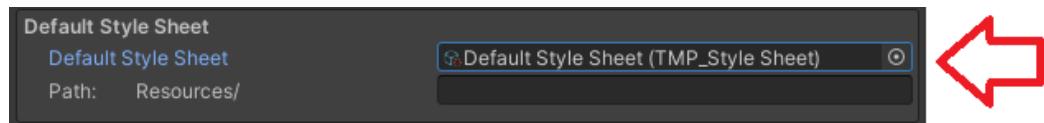


Figure 10.29: The Default Style Sheet setting from the Project Settings

Clicking on the **Default Style Sheet (TMP_Style Sheet)** object within the **Project Settings** will select the **Default Style Sheet** within the **Assets** folder:

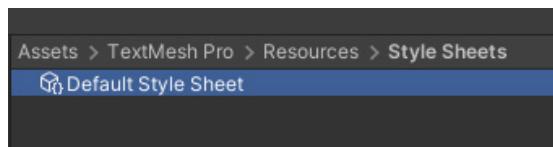


Figure 10.30: The Default Style Sheet asset within Resources

You can then view the **Default Style Sheet** asset's **Inspector** and see how each of the default style sheet tags are defined:

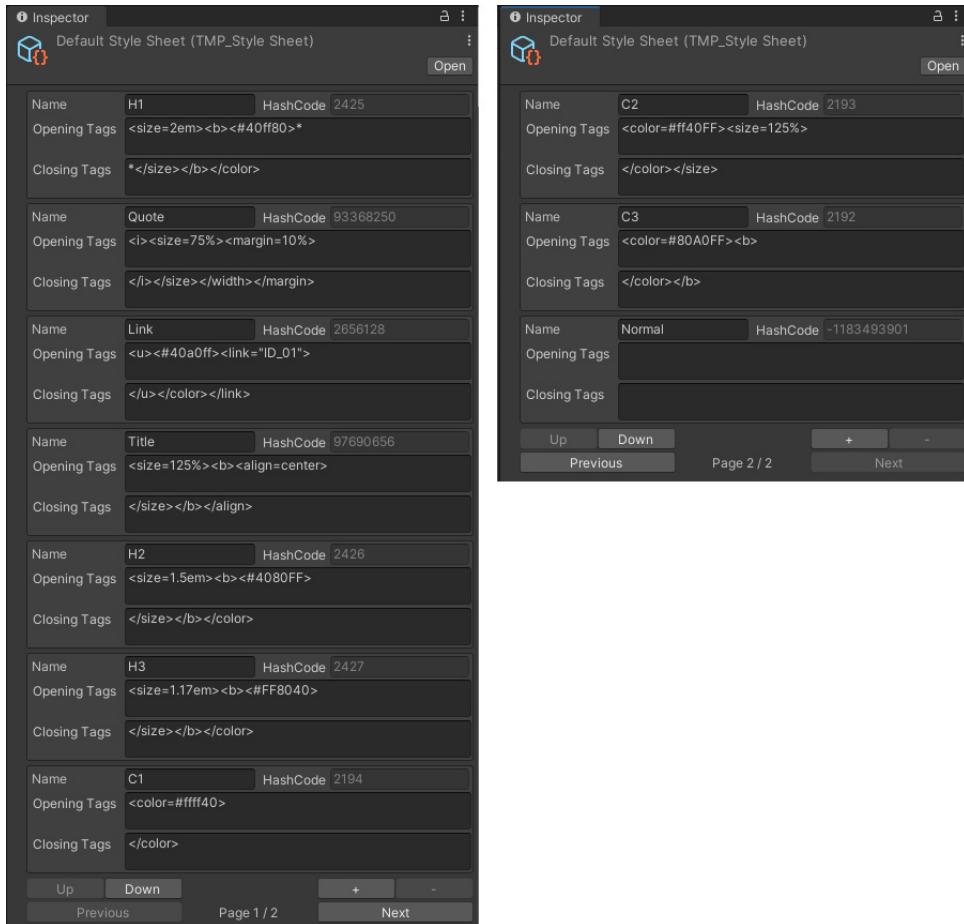


Figure 10.31: The Default Style Sheet properties

You are free to change the names and properties of any of these tags by editing this asset. You can add and remove tags.

You can also create a whole new style sheet and make it the default style sheet. To create a new style sheet, right-click within the **Assets** folder and select **Create | TextMeshPro | Style Sheet**.

You can apply these styles via the dropdown, as I demonstrated at the beginning of this section, or you can assign them to portions of the text via markup format using the `<style>` tag. For example, if you wanted to display the text shown in *Figure 10.32*, you could do so by typing Here's `<style="Title">how</style>` to use styles `<style="Link">in-line</style>`! into the **Text** field:

Here's **how** to use styles **in-line!**

Figure 10.32: Styles used in-line in a markup fashion

Now that we've reviewed the how to format fonts and with markup, let's move on to explore how you can use the various Text component properties, with translations.

Translating text

Odds are, if you are creating a game with text in it, you might want to translate that text. To make sure your games are easily translatable, there are a few key things that you can do to make the transition to different languages easier.

You want to make sure that the text you plan to translate will still fit within the necessary area if it gets longer or shorter when translated. You can accomplish this by using the **Align By Geometry** and **Best Fit** properties of the **Text** component. This will make the text fit within the required space. You could also use the Content Size Fitter (which we discussed in *Chapter 7*) to make sure any Panels around the text will shrink or expand to fit perfectly around the text. You can use the first option if you don't mind the font size varying across languages. You can use the second if you want the font size to remain consistent across languages. Keep in mind that some languages can render a single phrase in a very small amount of space, while others will render it in a very large amount of space.

Use fonts that can support the languages you will be translating to. If possible, a single font that works across all languages will be preferred as it will maintain a consistent style across all translations. You spent so much time picking the perfect font! You don't want all that to be thrown out the window when you translate your game and the font won't render all the glyphs of the language! It's not a glamorous or particularly stylish font, but one that will translate into *many* languages is the Noto Sans font family. If you know you'll be translating into a lot of languages, you might want to consider it: <https://fonts.google.com/noto/fonts>.

If you plan to translate into a language that renders right-to-left, such as Arabic, you will need to use a TextMeshPro object, rather than a Text object, as it will let you render text from RTL.

Note

At the time of writing, TextMeshPro, while able to render text from right-to-left, does not fully support right-to-left languages. If you'd like to display Arabic, Farsi, and/or Hebrew in your game's UI, I recommend the following package: <https://github.com/pnarimani/RTLTMPRO>.

You may also appreciate the following tutorial on rendering right-to-left text: <https://allcorrectgames.com/insights/unity-from-right-to-left/>.

The following is also an extremely helpful resource as it breaks down how to architect a localization solution for your project and also discusses right-to-left text translation: <https://phrase.com/blog/posts/localizing-unity-games-official-localization-package/>.

In the *Examples* section, I provide a small example that focuses on the UI layout and font aspects of translation.

Examples

In this chapter, we'll expand on the scene we've been building further and also add a new scene that occurs between our start screen and our main game screen.

Creating animated text

First, we will create a new scene that acts like a *cut scene* between our start screen and our gameplay scene. It will include our cat introducing itself. The text will animate as if it is being typed, and the user will have the option to speed it up by pressing a button. Once the text is fully displayed, pressing that same button will either show the next block of text or go to the gameplay scene. The text windows will appear as follows:



Figure 10.33: The end result of our animated text box

Let's start by creating a prefab to save us some development time.

Creating a Background Canvas prefab and a new scene

Before we can start making animated text, we need to build out our scene. In both the scenes we have created so far, we used `Background Canvas` to display the background image, and we will use it again in a new scene.

Since we will use this `Background Canvas` multiple times, we should create a `Background Canvas` prefab. As we learned in a previous chapter, a **prefab** is a reusable `GameObject`. Using a prefab in a scene creates an instance of the prefab within the scene. If you make a change to the saved prefab, the change will be reflected in all unbroken prefab instances across all scenes.

To create a reusable Background Canvas prefab GameObject, complete the following steps:

1. Open the Chapter9-Examples scene.
2. Drag the Background Canvas GameObject from the Hierarchy into the Prefabs folder within the **Project** view. The name Background Canvas should now be blue in the **Hierarchy** (symbolizing that it is a prefab).
3. Let's create a new scene in which we will use this Background Canvas prefab. Create a new scene called Chapter10-Examples-IntroScene and save it in the Scenes folder.
4. Drag the Background Canvas .prefab into the new scene from the **Project** view.
5. Assign the Main Camera to the **Render Camera** slot of the **Canvas** component on the Background Canvas and make sure that **Sorting Layer** is set to **Background**.

Now, we can start setting up the windows that will hold our animated text.

Laying out the text box windows

To create the text box windows that will display our text, complete the following steps:

1. Create a new UI Canvas and name it **Text Canvas**. Set its **Canvas** and **Canvas Scalar** properties, as shown here:

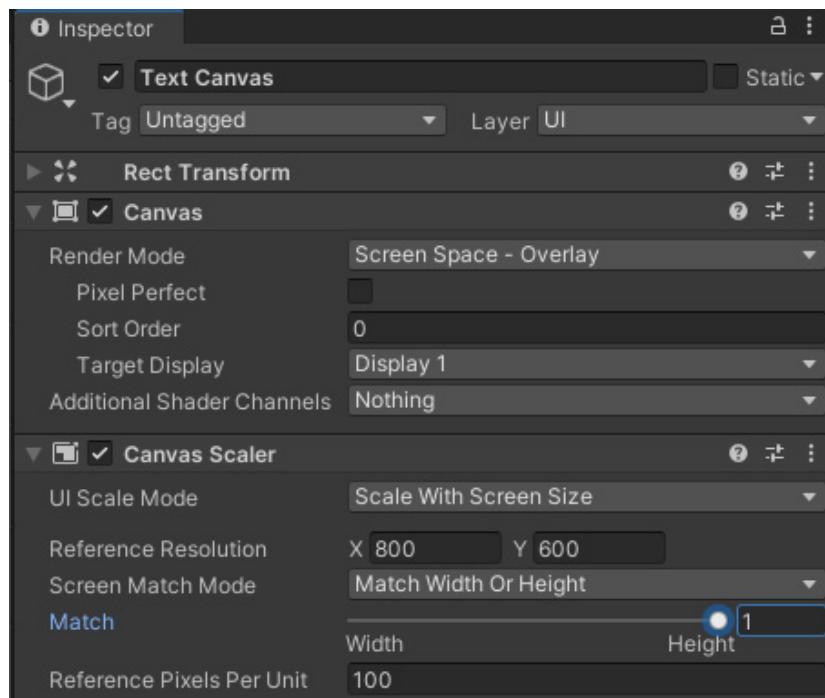


Figure 10.34: The Text Canvas Inspector

2. Create a new UI Image and name it **TextHolder1**. Set its anchor and pivot to **middle center**. Assign **uiElements_10** to the **Source Image**, and then select **Set Native Size** to cause the image's size to be set to 223 x 158.
3. This Panel will hold an image of our cat, some text, and a continue button. Utilizing anchors, pivots, and stretching, lay out the UI objects as children of **TextHolder1** so that they appear as illustrated:

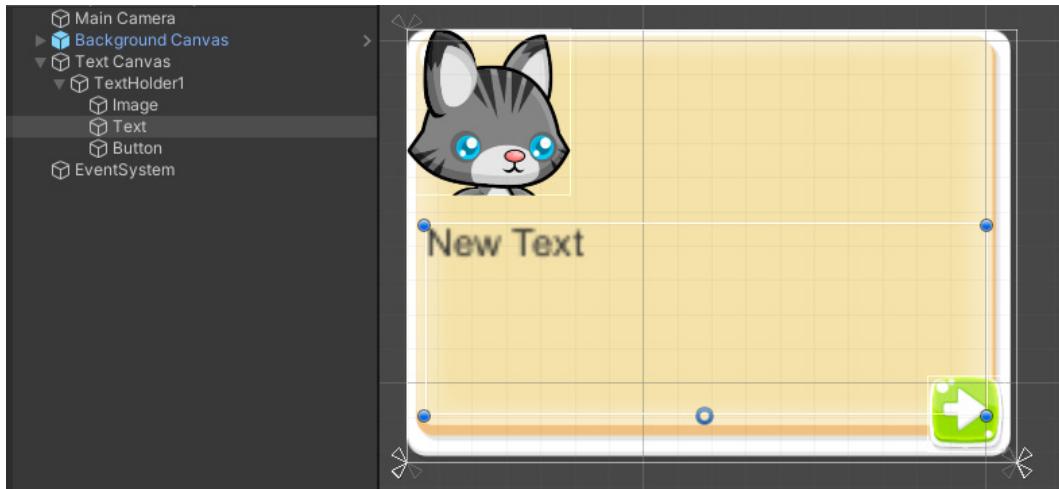


Figure 10.35: TextHolder1 and its children

Note that in the preceding screenshot, the **Text** child's Rect Transform does not stretch all the way across the image of **TextHolder1**. That way, the text won't cross over the white area of the window.

4. We'll want to be able to turn this window on and off, so add a **Canvas Group** component to **TextHolder1**. Leave the settings at their default values.
5. Now, let's change the font. Go to <https://www.dafont.com/milky-coffee.font> and download the font named *Milky Coffee*.
6. Add the *Milky Coffee* font to your Assets/Fonts folder and then drag it into the **Font** setting of the **Text** object's **Text** component.
7. Change the font **Size** to 18.
8. Duplicate **TextHolder1** and name the duplicate **TextHolder2**.
9. Replace the text on the **Text** child of **TextHolder1** with "Hello there!" and the text on the **Text** child of **TextHolder2** with "I'm a cat and, for some reason, I'm collecting food!".

10. Let's hide and disable `TextHolder2` by disabling **Interactable**, disabling **Blocks Raycasts**, and setting the **Alpha** value to 0 on the **Canvas Group** component.

Now, we're ready to start animating our text!

Animating the text box text

Now that we have our layout set up, we can animate our text. To do this, we will need to create a new script. This script will control the animation of the text, as well as load the next scene after all the text has been displayed.

To create animated text that looks like it's typing out, complete the following steps:

1. There are three things we want to group to create the animated text box: the Panel that holds the text, the text object that will display the message, and the string that we want to display. So, to group them all, we will need to create a class. Create a new class called `DialogueBox`.
2. Update the script so that it does not inherit from `MonoBehaviour` and looks as follows:

```
using UnityEngine;
using UnityEngine.UI;

[System.Serializable]
public class DialogueBox
{
    public CanvasGroup textHolder;
    public Text textDisplayBox;
    public string dialogue;
}
```

I've used `[System.Serializable]` so that we will be able to see these values in the Inspector. Remember we need to use the `UnityEngine.UI` namespace whenever we use a `Text` type.

3. Create a new C# script called `DialogueSystem.cs`.
4. We will be writing code that implements scene loading and uses various `System` methods and collections. Therefore, we need to include the following namespaces at the top of our new script:

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
```

5. Now, let's begin our variable declaration. First, we will create a list of `DialogueBox` objects called `dialogueBoxes`:

```
public List<DialogueBox> dialogueBoxes;
```

6. Create a `private` variable that will hold the name of the scene that loads after the text is done animating:

```
[SerializeField] string nextScene;
```

I marked it with the `SerializeField` attribute so that it can be assigned via the Inspector while still being private.

7. Before we proceed, let's assign these variables in the Unity Editor. Attach the `DialogueSystem.cs` script to the `Text Canvas` GameObject. You should see the following:

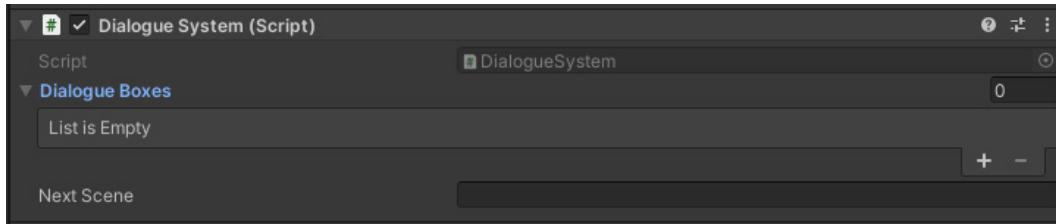


Figure 10.36: The Dialogue System component

8. Press the plus sign at the bottom of the **Dialogue Boxes** list twice. Since we made the `DialogueBox` class serializable, we should see the following:

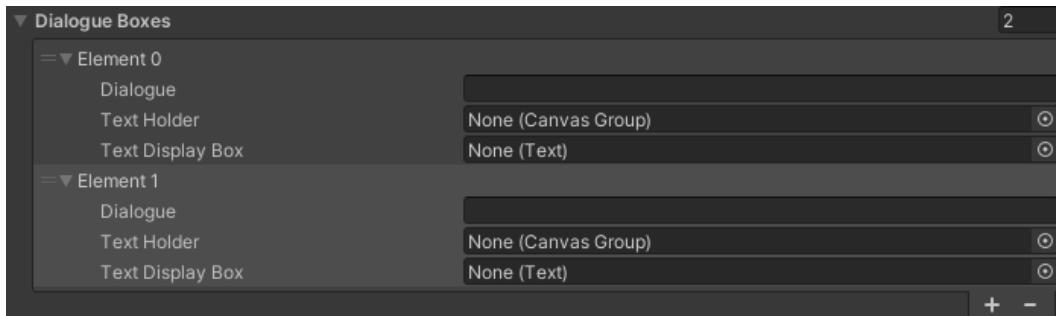


Figure 10.37: The Dialogue System component with two Elements

9. Now, let's drag the appropriate elements into the appropriate fields. Drag `TextHolder1` into **Element 0's Text Holder** and `TextHolder2` into **Element 1's Text Holder**.
10. Drag the `Text` child object of `TextHolder1` into **Element 0's Text Display Box** slot and the `Text` child object of `TextHolder2` into **Element 1's Text Display Box**.

11. Now, update the **Dialogue of Element 0** and **Element 1** to `Hello there!` and `I'm a cat and, for some reason, I'm collecting food!` respectively. We will be generating the text within the text boxes via code. In the previous subsection, we added the text to the text boxes that we placed in the scene. However, due to the code we will write, that step will be rendered unnecessary. It was helpful adding the text to those text boxes, though, because we were able to see exactly how it will display.
12. The last thing we need to do in the Editor, for now, is assign the scene that the game will navigate to once the dialogue has completed. We'll create a scene called `Chapter10-Examples` later, but, for now, let's just assign the `Chapter9-Examples` scene to the **Next Scene** slot by typing `Chapter9-Examples`. Your completed component should look as follows:

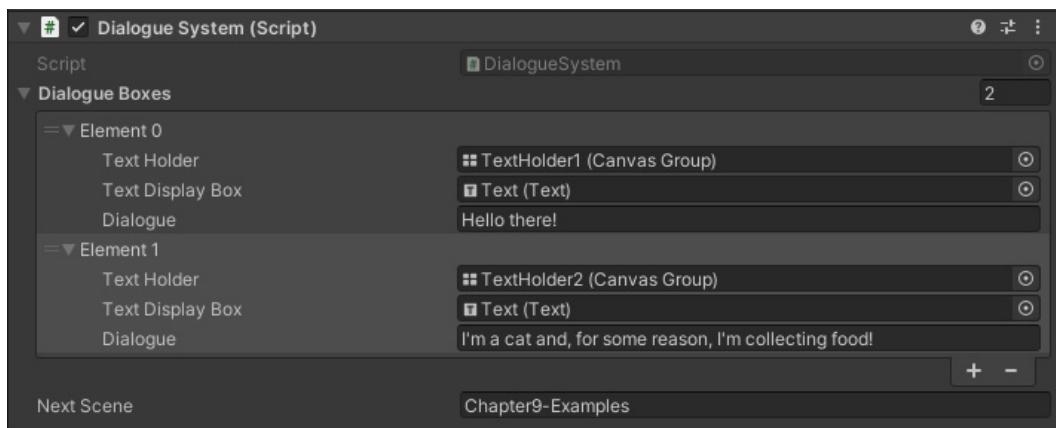


Figure 10.38: The Dialogue System component with all its properties filled out

13. Now, we can go back to our `DialogueSystem.cs` script. We aren't quite done with our variable declaration. We need two variables that will track the string in our dialogue we wish to display, as well as which character within that specific string we are displaying. Add the following variable declaration to your script:

```
int whichText = 0;
int positionInString = 0;
```

Note that these two variables are not public or serialized and thus cannot be adjusted in the Inspector. Their values will be adjusted by the script. The `whichText` variable will allow us to switch between displaying the first string in the dialogue list and the second. The code we will write will easily be extendable to more strings. The `positionInString` variable will keep track of which character is being typed out by the animation. We want to keep track of this so that we can tell whether the text is being sped up by the user or if they have already read the whole text and just want to proceed to the next part.

14. We will use a coroutine to animate our text one letter at a time. Coroutines work very well for timed and scheduled events. The last variable we need to declare will allow us to reference our coroutine so that we can easily stop it. Declare the following variable:

```
Coroutine textPusher;
```

15. The coroutine that will control the text animation is as follows:

```
IEnumerator WriteTheText()
{
    for (int i = 0; i <= dialogueBoxes[whichText].dialogue.Length; i++)
    {
        dialogueBoxes[whichText].textDisplayBox.text =
            dialogueBoxes[whichText].dialogue.Substring(0, i);
        positionInString++;
        yield return new WaitForSeconds(0.1f);
    }
}
```

This code finds the current string in the current dialogue box using the `whichText` variable and loops through all of its characters. With each step of the loop, the `text` property of the UI Text object is updated to display the first `i` characters of the string, where `i` represents the current step of the loop. It then increases the `positionInString` variable and waits a tenth of a second to display the next character by proceeding to the next step in the loop.

16. Before the preceding code will do anything, we need to start our coroutine. In the process of starting it, I also want to assign the `textPusher` variable. Add the following to the `Start()` function:

```
void Start()
{
    textPusher = StartCoroutine(WriteTheText());
}
```

If you play your game now, you should see the *Hello there!* text type out within your scene.

17. Currently, the coroutine only loops through the string in the first DialogueBox. We need to make it proceed to DialogueBox in the list by increasing the `whichText` variable. We also need to add functionality to allow players to show all the text so that they don't have to wait for it to fully animate if they're impatient. Let's create a function that will be called by the press of our buttons. We'll also create a function that compartmentalizes the enabling and disabling of a Canvas Group:

```
public void ProceedText()
{
    if (positionInString < dialogueBoxes[whichText].dialogue.Length)
```

```
{  
    StopCoroutine(textPusher);  
    dialogueBoxes[whichText].textDisplayBox.text =  
    dialogueBoxes[whichText].dialogue;  
    positionInString = dialogueBoxes[whichText].dialogue.  
Length;  
}  
else  
{  
    ToggleCanvasGroup(dialogueBoxes[whichText].textHolder,  
false);  
  
    whichText++;  
  
    if (whichText >= dialogueBoxes.Count)  
    {  
        SceneManager.LoadScene(nextScene);  
    }  
    else  
    {  
        positionInString = 0;  
        ToggleCanvasGroup(dialogueBoxes[whichText].  
textHolder, true);  
        textPusher = StartCoroutine(WriteTheText());  
    }  
}  
}  
  
public void ToggleCanvasGroup(CanvasGroup Panel, bool show)  
{  
    Panel.alpha = show ? 1 : 0;  
    Panel.interactable = show;  
    Panel.blocksRaycasts = show;  
}
```

When a button is pressed, the code first determines whether the whole string has been displayed using the `positionInString` variable. If the `positionInString` variable is smaller than the total characters in the current string, it displays the complete string; otherwise, it proceeds.

When the `positionInString` variable is less than the total characters in the current string, the coroutine is stopped early with `StopCoroutine(textPusher)`.

The `text` property of the `textDisplayBox` is updated to display the full string, and the `positionInString` is set to the length of the string; that way, if the button is clicked again, this function will know that it can proceed to the next step.

When the `positionInString` variable is not less than the total characters in the current string, the current Canvas Group is deactivated, and then the `whichText` variable is increased. Once this variable is increased, the code checks whether any more text boxes need animating. If there are not, the next scene loads. If more text boxes need animating, the `positionInString` variable is reset to 0, so the very first character in the string will be displayed first. The new Canvas Group is now activated, and the `textPusher` variable is reassigned so that the coroutine loop will play again.

18. Now that our code is done, we just need to hook up our buttons to perform the function described in the previous step. For both buttons on both `TextHolder` objects, set the **On Click()** event to run the `ProceedText ()` function in the `DialogueSystem` script attached to the `Text Canvas`. Now, when you play the game, clicking on the button when the text isn't finished typing will cause it to fully display, and clicking on the button when the text has fully displayed will display the next dialogue or the next scene.

To improve on this, you can also create a prefab of the `TextHolder` object and write code that instantiates into the scene based on `Dialogue List`. I recommend implementing this change if you will be making a more complicated dialogue system.

Note

The code example provided in this book's code bundle includes code comments not shown here as it was too cluttered to display in this text.

Translating the dialogue

Let's expand upon our animated text example so that it includes translations. This is a basic example to demonstrate how to access certain properties of Text components and isn't necessarily architected in a way that would be sustainable for a large project.

Note

Please note that I used Google Translate to obtain these translations, so they may not be fully accurate:

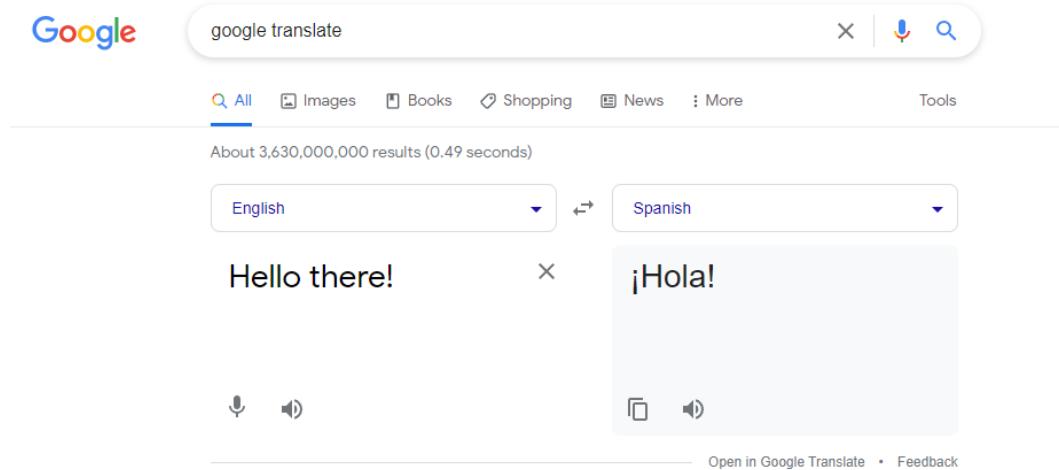


Figure 10.39: Google Translate

To add translation to the animated text example we completed previously, complete the following steps:

1. Open the `DialogueBox.cs` script and add the following namespace to the top of the script:

```
using System.Collections.Generic;
```

2. Let's add a subclass to this script that will group all the translations with the appropriate settings. We'll use this subclass to hold information about not only the translated text but the appropriate font:

```
[System.Serializable]
public class Translation
{
    public string languageKey;
    public string translatedString;
    public Font font;
    public FontStyle fontStyle;
}
```

The `languageKey` string will be used as a key to finding the appropriate translation.

- Now, add a list that will hold all of the translations:

```
public List<Translation> translations;
```

Your DialogueSystem component should now be updated to look like the following:

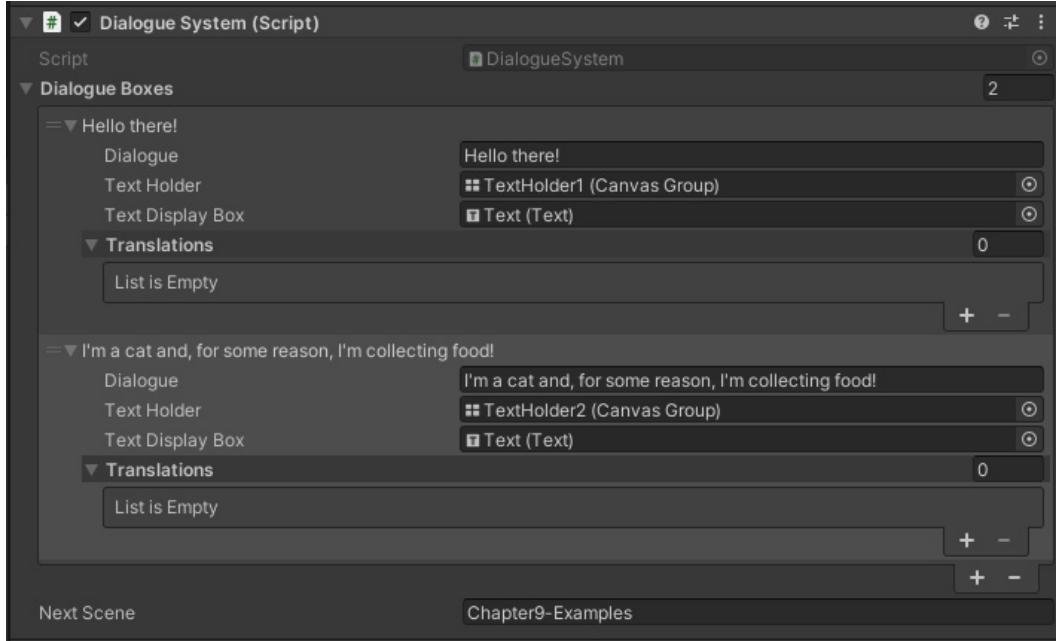


Figure 10.40: The Dialogue System component with translations

- Let's add a few translations to the `translations` list. We'll use the ISO 639-1 two-digit codes as keys for each language. Add the following four key codes to the translation list to indicate Spanish, Japanese, Simplified Chinese, and Korean:

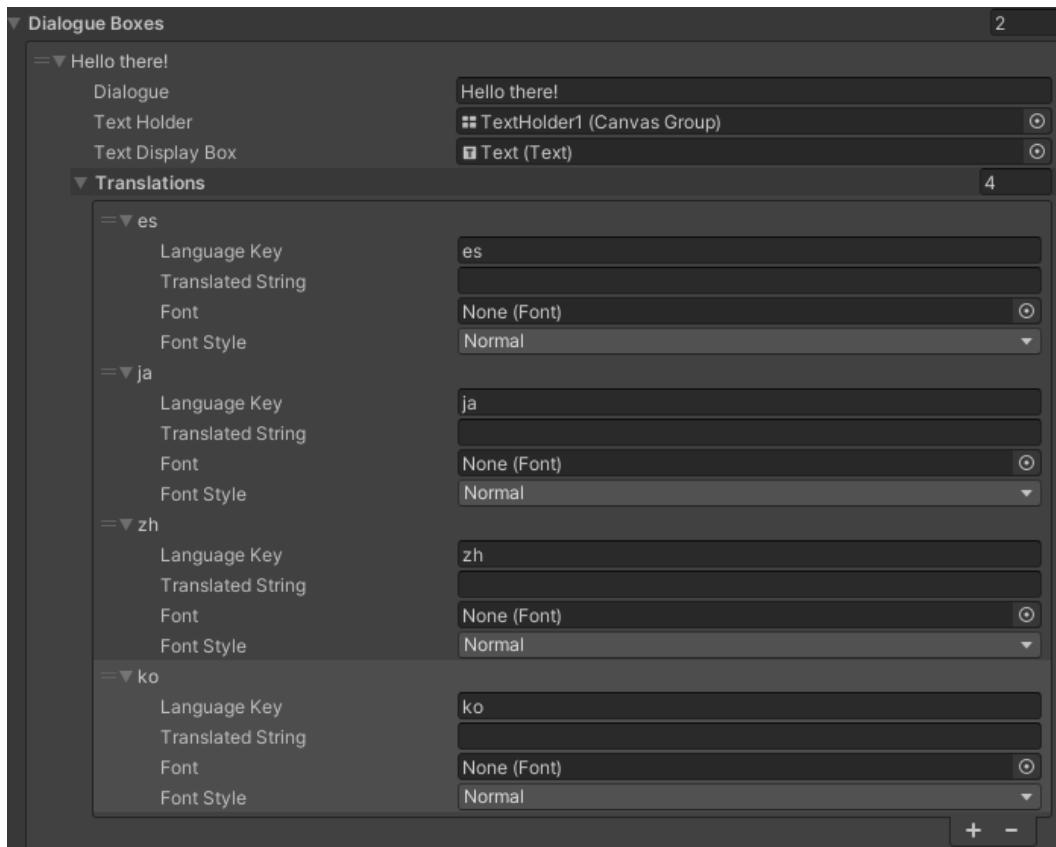


Figure 10.41: The translation keys filled out on the Dialogue Boxes

5. To save yourself time with entering these keys, copy this list by right-clicking on the word **Translations** and selecting **Copy**. Then, under the `I'm a cat and, for some reason, I'm collecting food!` element, paste this list to its **Translations** by right-clicking and selecting **Paste**.
6. Now, let's enter the translations. Enter the following data into the **Translated String** properties:

Key	Hello there!	I'm a cat and, for some reason, I'm collecting food!
es	¡Hola!	¡Soy un gato y, por alguna razón, estoy recolectando comida!
ja	こんにちは！	私は猫で、なぜか食べ物を集めています！
zh	你好呀！	我是一只猫，出于某种原因，我正在收集食物！
ko	안녕!	나는 고양이고, 웬지 모를 음식을 모으고 있다!

Table 10.5: The translation strings to enter

You'll notice that the Unity Engine is capable of rendering these languages in the Inspector.

- We now need to assign some fonts to each language. The current font, Milky Coffee, does not support these four languages. If we try to replace the text with one of these translations, the engine will render any non-supported glyph in a font that does support it but render all supported glyphs (such as punctuation) in the Milky Coffee font. This will result in an inconsistent style that honestly doesn't look great. For example, as shown in the following figure, the comma and exclamation point are not in the same font as the rest of the text:



Figure 10.42: The dialogue box with the Korean text rendering in two fonts

Therefore, we'll want to change the font whenever the translation occurs to one that we specifically choose. I will use the ZCOOL KuaiLe font for the Simplified Chinese translation and RocknRoll One for all the others.

Download the fonts from the following locations and add them to your Assets/Fonts folder:

- <https://fonts.googleapis.com/specimen/ZCOOL+KuaiLe>
- <https://fonts.googleapis.com/specimen/RocknRoll+One>

- Assign the correct fonts to each of the translations. Your two elements in your **Dialogue Boxes** list should look as follows:

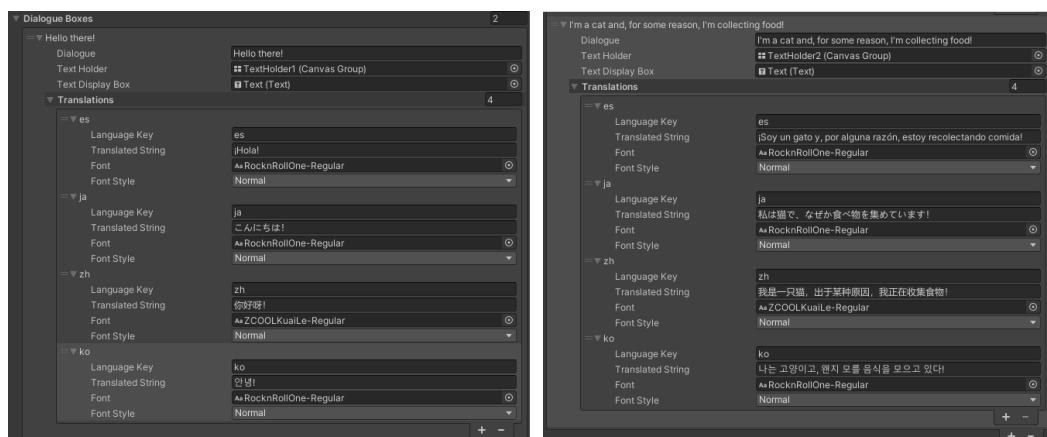


Figure 10.43: The two dialogue boxes with all properties completed

I've included the option to change the font style, but I'm going to leave them all at **Normal** for this example.

9. Let's write the code that will translate our text. Add the following variable declaration to the top of the `DialogueSystem.cs` script so that it will be the first variable declaration. This will be used to determine which language our game should display at runtime.

```
[SerializeField] string currentLanguage;
```

10. Add the following method to the bottom of the script:

```
private void Translate()
{
    foreach (DialogueBox dialogueBox in dialogueBoxes)
    {
        int index = dialogueBox.translations.FindIndex(x =>
x.languageKey == currentLanguage);
        if (index >= 0)
        {
            dialogueBox.dialogue = dialogueBox.
translations[index].translatedString;
            dialogueBox.textDisplayBox.font = dialogueBox.
translations[index].font;
            dialogueBox.textDisplayBox.fontSize = dialogueBox.
translations[index].fontSize;
        }
    }
}
```

This method will find which of the translations have the key designated by the `currentLanguage` variable. It will then change the `dialogue` variable, the `font`, and the `fontSize` to the appropriate values. If the `currentLanguage` variable is not found in any of the `languageKeys`, `index` will equal `-1` and no changes will be implemented.

11. To make sure the translation happens, we need to call the method from the `Awake()` method. Add the following above your `Start()` method:

```
void Awake()
{
    Translate();
}
```

12. Go back to the Editor and enter `es` into the **Current Language** slot. You'll see that the dialogue now translates and changes font when you press play. However, there is a problem displaying the text. You'll notice that the text gets cut off in the second Panel:

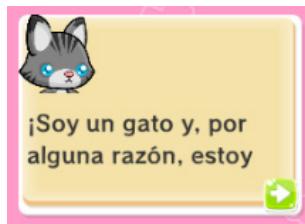


Figure 10.44: The Spanish translation getting cut off

13. Recall, from the *Translating text* section, that you should not only make sure that your font will render the text but that your text boxes will have enough room for the translated text, which may be much longer (especially when rendered in a different font)! The easiest way to fix this is to use the **Best Fit** setting on the **Text** component.

Select both **Text** objects in the Hierarchy and then select the **Best Fit** setting from their **Text** components.

14. After selecting the **Best Fit** property, set the **Max Size** property to 18. This will stop the text from getting too large.

A downside to using **Best Fit** in this example is the font size changes as the text animates. It's not ideal. But, to maintain the text box size, we didn't have much choice – for now! After we learn about Scroll Rects and Masks in *Chapter 12*, we can maintain the font size while scrolling through the text.

Custom font

Let's take a step away from building out our scenes for a moment to explore making a custom font. We won't be using this custom font in the scenes we've been working on, but the process of creating a custom font is still important to cover. We'll create a custom font that displays the numbers 0 through 9 using the following sprites:



Figure 10.45: The custom font we will create

The sprites used to create the font are modified from the free art asset found at <https://opengameart.org/content/shooting-gallery>.

To create an evenly spaced sprite sheet for this font, I used the TexturePacker program, along with Photoshop. The process can be done entirely with a photo editing software such as Photoshop, but TexturePacker simplifies the process. TexturePacker can be found at <https://www.codeandweb.com/texturepacker>.

The process of creating a custom font is time-consuming and kind of a pain. To create a custom font, you have to put in coordinate locations for each character you plan on rendering with the font, so I don't recommend it for anything other than numbers or a very limited character set.

If you want a custom font with a more robust character set, check out the Unity asset store for various options on streamlining the bitmap font process.

To create the custom font displayed in the preceding figure, complete the following steps:

1. Create a new folder called `Custom Fonts` within your `Assets/Fonts` folder.
2. Find the `customFontSpriteSheet.png` file within the code bundle and drag the file into the folder you created in *Step 1*. The sprite sheet appears as follows:



Figure 10.46: The custom sprite sheet

When manually creating a custom font, your characters must be spaced evenly. This will make your life significantly easier while entering the settings of the individual characters. You can leave the sprite sheet's import settings at the defaults of **Sprite (2D and UI) Texture Type** and **Single Sprite Mode**.

3. A custom font requires a material to render. Create a new material by right-clicking in your `Custom Fonts` folder and selecting **Create | Material**. Rename the new material `CustomFontMaterial`.
4. Select `CustomFontMaterial` to bring up its **Inspector**. Drag `customFontSpriteSheet.png` into the square next to **Albedo** in **Main Maps**. Once you do so, the material's preview image should update:

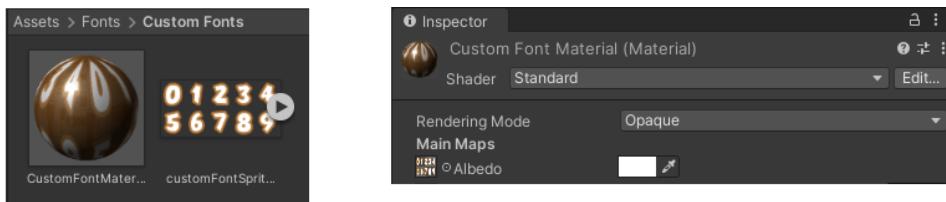


Figure 10.47: The custom font material

5. Now, we need to change the material's shader. There are a few different options you can use for the shader: you can use **UI | Default** or any of the unlit or unlit UI options. My preference is to use **GUI | Text Shader** as I tend to have the best luck with it rendering correctly, and I prefer the way it displays in the **Project** view:

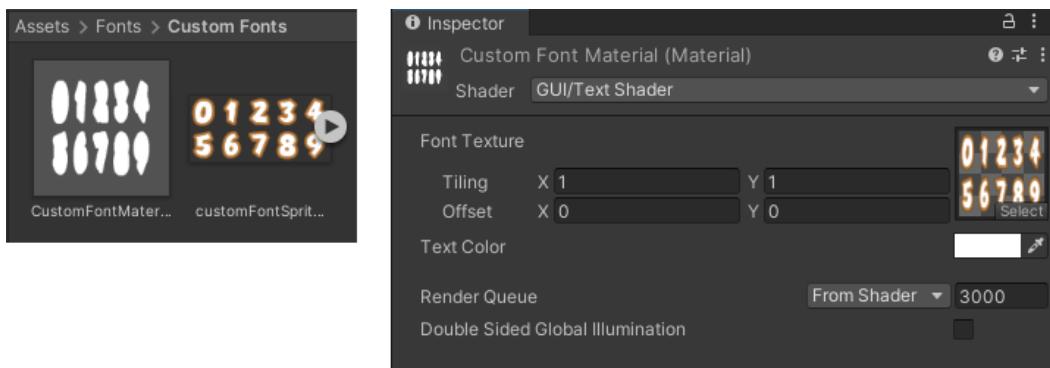


Figure 10.48: The custom font material updated

6. Now, we can create our custom font. Within the **Custom Fonts** folder, right-click and select **Create | Custom Font**. Name the new font **CustomFont**.
7. Select **CustomFont** and assign **CustomFontMaterial** to the **Default Material** slot.
8. The properties under **Character Rects** will specify the coordinates of the individual characters on the sprite sheet. The **Size** property specifies how many total characters are in our sprite sheet. Many of these properties are repeated for each character. To save time, I like to set the properties of **Element 0** before I increase **Character Rects**'s **Size** to the number of characters I wish to display. This way, the properties I specify in **Element 0** will be duplicated to all subsequent Elements. So, set the **Size** property to 1 for now. We will change it to 10 once we have the properties that will repeat for all characters entered. When you set the **Size** to 1, you should see the properties of **Element 0** appear:

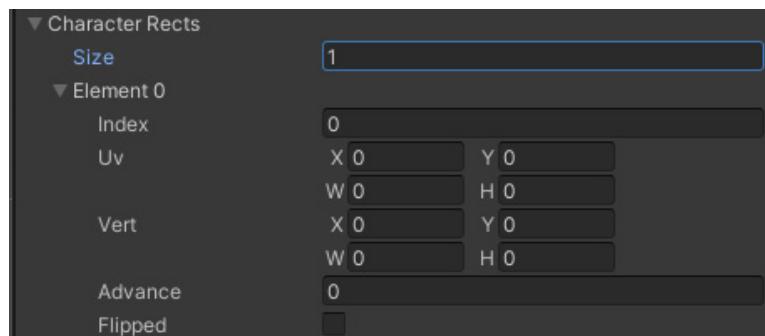


Figure 10.49: The Character Rects of the custom font

9. The first properties that repeat for all characters are **UV W** and **UV H**. These values represent the percentage of the sprite's total width and total height that the individual character takes up. Since our characters are all evenly spaced in our sprite sheet, these values will be the same for all characters. If your characters are evenly sized, you can calculate these values as follows:

$$\text{UV W} = \frac{1}{\text{columnCount}}$$

$$\text{UV H} = \frac{1}{\text{rowCount}}$$

Figure 10.50: Calculating the UV W and UV H values

There are a total of five columns of characters. Therefore, each sprite takes up one-fifth of the width of the sprite. One-fifth is equal to 20% or 0.2 as a decimal. We need to put 0.2 in the **UV W** slot. If you're not great with percentages, that's fine. Unity will perform the calculation for you! Typing 1/5 in the **UV W** slot and pressing *Enter* will automatically compute the 0.2 decimal.

There are a total of two rows of characters. Therefore, each sprite takes up one-half of the height of the sprite; one-half is equal to 50% or 0.5 as a decimal. Typing 1/2 in the **UV H** slot and pressing *Enter* will automatically compute the correct decimal of 0.5 in the slot.

10. The next values that repeat for each character are the **Vert W** and **Vert H** values. When the sprite sheet was created, each individual sprite was made from a 50x57 pixel image. So, type 50 in the **W** slot and -57 in the **H** slot. Remember that the **H** value will always be negative!
11. The last property that remains consistent throughout all the characters is the **Advance** property. This property is the space between the characters. Since our sprites have a width of 50, we should make this property 50 or larger. You can fiddle with this property to see what looks best to you, but I think it looks nice at 51.

After completing Steps 9 through 11, your **Element 0** character should have the following properties:

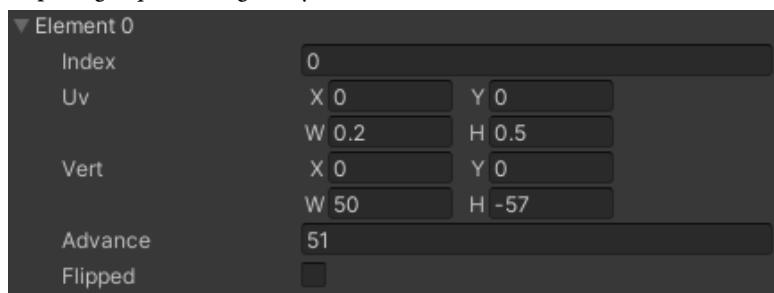


Figure 10.51: Element 0 with its values filled in

12. Now that we have placed all the repeated properties, we can increase the **Size** of our **Character Rects** set. There are a total of 10 characters in our sprite sheet, so change the **Size** property to 10. You'll see that once we do this, the properties of **Element 0** are repeated in **Element 1** through **Element 9**:



Figure 10.52: Element 0 duplicated through Element 10

13. Now, we need to specify the ASCII index of each element. This will tie the typed text to the correct sprite. Without this, the font wouldn't know that typing the number 0 should show the first sprite in the sheet. The following table indicates that the numbers 0 to 9 are the ASCII numbers 48 to 57: <http://www.ascii.cl/>.

Therefore, we should enter the **Index** values of 48 through 57 in **Element 0** through **Element 9**:

Element	0	1	2	3	4	5	6	7	8	9
Index	48	49	50	51	52	53	54	55	56	57

Table 10.6: The index values of each element

14. The next step is to specify the **UV X** and **Y** values. This is the part of creating custom fonts that takes up the most time. These values represent the UV coordinate position of the characters in the sprite sheet. These numbers are calculated based on the row and column numbers that the character lies in. The row and column numbers start at 0 and start in the bottom-left corner:

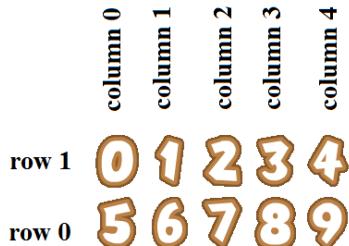


Figure 10.53: The rows and columns of the sprite sheet

To calculate the **UV X** and **UV Y** values, use the following formulas:

$$\text{UV X} = \text{columnNumber} \times \text{UV W}$$

$$\text{UV Y} = \text{rowNumber} \times \text{UV H}$$

Figure 10.54: Calculating UV X and UV Y

Remember that since our characters are evenly spaced, our **UV W** and **UV H** values are the same for each character.

So, if we look at **Element 0**, its **UV X** value can be found by multiplying 0 (for column 0) by 0.2 (the **UV W** value) to get 0. Its **UV Y** value can be found by multiplying 1 (for row 1) by 0.5 (the **UV H** value) to get 0.5.

The following chart represents the **UV X** and **UV Y** values that should be entered for each character:

Element	UV X	UV Y
0	0	0.5
1	0.2	0.5
2	0.4	0.5
3	0.6	0.5
4	0.8	0.5
5	0	0
6	0.2	0
7	0.4	0
8	0.6	0
9	0.8	0

Table 10.7: The UV X and UV Y properties of the font

The following figure provides a more visual representation of the coordinate pattern:

(UV X, UV Y)

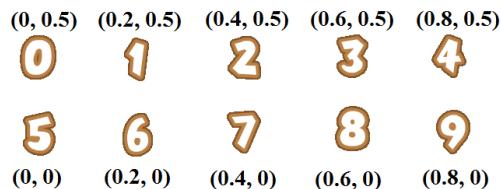


Figure 10.55: The coordinates of the sprite sheet

- Now, we can test our font to see whether it works. Within any of your scenes, create a new **UI Text** object named **Custom Font Text**. I have created a new scene called **Chapter10-Examples-CustomFont** on which I have my test fonts. Change the **Text** to **0123456789** and drag the **CustomFont** into the **Font** slot. You should see something like the following:



Figure 10.56: The custom font as it should be displayed after Step 15

- To make it display correctly, expand the size of the text box to accommodate all the characters and change the text color to white:



Figure 10.57: The custom font after adjusting some properties

Now that we've completed importing our custom font, let's look at adjusting it further.

Adjusting the character spacing and changing the font size

In our example, all the numbers are evenly spaced and the font size cannot be changed. You will likely want your numbers to be closer together or of a different font size. Let's alter our custom font so that the sprites are closer together. The following figure shows our font after the adjustments versus the original we created in the last part of this example:



Figure 10.58: The custom font with and without spacing adjustment

To change the spacing of the characters, complete the following steps:

1. Duplicate the `CustomFont` with `Ctrl + D` and rename the duplicate `CustomFontTight`.
2. Open the Inspector of `CustomFontTight`.
3. If you'd like the numbers to be closer together, you simply have to change the **Advance** property for the specific elements. The **Advance** property represents the pixels from the start of the sprite to the start of the next sprite. So, if we wanted the number 2 to appear closer to the number 1, we could change the **Advance** property of **Element 1** to something smaller, as shown here:

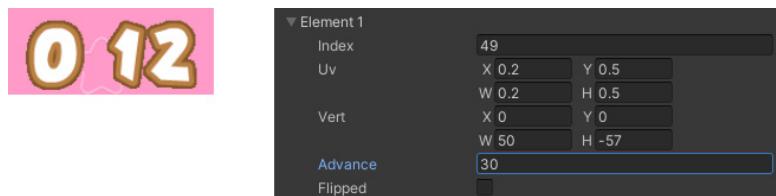


Figure 10.59: The 1 glyph spacing problem

Note

You can change the settings on your custom font while previewing the results in the scene. To get the text to refresh in the scene after making a change to your custom font, you have to save the scene first.

4. Adjust the **Advance** properties of each element based on the following chart:

Element	0	1	2	3	4	5	6	7	8	9
Advance	38	30	35	35	35	35	35	35	38	35

Table 10.8: Advance properties of the custom font

5. The font should now appear as follows:



Figure 10.60: All the glyphs of the font displayed

6. There is still a bit too much spacing between the 0 and the 1; however, if you try to reduce the **Advance** property on the 0 to bring the 1 closer, other numbers will overlap the 0 if they follow the 0. The following example shows what will happen if the **Advance** property of **Element 0** were reduced to 35; the 1 looks good following 0, but the 9 overlaps it:



Figure 10.61: Different Advance settings on the 0 glyph

7. Keeping that in mind, we need to bring 1 closer in our scenario. To move the 1 closer, we need to change the **Vert X** property on **Element 1**. Change the **Vert X** property on **Element 1** to -3 to shift it left just a smidge. This will give the character a more favorable spacing.

Now, if you'd like to adjust the font size, you cannot change the size of a custom font by changing the **Font Size** property of the Text component; changing the **Font Size** will do nothing. To change the size of the font, you must change the **Scale X** and **Scale Y** properties of the Rect Transform component. To get the font half the size, change **Scale X** and **Scale Y** to 0.5.

As I mentioned earlier, we probably won't be using this font to build our master example scene, but the process of creating a custom font is still a useful exercise. Now, let's move on to creating some other common UI assets – health bars and progress bars.

TextMeshPro - Warped Text with Gradient

The banner of our Pause Panel looks a bit bare currently. Now that we've covered using TextMeshPro - Text, we can create a nice curved text with a gradient that lines up well with our banner:

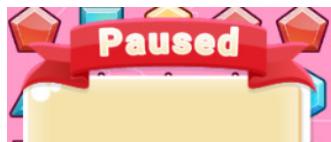


Figure 10.62: The banner text with a gradient and warp

To create the curved text shown in the preceding screenshot, complete the following steps:

1. Duplicate the Chapter 9 - Examples scene and name the new scene Chapter 10 - Examples.
2. Select the Pause Banner UI Image and give it a child Text (TMP) object by right-clicking on it in the Hierarchy and selecting **UI | Text - TextMeshPro**. Rename the child object Paused TMP.
3. If you have not done so already, select **Import TMP Essentials** and **Import TMP Examples & Extras** when the popup prompts you to do so. If you previously only imported TMP Essentials, but not the examples, go to **File | Project Settings | TextMeshPro** and select **Import TMP Examples & Extras**.
4. In the **TextMeshPro - Text (UI)** component, change the **Text Input Box** setting to the word Paused.
5. Set the **Font Style** to **Bold**.
6. Center-align the text horizontally and vertically.
7. Set the **Font Size** to 43.
8. Change the font to Roboto-Bold. (Note this **Font Asset** will only be available if you imported the TMP Examples.)
9. Adjust the Rect Transform so that the text is centered more within the banner. Your text should appear as follows:



Figure 10.63: The banner text's placement

10. Now, let's give the text a gradient fill. Select the checkbox next to **Color Gradient** in the **TextMeshPro - Text (UI)** component.
11. To achieve the desired look, we will leave the top-left and top-right colors white. Select the white rectangle at the bottom left to bring up the Color picker. Select the eye dropper at the top of the Color picker and then move your mouse over the tan area of the Pause Panel image. When you close the Color picker window, the tan color will be in the bottom-left slot.

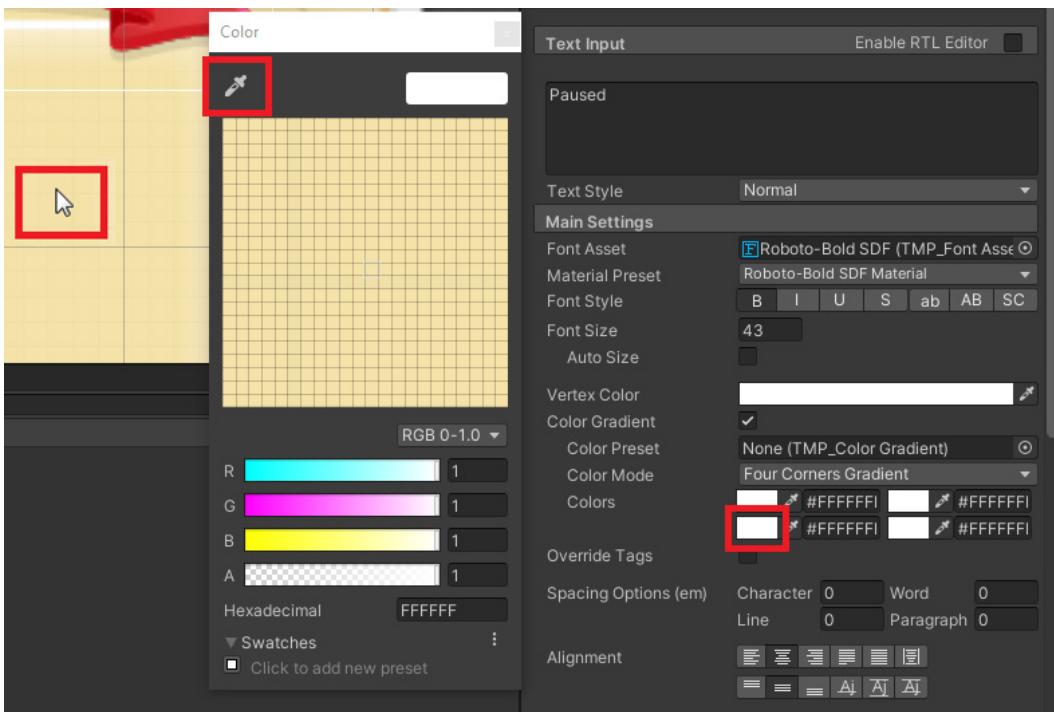


Figure 10.64: Using the eye dropper tool to get the text color

12. Right-click on the color in the bottom-left slot and select **Copy**.
13. Right-click on the white square in the bottom-right slot and select **Paste**. Now, the two top colors should be white and the two bottom colors should be tan.



Figure 10.65: The Color Gradient properties completed

14. In the **Outline** settings (at the bottom, outside of the **TextMeshPro - Text (UI)** component), select the checkbox next to the word **Outline** to enable the outline. Set the **Color** to white and change the **Thickness** to **0 . 25**.
15. The last thing to do is curve the text. TextMeshPro has made this pretty easy for us by providing an example script that curves text at runtime. To view the changes, you have to play the game, and they are not represented in the Scene view. Select the **Add Component** button and choose **Scripts | TMPro.Examples | Warp Text Example**.

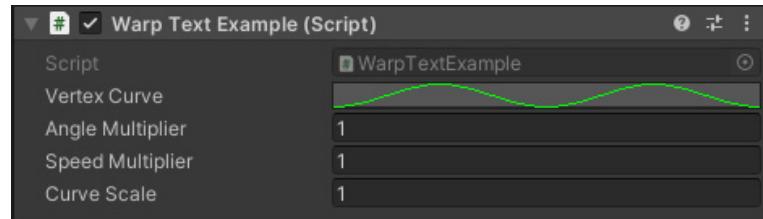


Figure 10.66: The Warp Text Example component

16. Select the wavy green line in the **Vertex Curve** slot to bring up the curve editor. Select the option on the far left – that is, the flat line.

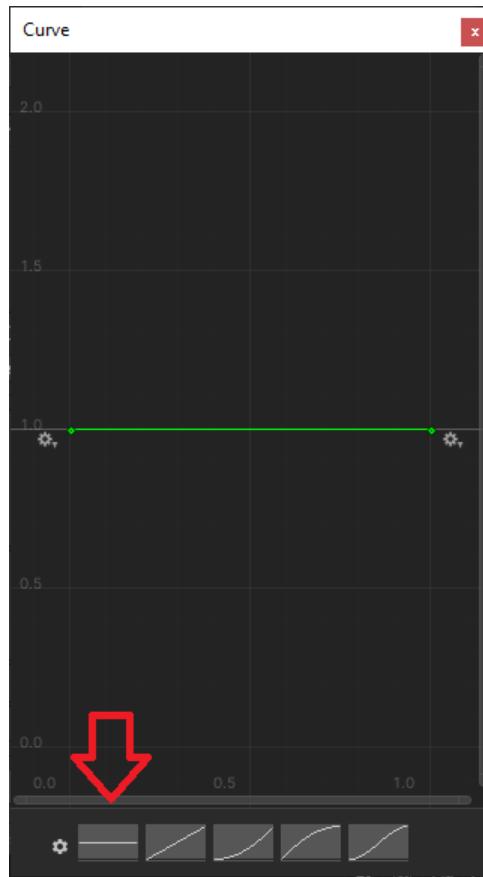


Figure 10.67: Selecting a flat curve for the Vertex Curve

17. Right-click on the green line at the 0.5 mark and select **Add Key**.
18. Select that new key and drag it upward to a little below 1.3.

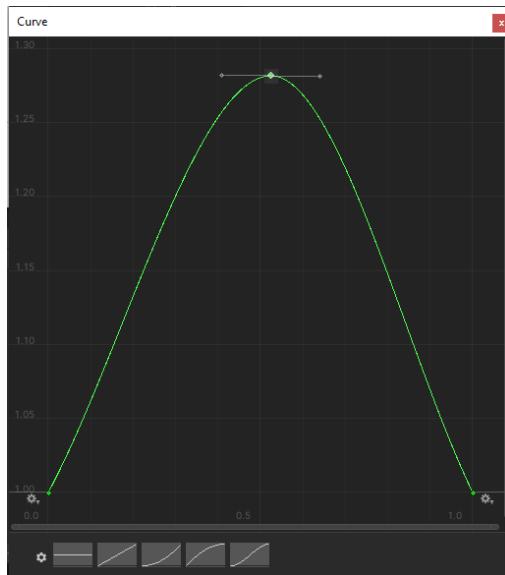


Figure 10.68: The final version of the Vertex Curve

19. Play the game from this scene and press the **P** key to view the **Pause Panel**. The text should now appear as it did at the beginning of this example.
20. To view the entire flow, Start Screen, Intro Scene, and this scene with the updated Pause Panel, we'll need to update some old scenes. Open **Chapter10-Examples-IntroScene** and change the **Next Scene** on the **Text Canvas' Dialogue System** component from **Chapter9-Examples** to **Chapter10-Examples**.
21. Duplicate the scene called **Chapter9-Examples-StartScreen** and name it **Chapter10-Examples-StartScreen**.
22. Open the new scene and select the **Play Button** child of the **Button Canvas**.
23. Change the **Scene To Load** on the **Level Loader** component to **Chapter10-Examples-IntroScene**.
24. Open the **Build Settings** and update the **Scenes in Build** to the following:

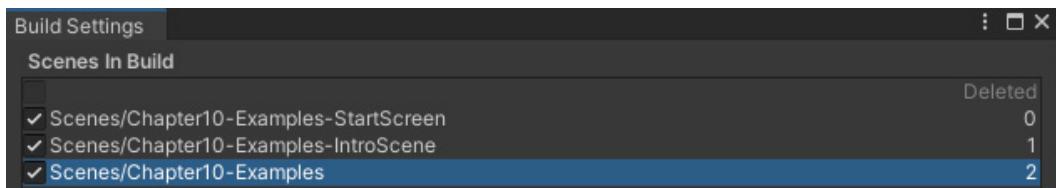


Figure 10.69: The Build Settings with all the appropriate scenes

You can remove unnecessary scenes by right-clicking on them and removing them.

25. With the Chapter10-Examples-StartScreen scene open, press the play button in the Editor and watch the game's flow complete.

Note

I'd like to point out that, while I am duplicating and renaming each scene with each new chapter, you do not have to do so. I am doing it to maintain an easy-to-view progression log of what is happening to our scenes for this book, but admittedly, your project is probably getting a bit cluttered if you've been doing it, too. So, if you are wondering "Why can't I just keep adding to my scenes instead of starting a new one each time?" you can!

And that's it for creating a wrapped text using TextMeshPro!

Summary

Wow! I bet you thought, *how much can you really say about text?* when this chapter started and didn't expect to be faced with the longest chapter so far! And I almost made it longer by adding more examples! Alas, as much as I would like to provide even more examples, I have a page limit I have to adhere to – even though I have already blown past it. If you are hankering for even more examples concerning UI Text and Text-TextMeshPro, I strongly recommend you review the various examples linked within this chapter, as well as the example scenes that you downloaded with the TextMesh – Pro examples.

In the next chapter, we'll do a deep dive into UI Images and Effects.

11

UI Images and Effects

We've worked with UI Images in the previous chapters, but now we'll learn more about the component's specific properties, as well as how to access the component via code. We'll also look at some of the UI effect components that we can apply to our UI objects for visual appeal. While we will look at the components thoroughly, the majority of this chapter focuses on specific worked-out examples of UI functionality that you will find in video games, particularly mobile video games.

In this chapter, we will discuss the following topics:

- Creating UI Images and setting their properties
- Using the various UI effects components to further customize our graphical UI
- Implementing horizontal and circular progress bars
- How to create Buttons that swap sprites without using the built-in transitions, like a mute/unmute Button
- Adding a press-and-hold/long-press functionality
- Creating an onscreen four-directional virtual D-Pad
- Creating a floating eight-directional virtual analog stick

Note

All the examples shown in the sections before the *Examples* section can be found within the Unity project provided in the code bundle. They can be found within the scene labeled Chapter11.

Each example figure has a caption stating the example name within the scene.

In the scene, each example is on its own Canvas, and some of the Canvases are deactivated. To view an example on a deactivated Canvas, simply select the checkbox next to the Canvas' name in the Inspector. Each Canvas is also given its own Event System. This will cause errors if you have more than one Canvas activated at a time.

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2011>

UI Image component properties

We've created a UI Image before, but let's look at its properties and components.

You can create a new UI Image object using + | UI | **Image**.

The UI Image object contains the **Rect Transform** and **Canvas Renderer** components as well as the **Image** component. We've looked at the **Rect Transform** and **Canvas Renderer** components extensively; now, let's look at the **Image** component.

The first setting on the **Image** component is the **Source Image** property, which represents the sprite that will be rendered. The **Color** property represents the base color of the sprite being rendered. Leaving the color at white will make the Image appear exactly as the sprite, but changing the color will add a tinted color overlay to the Image. You can also change the transparency of the Image by reducing the alpha value. The **Material** property allows you to add a material to the Image.

The **Raycast Target** and **Raycast Padding** properties work the same way they do on the **Text** component, by specifying whether the Image will block clicks on UI objects behind it or not and if there is any padding to the block. The **Maskable** property determines if the Image can be affected by masks or not.

When a sprite is assigned to the **Source Image** slot, new options appear in the **Image** component under **Image Type**, as shown in the following figure:

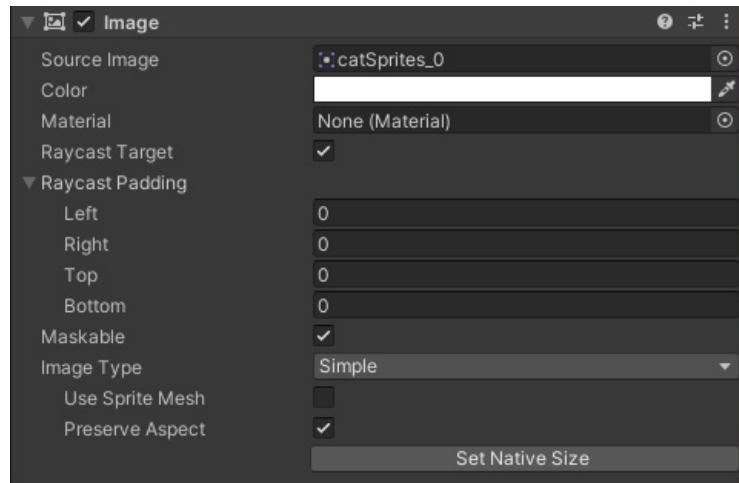


Figure 11.1: The UI Image component and all its properties

Let's look at the various options for **Image Type** and how they affect a sprite.

Image Type

The **Image Type** property determines how the sprite specified by **Source Image** will appear. There are four options: **Simple**, **Sliced**, **Tiled**, and **Filled**. Let's take a look at them.

Simple

An Image with its **Image Type** property set to **Simple** scales evenly across the sprite. This is the default type. When **Simple** is selected as the **Image Type**, a toggle labeled **Use Sprite Mesh**, a toggle labeled **Preserve Aspect**, and a button labeled **Set Native Size** become available.

Selecting the **Use Sprite Mesh** toggle will have the Image use the sprite mesh created by the **TextureImporter**. By default, this property is deselected, and the sprite's mesh is a quad. You will select this property if you don't want the Image represented by a rectangle, but instead want it to have a mesh that fits tightly around the visible area of the Image.

When the **Preserve Aspect** property is checked, the sprite will display with its portions preserved and may not appear to fill the entire area of the Rect Transform. Selecting this property ensures that your sprites look as originally intended and are not stretched out.

Selecting the **Set Native Size** button sets the dimensions of the Image to the pixel dimensions of the sprite.

Sliced

Sliced Images are split into nine areas. When a **Sliced** Image is scaled, all areas of the Image are scaled, except the corners. This allows you to scale an Image without distorting its corners. This works particularly well with sprites that have rounded corners that you want to be able to stretch into rounded rectangles.

When an image is set to **Sliced**, the **Fill Center** and **Pixels Per Unit Multiplier** properties appear. The following figure shows a rounded rectangle with five alternate versions of it being stretched. You can see how selecting **Sliced** allows the rounded rectangle to stretch in a way that maintains the rounded rectangle shape while leaving the **Image Type** at **Simple** causes a distortion in the edges of the Image when it is scaled.

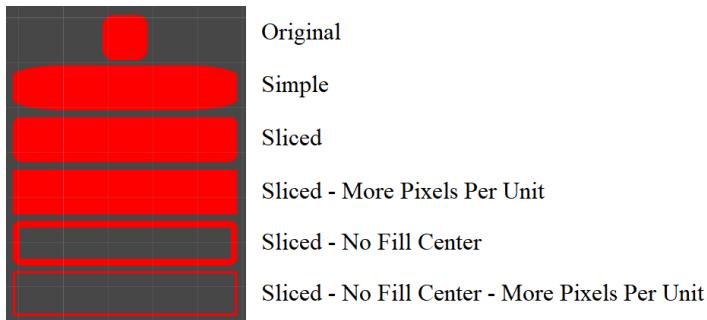


Figure 11.2: Sliced Image Type Example in the Chapter11 scene

You must specify where the nine areas will be, within the **Sprite Editor** of the sprite or sprite sheet. If you have not specified the regions, a message will appear within the **Image** component.

To specify the area in the **Sprite Editor**, you need to drag the green boxes on the edges of the sprite to the desired position. As you can see from the following screenshot, you want to drag the green lines so that they stop surrounding the curves of the edges:

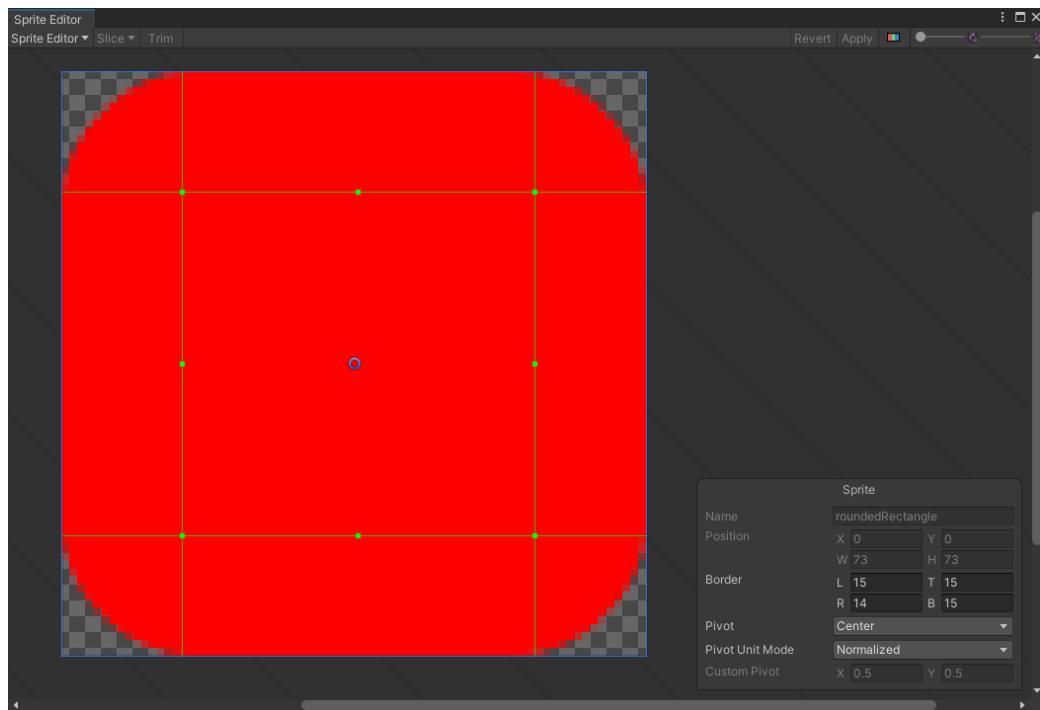


Figure 11.3: Specifying the nine areas of a sprite in the Sprite Editor

Next, let's talk about the **Tiled** option under the **Image Type**.

Tiled

Selecting **Tiled** for **Image Type** will cause the Image to repeat to fill the stretched area. The following figure demonstrates how selecting **Simple** and **Tiled** for **Image Type** affects the scaled Images:

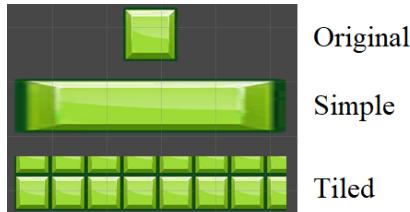


Figure 11.4: Tiled Image Type Example in the Chapter11 scene

Next, let's talk about the **Filled** option under the **Image Type**.

Filled

Images with **Filled** selected for their **Image Type** will fill in a percentage of the sprite, starting at an origin in a specified direction. Any part of the sprite past the designated percentage will not be rendered. When **Filled** is selected, new properties are displayed:

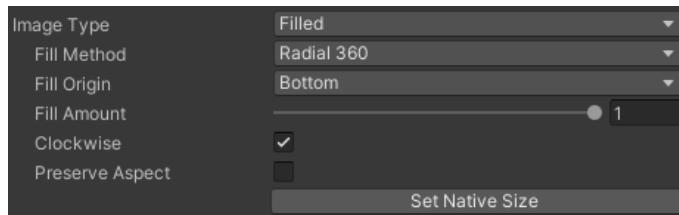


Figure 11.5: Properties for a Filled Image

The **Fill Method** property determines whether the sprite will be filled horizontally, vertically, or radially. There are five options: **Horizontal**, **Vertical**, **Radial 90**, **Radial 180**, and **Radial 360**. Each of these **Fill Method** options will begin drawing the sprite at the **Fill Origin** up to the **Fill Amount**. When one of the radial methods is selected, you can also select the option to have the fill progress **Clockwise**; if you choose not to select this option, the Image will fill counterclockwise. The following figure demonstrates the three **Fill Method** options, all with **Fill Amount** values of 0 . 75 or 75 percent:

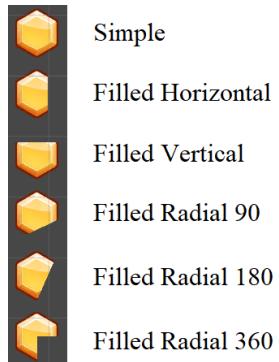


Figure 11.6: Filled Image Type Example in the Chapter11 scene

The **Horizontal** and **Vertical Fill Method** options are somewhat self-explanatory when you see them in action, but it's a little more difficult to determine exactly how the three radial methods work just from looking at them. **Radial 90** places the center of the radial at one of the corners, **Radial 180** places the center of the radial at one of the edges, and **Radial 360** places the center of the radial in the center of the sprite.

The **Filled Image Type** option also has the **Set Native Size** property.

Now that we've explored the UI Image component, we can look at some UI effect components.

UI effect components

Three effects components allow you to add special effects to your Text and Image objects: **Shadow**, **Outline**, and **Position as UV1**. They can all be found under **Add Component | UI | Effects**. Let's look at each one individually, starting with the Shadow component.

Shadow

The **Shadow** component adds a simple shadow to your Text or Image object.

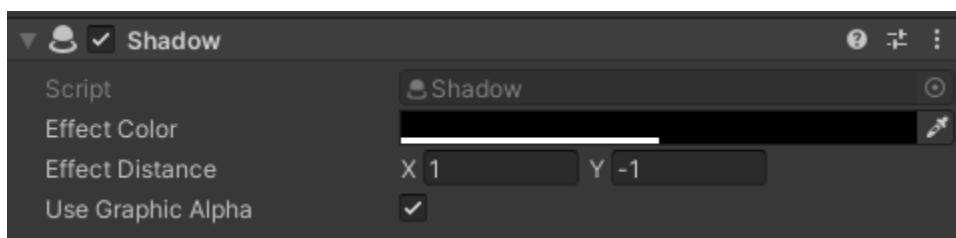


Figure 11.7: The Shadow component

You can change the color and transparency of the shadow with the **Effect Color** property. The **Effect Distance** property determines its position relative to the graphic to which it is attached. The **Use Graphic Alpha** property will multiply the color of the shadow with the color of the graphic on which the shadow is attached. So, if this property is checked and the alpha (opacity) of the original graphic is reduced, the alpha of the shadow will reduce as well, with the resulting shadow being a product of the two alpha values. However, if the **Use Graphic Alpha** property is unchecked, the shadow will maintain its alpha value regardless of the alpha of the original graphic. So, if you turned the alpha of the original graphics all the way down to 0, rendering it invisible, the shadow would remain visible based on the alpha specified on the **Effect Color** property.

The following figure shows a few examples of the **Shadow** component in action:



Figure 11.8: Shadow Component Example in the Chapter11 scene

All four bananas have the same alpha value set on their **Shadow** component's **Effect Color** property. The **Image** component's **Color** property of the first banana has the alpha set to full opacity. The second, third, and fourth bananas have the opacity of their **Image** component reduced. The second and third bananas have identical properties, except that the second banana uses the **Use Graphic Alpha** property and the third does not. So, you can see that the shadow of the third banana has not been dimmed by the dimming of the banana's **Image** component. The fourth and final banana has its **Image** component's opacity set to 0, but since **Use Graphic Alpha** is not selected on the **Shadow** component, the shadow did not dim with the banana and remains at its designated alpha value.

Outline

The **Outline** component simulates an outline around the graphic by creating four shadows around it at specified distances.

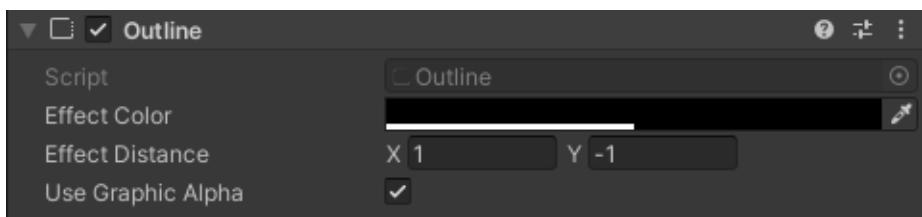


Figure 11.9: The Outline component

The **Outline** component will create two shadows to the left and right of the original graphic based on **Effect Distance X** and two shadows to the top and bottom of the original graphic based on **Effect Distance Y**. Unlike the **Shadow** component, there is no difference in a negative or positive value for these two distances because the two shadows created for each axis are mirrored.

Setting the **Effect Distance X** value to `-3` essentially just switches the positions of the two horizontal shadows, but the effect looks the same, as shown in the following figure:

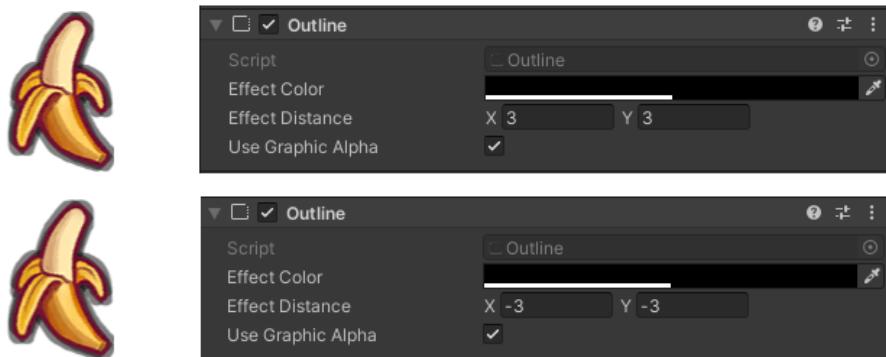


Figure 11.10: Outline Component Example 1 in the Chapter11 scene

The **Use Graphic Alpha** property works identically on this component as it does on the **Shadow** component, as shown:



Figure 11.11: Outline Component Example 2 in the Chapter11 scene

Next, let's look at the **Position As UV1** component.

Position As UV1

The **Position As UV1** component allows you to change the UV channel that the Canvas renders on. This is used if you want to create custom shaders that utilize baked light maps.

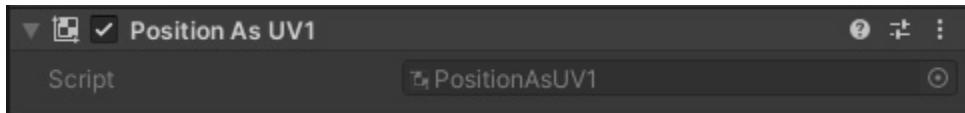


Figure 11.12: The Position As UV1 component

Sadly, custom shaders are a pretty heavy topic and go past the scope of this text, so I won't go any further into the usage of this component.

Now that we've reviewed the UI Image component and some UI effect components, let's look at some examples of ways we can use these components.

Examples

In this chapter, we'll expand on the scene we've been building further by adding some new UI elements. We'll also look at some mobile/touchscreen UI and interactions.

Some of these examples may seem better suited for the chapter on Buttons, but since they include access to the Image component's properties, I placed them here.

Note

We have created two scenes that load into the scene we've been building upon: a start screen and an intro scene. Since I've been duplicating our main scene to make progress from each chapter easy to track, our intro sScene will not navigate to the updates we make in this and future chapters unless we keep updating the **Next Scene** variable on our **Dialogue Boxes** component in the intro sScene and including the new scene in our **Build Settings**.

I will not be including this update in the steps since scene navigation is no longer a focus of these examples. However, it will be included in the packages I include in each chapter's completed scenes.

Horizontal and circular health/progress meters

Let's get back to our main scene. Duplicate the Chapter10-Examples scene to create a Chapter11-Examples scene.

In this section, we'll cover how to create two types of progress meters, a horizontal one and a circular one, as shown in the following screenshot:

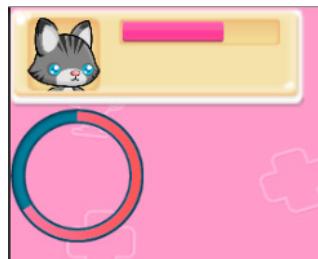


Figure 11.13: Example of horizontal and vertical progress bars

We'll hook up the circular and horizontal progress meters so that they both display the progress of the same variable, and we can watch them both change at the same time.

The circular progress meter doesn't really fit in the main scene that we've been building, and we'll hide it after this chapter, but circular progress bars are common game elements, so I thought it was important to include an example of how to do them in this chapter.

Horizontal health bar

There are a few different ways that a horizontal health bar can be created, but the quickest and easiest way is to scale a single axis based on percentage. When setting up a horizontal health bar in this way, it is important to ensure that the anchor is set at a position that represents a completely depleted bar.

Remember that back in *Chapter 6*, we set the anchor of the health bar to the left, so we have already set the anchor correctly. We also scaled the health bar in the *x* direction to show what the bar would look like as it depleted.

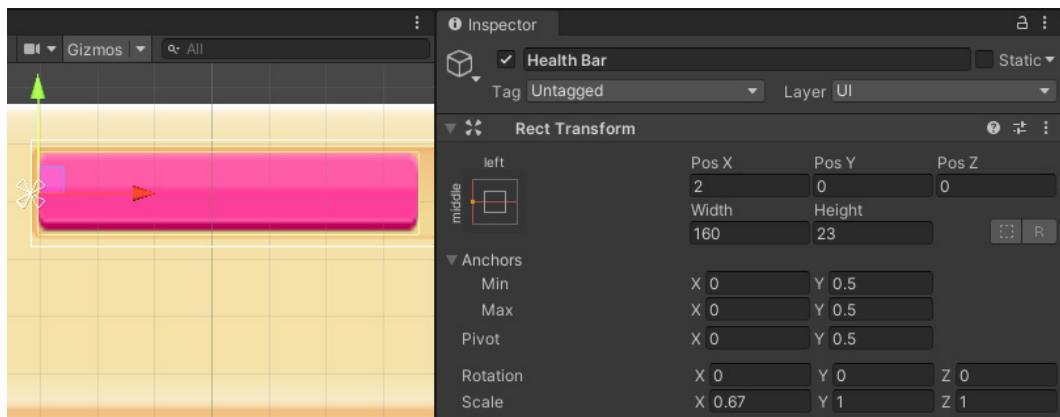


Figure 11.14: The Health Bar's Rect Transform component

Now, all we need to do is tie the percentage to the **X Scale** value of the health bar.

To tie the fill of the health bar to an actual value, complete the following steps:

1. Create a new C# script in your *Scripts* folder and name it *ProgressMeters.cs*.
2. In the *ProgressMeters* script, initialize the following four variables:

```
public uint health;
[SerializeField] uint totalHealth;
[SerializeField] float percentHealth;
[SerializeField] RectTransform healthBar;
```

The `health` variable represents the current health of the player, and the `totalHealth` variable represents the total health the player can obtain. As it doesn't make sense for these values to be negative, they have been initialized at the `uint` type or a positive integer. I have made the `health` variable `public` so that it can be accessed via other scripts and seen within the Inspector. I made `totalHealth` a private `SerializeField` so that it cannot be accessed via other scripts but still be seen and assigned via the Inspector.

The `percentHealth` variable will be calculated based on the quotient of the `health` and `totalHealth` variables. I made this value private and serialized, not so that we can edit it in the Inspector but so that we can easily see its value change in the Inspector.

The `healthBar` variable stores the `RectTransform` component of the `Health Bar` UI Image within our scene.

Note

Since `RectTransform` inherits from `Transform`, we could have declared `healthBar` as a `Transform` and the following code would still work.

3. Return to the Unity Editor and drag the *ProgressMeters* script onto `HUD Canvas > Top Left Panel`. Assign the value 500 to both the **Health** and **Total Health** slots. Drag the `Health Bar` UI Image into the **Health Bar** slot. Any value you try to type into the **Percent Health** slot will be overridden by the code we write in the next step. Your component should look as follows:

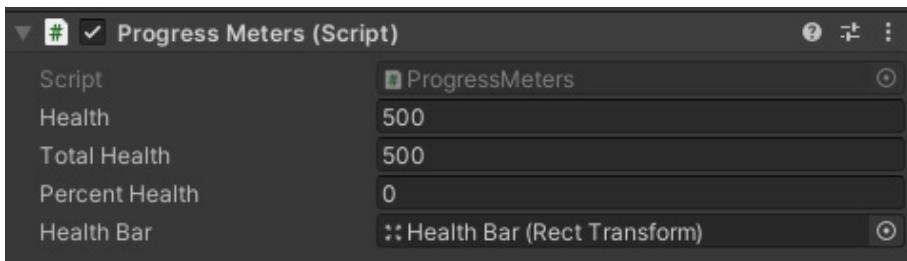


Figure 11.15: The Progress Meters component

4. We want any changes made to our `health` value to automatically update the `percentHealth` value and the scale of our `healthBar`. To do that, we can put the following code in the `Update()` function:

```
void Update()
{
    // Cap health
    if (health > totalHealth)
    {
        health = totalHealth;
    }

    // Calculate health percentage
    percentHealth = (float)health / totalHealth;

    // Update horizontal health bar
    healthBar.localScale = new Vector2(percentHealth, 1f);
}
```

Declaring our `health` and `totalHealth` variables with the `uint` type stopped them from becoming negative, but we still need to put an upper cap on our `health` variable. It doesn't make sense for it to exceed the `totalHealth` variable.

While `percentHealth` is a `float` variable, performing a division between two `uint` variables will result in a `uint` type, so adding `(float)` at the beginning of the integer division provides a `float` result from the division.

The last part of the code sets the `localScale` value of the `healthBar`. When you scale a UI object, you have to use `localScale`. This scales the object locally, meaning relative to its parent object.

5. Now, we can test the code easily in the Editor. Play the game and hover your mouse over the word **Health** in the **Progress Meters** component until the mouse displays two arrows around it, as shown in the following figure:

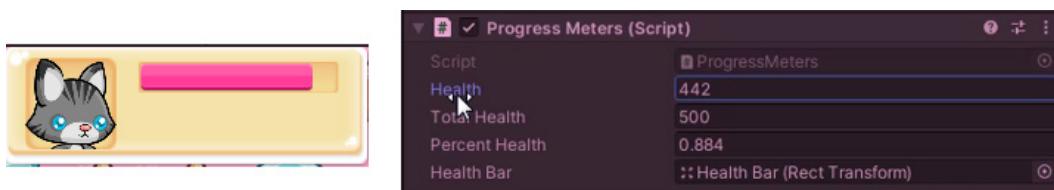


Figure 11.16: The Progress Meters component's effect on the meter

When these arrows appear, clicking and dragging will manipulate the values of the variable based on your mouse position. You'll see, as you do this, that as the **Health** value decreases, the **Percent Health** value decreases, and the **Health Bar** in the scene changes size. You'll note that you cannot set the value of **Health** below 0 or above 500.

As you can see, setting up a horizontal health bar isn't terribly difficult. Duplicating this process in a game where the health reduces by Events won't require a lot of steps to achieve. Just ensure that you set the anchor of the health bar correctly. This process will work similarly for a vertical health bar.

Circular progress meter

Horizontal health bars didn't take a lot of work to set up. The work to make a circular progress meter is just about as easy and can be completed with only two more lines of code. Since we don't already have a circular progress bar in our scene, we will have to start with a bit of setup first.

To create a circular progress bar, complete the following steps:

1. From the code bundle, drag the `circularMeter.png` sprite into the **Sprites** folder of your project.
2. Set the **Sprite Mode** of the `circularMeter.png` sprite to **Multiple** and automatically slice it in the **Sprite Editor**.
3. Select the **Top Left Panel** within the **Hierarchy** and give it a new **UI Image** child. Name the **Image Progress Holder**.
4. Similar to how we set up the health bar, there will be a holder and a fill. Drag the `circularMeter_0` sub-sprite into the **Source Image** slot of the **Image** component of **Progress Holder**.
5. It's important that we get the right proportions for our holder and fill Images. So, to ensure that the **Image** is correctly proportioned, hit the **Set Native Size** button on the **Image** component.
6. Now, add a child **UI Image** to **Progress Holder** called **Progress Meter**.
7. Set the anchor preset of **Progress Meter** to middle center. Do not stretch it.
8. Add `circularMeter_1` to the **Source Image** slot of the **Image** component on the **Progress Meter**.
9. Hit the **Set Native Size** button for the **Progress Meter** **Image** component as well. After completing this step, you should see the following:

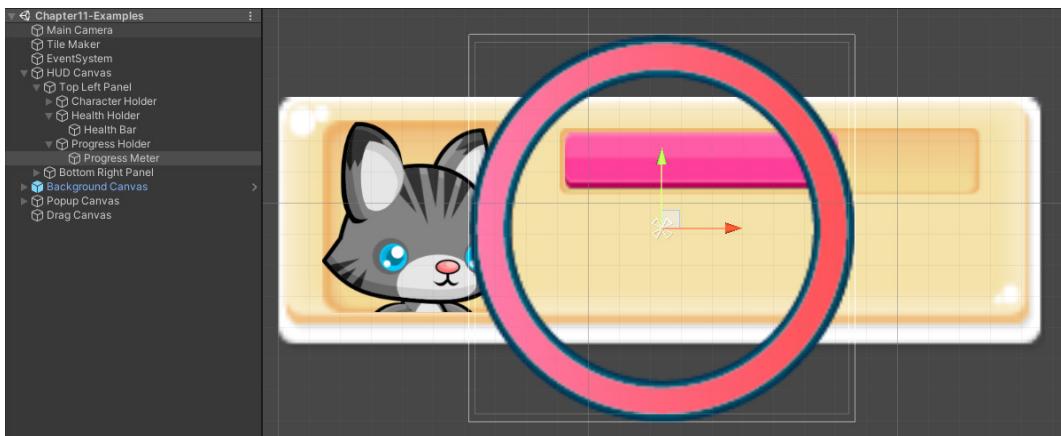


Figure 11.17: The progress on the Progress Meter

If the pink fill is not perfectly nested inside the blue holder, you may have forgotten to hit the **Set Native Size** button on one of the Images or the Progress Meter does not have its anchor preset set to middle center.

10. Let's move this meter and scale it a bit. Select **Progress Holder** and move it so that it is positioned in the scene below **Character Holder**. Set the **Scale X** and **Scale Y** values of **Progress Holder** to **0.8** to make it a little smaller.



Figure 11.18: Repositioning the circular progress meter

11. The last thing to do to the code is to change the **Image Type** of the **Progress Meter**. Change the **Image Type** to **Filled** with a **Radial 360 Fill Method**. Change the **Fill Origin** to **Top**. Adjust the scroll bar on the **Fill Amount** to preview the meter filling:

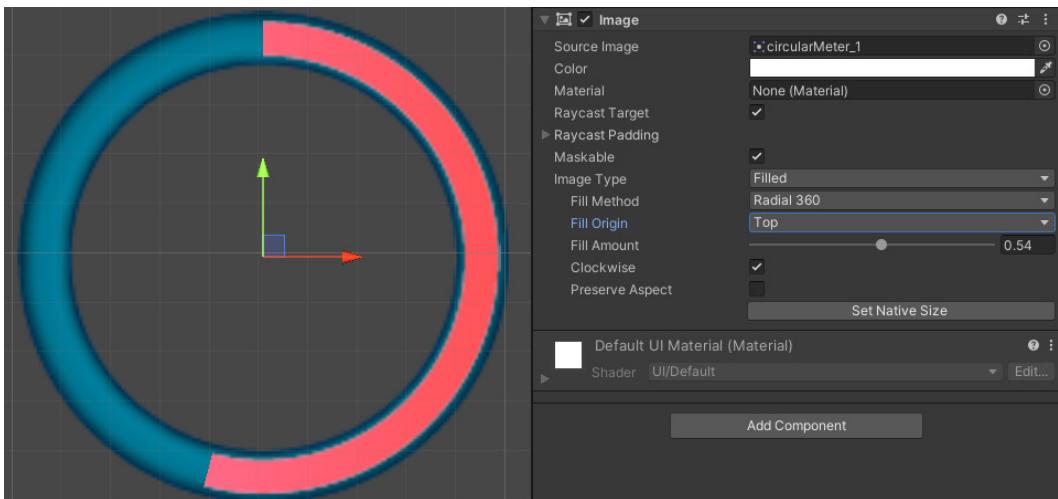


Figure 11.19: Adjusting the fill amount on the circular progress meter

- Now, we're ready to write some code. As you have probably guessed from adjusting the **Fill Amount** value in the Inspector, we'll want to tie the `fillAmount` value to the `percentHealth` variable in our code. First, we need to create a variable with which we can access the `Image` component of our `Progress Meter`. Declare the following variable at the top of your code:

```
[SerializeField] Image progressMeter;
```

- Now, add the following at the end of the `Update()` function:

```
// Circular progress meter
progressMeter.fillAmount = percentHealth;
```

- The last thing we need to do is hook the `Progress Meter` UI `Image` to the `progressMeter` variable. Drag the `Progress Meter` into the **Progress Meter** slot.

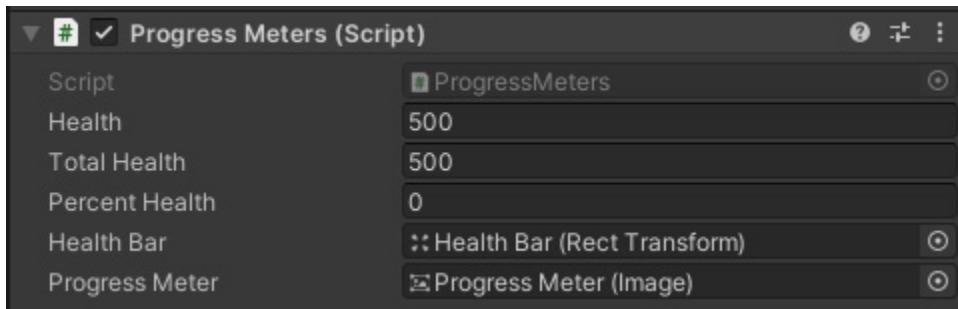


Figure 11.20: The `Progress Meter` component's updates

15. Play the game and adjust the **Health** value in the Inspector as you did earlier, and watch the two meters move in unison.

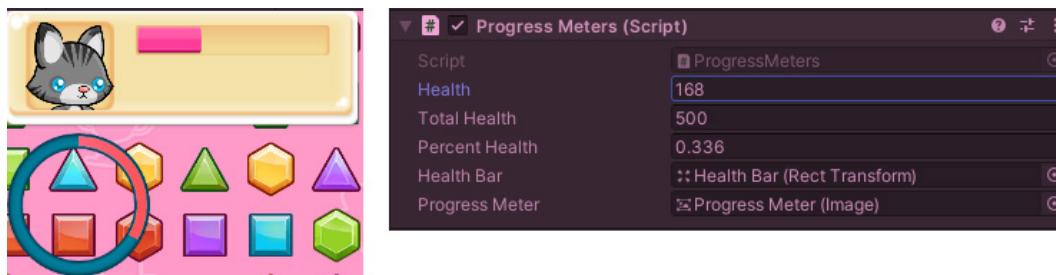


Figure 11.21: Result of the Progress Meter

As you can see, making a circular progress meter is really not more difficult than making a horizontal one!

Note

In the same way we used the fill amount for the circular progress bar, we could have used the fill amount for the horizontal health bar. Setting the **Image Type** property to **Filled** and **Horizontal** and then affecting the **Fill Amount** value rather than the scale would have had a similar effect.

Because this circular progress meter wasn't part of the original UI plan and was only placed in the scene for demonstration purposes, I am going to disable it in all future figures and screenshots.

Mute Buttons with sprite swap

Now, let's look at an example where we swap the sprite of a Button based on a pre-defined state. This is different than a sprite swap transition, which we discussed in *Chapter 9*, because it won't use the states of highlighted, pressed, selected, or disabled. It's included in this chapter rather than the Buttons chapter since it involves affecting the Image component, not the Button component.

In the scene, we have a **Pause Panel** that pops up when the **P** key is hit on the keyboard. On this Panel, we will place two mute Buttons, one for music and one for sound, which will toggle between muted and unmuted states. The Panel will appear as shown in the following screenshot:



Figure 11.22: The Pause Panel with new mute Buttons

To add the music and sound Buttons shown in the preceding screenshot, complete the following steps:

1. First, we need to bring in a new art asset. The Button sprites on the sprite sheet we imported previously are a bit too small and don't contain a muted version. So, I edited them a bit and provided a new sprite for you. In the book's source files, you should find an .png file named `muteUnmute.png`:



Figure 11.23: The `muteUnmute.png` sprite

Import this .png file into your project's `Assets/Sprites` folder.

2. Slice the sprite into multiple sub-sprites by changing its **Sprite Mode** to **Multiple**, opening its **Sprite Editor**, and applying the automatic slice type. Multiple sprites should appear as follows:

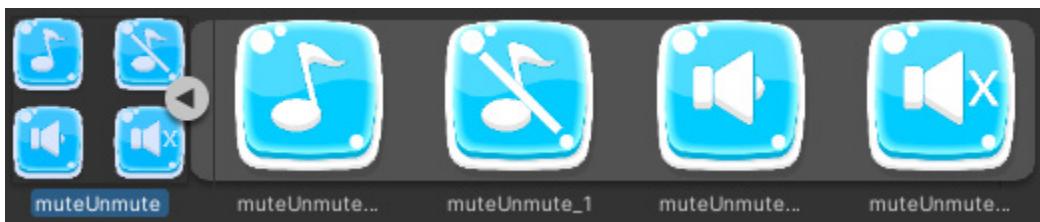


Figure 11.24: The `muteUnmute.png` sprite sliced

3. Create two new Buttons as children of the Pause Panel and name them Music Button and Sound Button. Delete their text children because we do not need them.

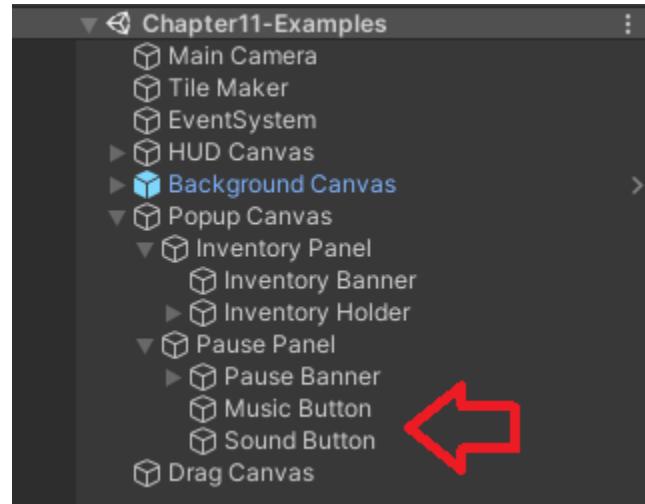


Figure 11.25: The current view of the Hierarchy

4. Give the two new Buttons the following **Rect Transform** and **Image** properties:

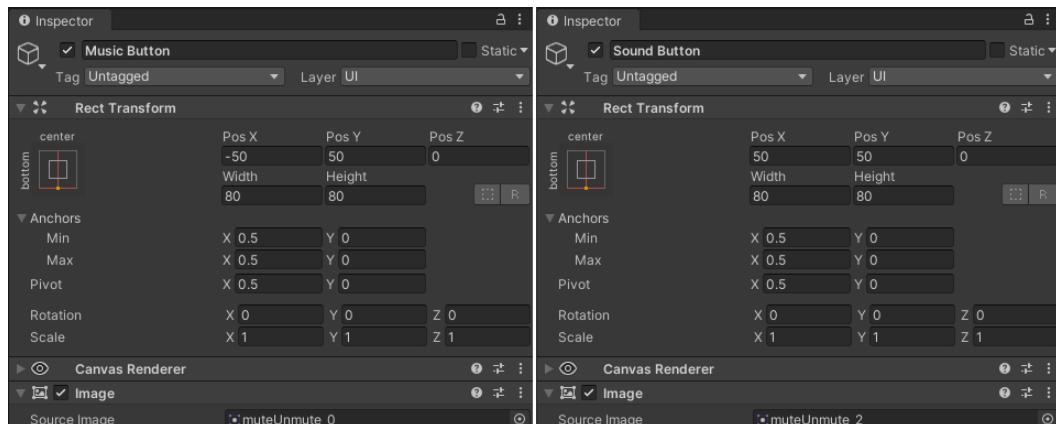


Figure 11.26: The Rect Transform of the two Buttons

Your Panel should now look just like the one at the beginning of this example:



Figure 11.27: The Pause Panel

5. Now, let's write some code to make these Buttons swap sprites that will represent the sound and music toggling on and off. Create a new script called `MuteUnmute.cs` in your `Assets/Scripts` folder.

Replace the code of `MuteUnmute` with the following:

```
public class MuteUnmute : MonoBehaviour
{
    [SerializeField] Button musicButton;
    private Image musicImage;
    [SerializeField] private Sprite[] musicSprites = new
    Sprite[2];
    private bool musicOn = true;

    [SerializeField] Button soundButton;
    private Image soundImage;
    [SerializeField] private Sprite[] soundSprites = new
    Sprite[2];
    private bool soundOn = true;

    void Awake()
    {
        musicImage = musicButton.GetComponent<Image>();
        soundImage = soundButton.GetComponent<Image>();
    }
}
```

```
public void ToggleMusic()
{
    musicOn = !musicOn;
    musicImage.sprite = musicSprites[Convert.
ToInt32(musicOn)];
}

public void ToggleSound()
{
    soundOn = !soundOn;
    soundImage.sprite = soundSprites[Convert.
ToInt32(soundOn)];
}
```

As you can see, this code contains two main functions: `ToggleMusic()` and `ToggleSound()`. These two function identically by simply swapping the sprite on the specified Button based on the `musicOn` and `soundOn` Boolean values.

To swap the sprite, the script first finds the Image component on the two Buttons specified as `musicButton` and `soundButton`, within the `Awake()` function. These Buttons will be assigned in the Inspector. It then swaps the sprite of the Image component to the correct sprite from an array of sprites. The sprites for the mute and unmute states will be assigned in the Inspector in a future step.

Note

Sadly, this book does not cover adding sound and music to a Unity project. The code provided here doesn't actually mute and unmute audio; it simply swaps sprites. You will simply need to include two audio sources: one for playing music and one for playing sounds that have the **Music** and **Sound** tags, respectively.

1. Go back to the Unity Editor and attach the `MuteUnmute.cs` script to the `Pause Panel` by dragging it into its **Inspector**:

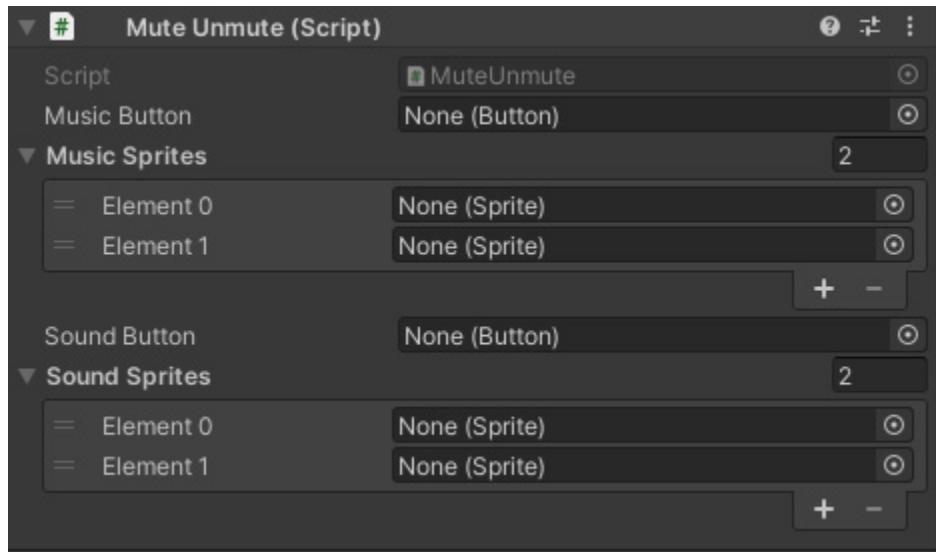


Figure 11.28: The Mute Unmute component

- Now, we want to assign the appropriate Buttons and sprites to the slots. Drag the Music Button and Sound Button into their designated slots from the Hierarchy. Drag and drop the audio Button sprites from the project view to their appropriate slots, making sure to put the muted sprite in the 0 element of the array.



Figure 11.29: The updated Mute Unmute component

3. Now, we just need to hook up the Buttons to call the appropriate functions. Select the Music Button. Select the + sign at the bottom of the `OnClick()` Event list of the **Button** component to add a new Event. The script we want to access, `MuteUnmute.cs`, is on the Pause Panel, so drag the Pause Panel into the object slot. Now, from the function dropdown menu, select **MuteUnmute | ToggleMusic**.
4. Perform the same actions as you did in the previous step for the Sound Button, but this time select **MuteUnmute | ToggleSound** from the function dropdown list.

Now, play the game, press *P* to bring up the Pause Panel, and you will see the Buttons toggle back and forth between their two different sprites.

Now that we've looked at how to implement progress meters and sprite swap Buttons, let's look at how to implement a few different mobile-specific interactions.

Adding press-and-hold/long-press functionality

Press-and-hold is utilized frequently in mobile games. Many games that use right-click on a PC or the web use press-and-hold when they are converted to the mobile platform.

To demonstrate how to implement press-and-hold functionality, we will create a Button that has a growing ring that represents hold time. Once a specified amount of time has passed, a function will fire:



Figure 11.30: Press-and-hold Button example

When working on this example, it is important to remember that even though the code is referencing a pointer, this functionality does not work exclusively with a mouse. Placing a finger on a touchscreen functions in the same way as a pointer down, and picking up the finger works the same as a pointer up.

To create a Button with a growing ring that represents hold time, complete the following steps:

1. Create a new scene named `Chapter11-Examples-Buttons1` in the `Assets/Scenes` folder and open the new scene.
2. Select `+ | UI | Button` to create a new Button in the scene.
3. Set the Button's **Transition** type on the **Button** component to **None**.
4. Change the text on the Button to say `Press and Hold`.
5. Right-click the **Button** in the Hierarchy and select **UI | Image** to add an **Image** child to the **Button**.
6. Change the **Width** and **Height** on the **Image**'s **Rect Transform** component to 50.

7. Assign the `circularMeter_1` sprite to the **Source Image** property of the **Image** component.
8. Change the **Image Type** to **Filled** and change the **Fill Amount** to 0.
9. To create press-and-hold functionality on the **Button**, we will utilize the **Pointer Down** and **Pointer Up** Events. Add the **Event Trigger** component to the **Button** object.
10. Select **Add New Event Type** and select **PointerDown**.
11. Select **Add New Event Type** and select **PointerUp**.
12. Now, we need to actually write the functions that will be called by the Event Triggers we set up in the previous steps. Create a new script in the `Assets/Scripts` folder called `LongPressButton`.
13. Before opening the script, go ahead and attach it as a component to **Button**.
14. Add the `UnityEngine.UI` namespace to the top of the script with the following:

```
using UnityEngine.UI;
```

15. To check how long the **Button** is being pressed, we will use a Boolean variable that checks to see if the **Button** is being held and a few different variables related to time. Add the following variable declaration to your script:

```
private bool buttonPressed = false;  
private float startTime = 0f;  
private float holdTime = 0f;  
[SerializeField] private float longHoldTime = 1f;
```

The `buttonPressed` variable will be set to `true` with the **Pointer Down** Event and `false` with the **Pointer Up** Event. The `startTime` variable will be set to the current time when the **Pointer Down** Event is triggered. The `holdTime` variable will determine how much time has passed since `startTime`. The `longHoldTime` variable is the amount of time the **Button** must be held down before the long press is complete. It is serialized so that it can be easily customized.

16. The last variable we need will represent the radial filling **Image**. Add the following variable declaration to your code:

```
[SerializeField] private Image radialFillImage;
```

17. Now, we need to write a function that will be called by both the **Pointer Down** and **Pointer Up** Events:

```
public void PressAndRelease(bool pressStatus)  
{  
    buttonPressed = pressStatus;  
  
    if (!buttonPressed)  
    {
```

```

        holdTime = 0;
        radialFillImage.fillAmount = 0;
    }
    else
    {
        startTime = Time.time;
    }
}

```

This function accepts a Boolean variable from the Event Trigger. It then sets the value of `buttonPressed` to the passed value.

When the Button is released, a value of `false` will be passed to the function. If the value passed is `false`, the amount of time that has passed, `holdTime`, is reset to 0, and the `radialFillImage` Image is reset to have a `fillAmount` value of 0.

When the Button is pressed, the `startTime` value will be set to the current time.

18. Create a function that will be called once the full amount of time needed for the long press, specified by `longHoldTime`, has completed:

```

public void LongPressCompleted()
{
    radialFillImage.fillAmount = 0;
    Debug.Log("Do something after long press");
}

```

This function doesn't really do anything but reset the filling Image and print out a `Debug.Log`. However, you can later reuse this code and replace the `Debug.Log` line with more interesting and meaningful actions.

19. The `Update()` function can be used to make the timer count upward. Adjust the `Update()` function as follows:

```

void Update()
{
    if (buttonPressed)
    {
        holdTime = Time.time - startTime;

        if (holdTime >= longHoldTime)
        {
            buttonPressed = false;
            LongPressCompleted();
        }
    }
}

```

```
        radialFillImage.fillAmount = holdTime /  
longHoldTime;  
    }  
}  
}
```

This code makes the value of `holdTime` tick upward if the `buttonPressed` value is set to `true`. Remember—`buttonPressed` will be set to `true` with a **Pointer Down** Event and `false` with a **Pointer Up** Event. So, it will only be true if the player has pressed the Button and not yet released it.

Once the `holdTime` value reaches the value specified by `longHoldTime`, the timer will stop ticking up, because `buttonPressed` will be reset to `false`. Additionally, the `LongPressCompleted()` function is called. If `longHoldTime` has not yet been reached, the Image's radial fill will update to represent the percentage of total required time that has transpired.

20. Now that the script is completed, we can hook up the `PressAndRelease()` function with the Event Triggers on the Button. Add the `PressAndRelease` function from the static list to both the **Pointer Down** and **Pointer Up** Event Triggers. Since the `PressAndRelease()` function accepts a Boolean variable, there is a checkbox representing the Boolean value that should be passed. Select the checkbox for the **Pointer Down** Event (sending `true`) but not for the **Pointer Up** Event (sending `false`).

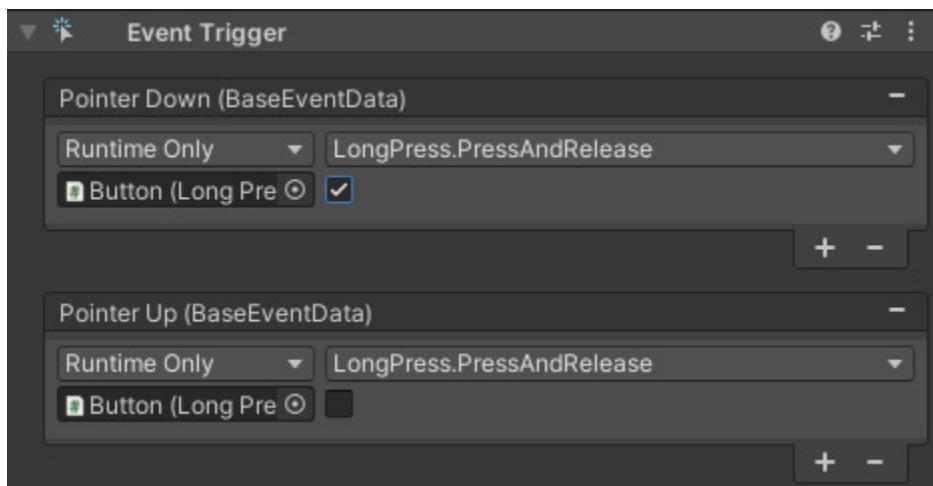


Figure 11.31: The Event Trigger component

21. Now, we need to assign the Image to the **Radial Fill Image** slot on the **Long Press** component.

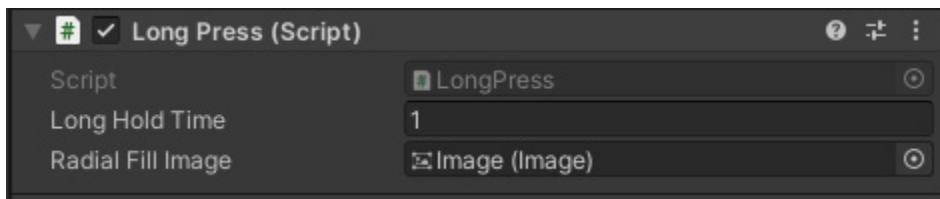


Figure 11.32: The Long Press component

Playing the game now will demonstrate the Image radially filling when you hold the Button and printing `Do something after long press` in the console. If you release the Button before the fill has completed, it will go away and reset for when you start clicking again.

Press-and-hold is a pretty common functionality, and while it isn't a pre-installed Event in the Unity Event library, luckily it isn't too difficult to hook up. I recommend holding on to that script so that you can reuse it in the future.

Creating a static four-directional virtual D-Pad

A D-Pad is simply four Buttons on a directional pad. To create a D-Pad for a mobile game, you just need to create a graphic that contains four Buttons on the directions.

The art used in this example was obtained from <https://opengameart.org/content/onscreen-controls-8-styles>.

To create a virtual D-Pad, complete the following steps:

1. Create a new scene named `Chapter11-Examples-Buttons2` in the `Assets/Scenes` folder and open the new scene.
2. Import the `dPadButtons.png` sprite sheet into the `Assets/Sprites` folder.
3. Change the newly imported sprite's **Sprite Mode** to **Multiple** and automatically slice it.
4. Create a new Canvas with `+ | UI | Canvas`. Name the new Canvas `D-Pad Canvas`.
5. The size of a D-Pad is incredibly important on mobile devices. Even if the screen gets smaller, you'll probably want the D-Pad to be about the same size. If it gets too small, the game can be unplayable or uncomfortable. Therefore, set the **Canvas Scaler** component's **UI Scale Mode** value to **Constant Physical Size**.
6. Add a new Image as a child of `D-Pad Canvas` with `+ | UI | Image` and rename it `D-Pad Background`.
7. Set the **Source Image** to `dPadButtons_4`.
8. Set its anchor and pivot to the lower-left corner of the screen, and set its **Pos X** and **Pos Y** values to `30`.

9. Set its **Width** and **Height** to 200:

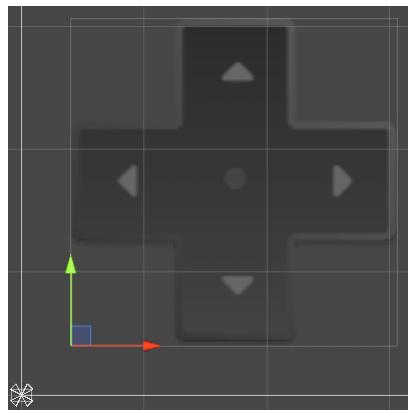


Figure 11.33: The D-Pad positioned correctly

10. Right-click on D-Pad Background and add a new Button as a child with **UI | Button**. Rename the new Button Up.
11. Remove its child Text object.
12. Set the **Pos X**, **Pos Y**, **Width**, and **Height** values of the Up Button to 0, 65, 60, and 60, respectively. This will create a square over just the up position of the D-Pad Image.

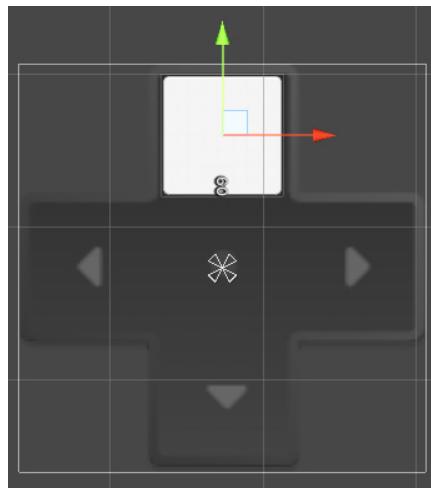


Figure 11.34: The D-Pad with an Up Button

13. Duplicate the Up Button three times and rename the duplicates Right, Left, and Down.
14. Set the Right Button's **Pos X** and **Pos Y** values to 65 and 0, respectively.

15. Set the **Left** Button's **Pos X** and **Pos Y** values to **-65** and **0**, respectively.
16. Set the **Down** Button's **Pos X** and **Pos Y** values to **0** and **-65**, respectively. You should now have four Buttons positioned as follows:

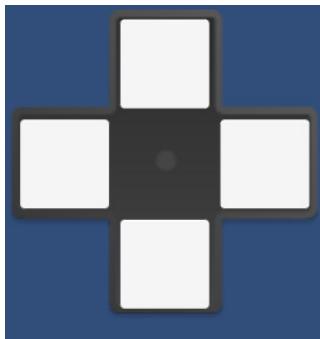


Figure 11.35: The D-Pad with all of its Buttons

These four Buttons cover the entire area of the arms of the directional pad. They will act as the hit area for the directions.

17. We only really want these four Buttons for their hit area and don't want to actually have them visible in the UI. Select all four of the Buttons in the Hierarchy and set the alpha value on the **Color** property of their **Image** component to **0**.
18. Since the directional pad Image is static and not split into four separate Buttons, any transitions applied to it would cover the whole Image. However, we can make the individual directions look as if they are being pressed and have some sort of color transition by adding sub-Images for the arrows on the directions. Right-click the **Up** Button and add an Image as a child with **UI | Image**. Rename the new Image **Arrow**.
19. Assign the **dPadButtons_5** sprite to the **Source Image** on its **Image** component and select **Preserve Aspect**.
20. Scale and move the Image so that it is appropriately lined up with the arrow displayed on the D-Pad background Image:

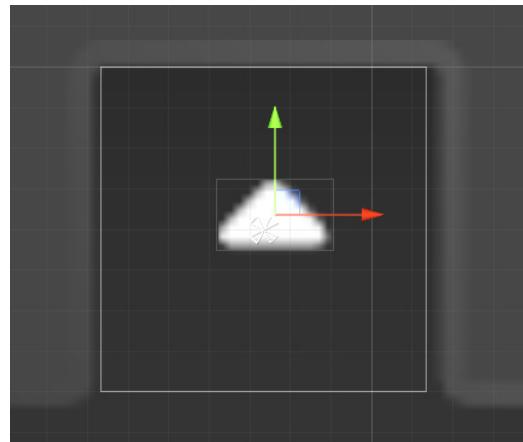


Figure 11.36: The Up Arrow of the D-Pad

21. Select the **Color** slot on the **Arrow Image** component and use the eye dropper tool to grab the color of the arrows from the **D-Pad Background Image**. This will make it a light gray instead of white.
22. Create **Arrow** children for each of the other three Buttons, and size, position, and color them appropriately. Make sure to also use the correct sub-Image of the **dPadButtons** sprite sheet. Once completed, your D-Pad and Hierarchy should appear as follows:

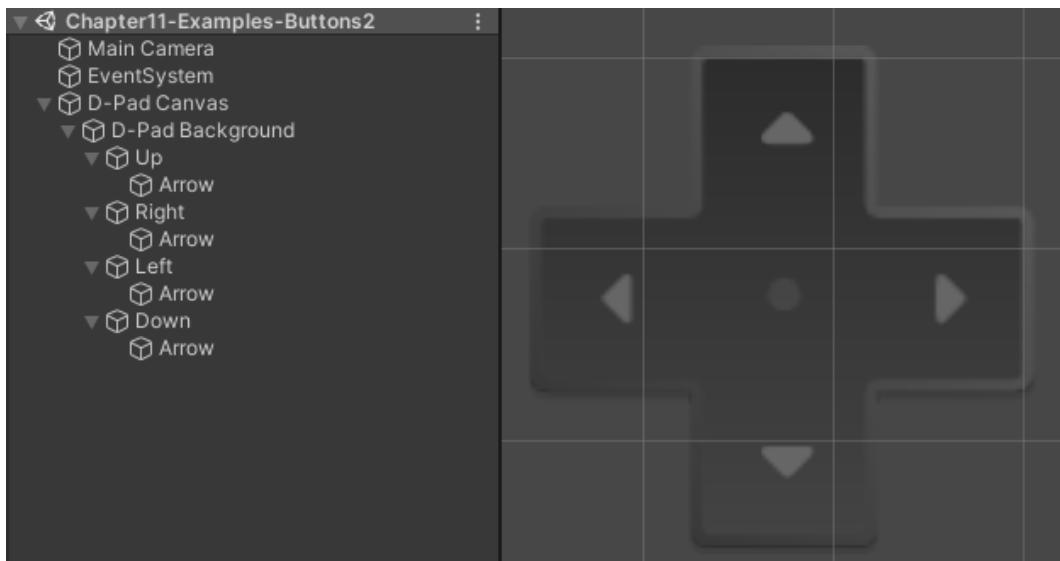


Figure 11.37: All Arrows on the D-Pad

23. Now, so that the D-Pad will react visually when the four directions are pressed, we will set the four Arrow children to have color tint Button **Transitions** when the four Buttons are pressed. For each Button, drag its Arrow child into the **Target Graphic** slot on its **Button** component. Now, when you press the individual Buttons, you will see a slight change in the color of the arrows, indicating which direction is pressed. You may wish to change the pressed color to something a bit more drastic than the default gray if you are having difficulty telling that a change is occurring.
24. Add the script named DPad.cs from the book's code bundle to the Assets/Scripts folder. This is an incredibly simple script that contains four functions that only write to the console. Hooking up these four functions to the individual directional Buttons won't do anything fun, but it will allow us to see logs in the console that let us know the Buttons are performing as they should.
25. Attach the script to the D-Pad Background object.
26. Select each of the four directional Buttons and, with all of them selected, add an **On Click ()** Event to the **Button** component.
27. Now, drag the D-Pad Background object into the object slot of the **On Click ()** Event.
28. Select each Button individually and assign the appropriate functions, PressUp, PressDown, PressLeft, and PressRight, to their **On Click ()** Events.

Playing the game and selecting the four directional Buttons should result in the appropriate message being displayed on the console.

Many D-Pads actually accept nine inputs: the four directs, the four diagonals (corners), and the center. If you want to accept diagonal inputs as well as a center-click for your D-Pad, I'd suggest using a grid layout group to evenly space your nine Buttons.

Since D-Pads tend to allow press-and-hold, you may want to combine the process used in this example with actions similar to those described in the previous example. Instead of using the **On Click ()** Event, you could set up an Event Trigger for using the OnPointerDown and OnPointerUp Events. These Events could then set a Boolean variable to true and false. For example, on the Right Button, you could have the OnPointerDown Event set a variable called moveRight to true and the OnPointerUp Event set moveRight to false.

Creating a floating eight-directional virtual analog stick

In this example, we will create a floating eight-directional virtual analog stick. First, we will create an eight-directional D-Pad that simulates a control stick that moves in the direction the player drags:

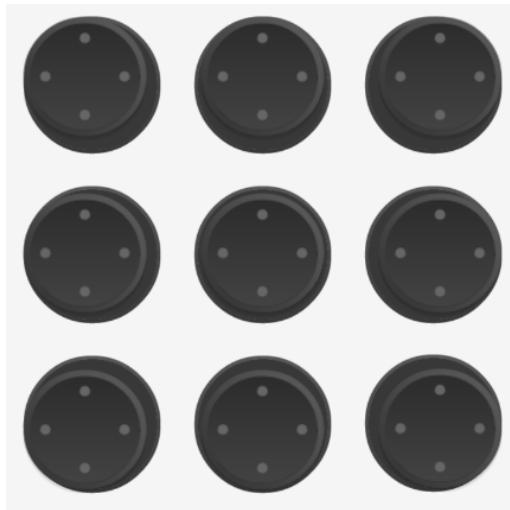


Figure 11.38: The positions of the floating analog stick

Then, we will expand the eight-directional D-Pad so that it is floating, which means it will not be visible in the scene until the player presses somewhere in the screen. Then, it will appear where the player's thumb is located and perform the eight-direction movement based on the player's thumb dragging.

Setting up the eight-directional virtual analog stick

To create an analog stick that moves in eight directions, as shown in the previous figure, complete the following steps:

1. Create a new scene named Chapter11-Examples-Buttons3 in the Assets/Scenes folder and open the new scene.
2. Create a new Image with + | UI | Image. Name the new Image Stick_Base.
3. Add the dPadButtons_15 sprite to the **Source Image** slot of its **Image** component.
4. Resize it so that it has a **Width** and **Height** of 200 and give it a **Pos X** and **Pos Y** of 0.
5. Right-click Stick_Base in the Hierarchy and select **UI | Image** to add an **Image** child to the Stick_Base. Name the child **Stick**.
6. Resize the **Stick** Image to match the **Stick_Base** by setting its **Rect Transform** stretch and anchor to stretch fully across both directions.
7. Add the dPadButtons_0 sprite to the **Source Image** slot of the **Stick** Image component.
8. Set the **Left**, **Top**, **Right**, and **Bottom** properties of the **Rect Transform** component all to 10 to give some padding around the edges of the **Stick**.
9. Now, set the pivot and position to **middle center**. This is an important step! Without doing this, the **Stick** will not move around appropriately on the **Stick_Base**.

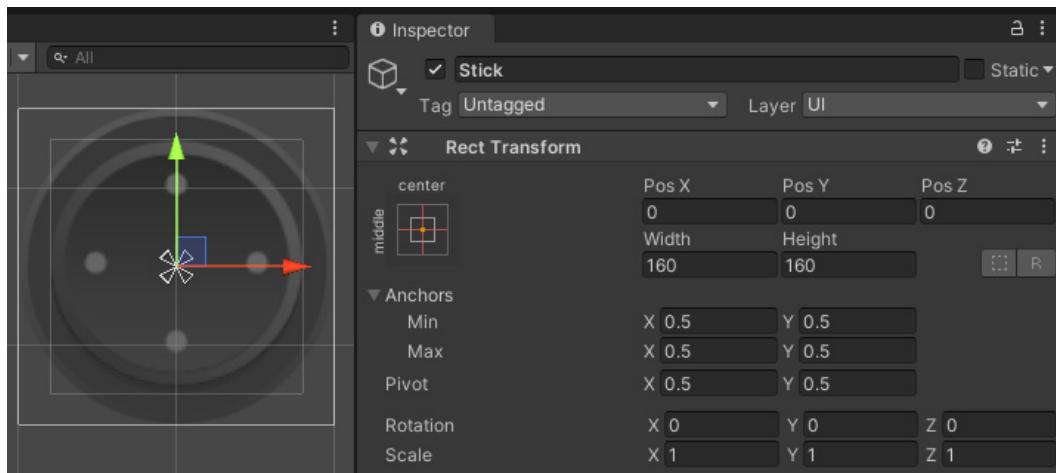


Figure 11.39: The Rect Transform component of the Stick

10. That's all there is for the setup to get our virtual analog stick working. We'll just leave it in the center of the screen for now. Now, we need to write some code. Create a new script in the `Assets/Script` folder and name it `FloatingAnalogStick`.
11. Add the `UnityEngine.UI` namespace to the top of the script with the following:

```
using UnityEngine.UI;
```

12. To make the stick wiggle around on top of the base, we need the following variables:

```
[SerializeField] private RectTransform theStick;
private Vector2 mouseStartPosition;
private Vector2 mouseCurrentPosition;
[SerializeField] private int dragPadding = 30;
```

The first three variables should be pretty self-explanatory. The `dragPadding` variable will be used to determine how far the player has to drag the stick before it actually registers as being moved.

13. Before we write the code that checks how far the player has dragged their finger, let's add a few dummy functions that would allow this analog stick to actually control something in the future. Add the following functions to your script:

```
public void MovingLeft()
{
    Debug.Log("move left");
}

public void MovingRight()
```

```
{  
    Debug.Log("move right");  
}  
  
public void MovingUp()  
{  
    Debug.Log("move up");  
}  
  
public void MovingDown()  
{  
    Debug.Log("move down");  
}
```

14. The Stick will move outward when the player moves their finger from their starting finger-down position. So, let's create a function that will find the starting position when the player begins dragging their finger. Add the following function to your script:

```
public void StartDrag()  
{  
    mouseStartPosition = Input.mousePosition;  
}
```

Remember—when working with a touchscreen, `Input.mousePosition` will give the value of the touch position.

15. Now, let's create a function that checks how far the player has dragged their finger and moves the Stick based on that information. Add the following function to your script:

```
public void Dragging()  
{  
    float xPos;  
    float yPos;  
    mouseCurrentPosition = Input.mousePosition;  
  
    if (mouseCurrentPosition.x < mouseStartPosition.x -  
        dragPadding)  
    {  
        MovingLeft();  
        xPos = -10;  
    }  
    else if (mouseCurrentPosition.x > mouseStartPosition.x +  
        dragPadding)  
    {  
        MovingRight();  
        xPos = 10;  
    }  
}
```

```

        }

        else
        {
            xPos = 0;
        }

        if (mouseCurrentPosition.y > mouseStartPosition.y +
dragPadding)
{
    MovingUp();
    yPos = 10;
}
else if (mouseCurrentPosition.y < mouseStartPosition.y -
dragPadding)
{
    MovingDown();
    yPos = -10;
}
else
{
    yPos = 0;
}

theStick.anchoredPosition = new Vector2(xPos, yPos);
}

```

16. The last piece we need to add is something that will reset the stick to its original position once the player stops dragging or lifts up their finger. Add the following function to your script to do so:

```

public void StoppedDrag()
{
    theStick.anchoredPosition = Vector2.zero;
}

```

17. Now, we need to hook this script and these functions to the items within the scene. Add the **FloatingAnalogStick** script to the **Stick Base** Image.
18. Add the **Stick** Image to the **Stick** property in the **Floating Analog Stick** component.
19. Add an **Event Trigger** component to the **Stick Base** object with **Add Component | Events | Event Trigger**. This will allow the user to use Event Types other than **On Click()**.
20. Add the **Begin Drag**, **Drag**, and **End Drag** Event Types with the **Add New Event Type** button.
21. Add the appropriate functions on the **FloatingAnalogStick** script attached to the **Stick Base** to the Events:

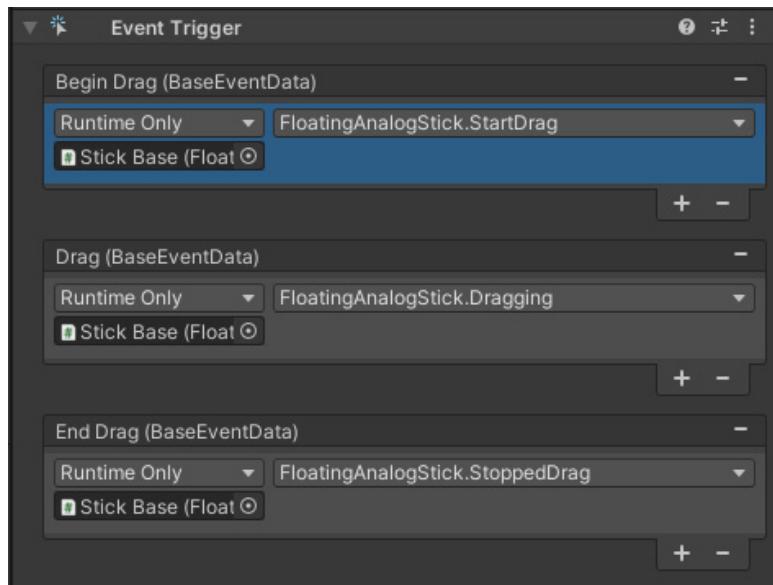


Figure 11.40: The Event Trigger component

If you play the game now, you should see the eight-directional analog stick responding appropriately. Clicking on it and dragging in any direction will cause the stick to move in the direction of the drag.

Making the eight-directional virtual analog stick float

If all you want is an eight-directional analog stick, you're good to go! But if you want the analog stick to float—appear where the players press on the screen and disappear when they lift their finger—you have to do a little bit more work.

To make the analog stick appear where the player clicks, complete the following steps:

1. First, we need to create an area where the player will click to bring up the analog stick. Right-click **Canvas** in the Hierarchy and select **UI | Button** to add a **Button** child to the **Canvas**. Rename the **Button** child **Click Area**.
2. Remove the **Text** child object from the **Click Area**.
3. Stretch the **Click Area** to fill the whole **Canvas**.
4. Add some padding to the sides of **Click Area** by changing the **Left**, **Top**, **Right**, and **Bottom** properties on the **Rect Transform** component to 50. I have added this padding so that the player cannot click on the very edge of the screen and have the analog stick appear mostly off-screen.
5. In the Hierarchy, move **Click Area** so that it is above **Stick Base**. Now, **Click Area** will render behind the analog stick:

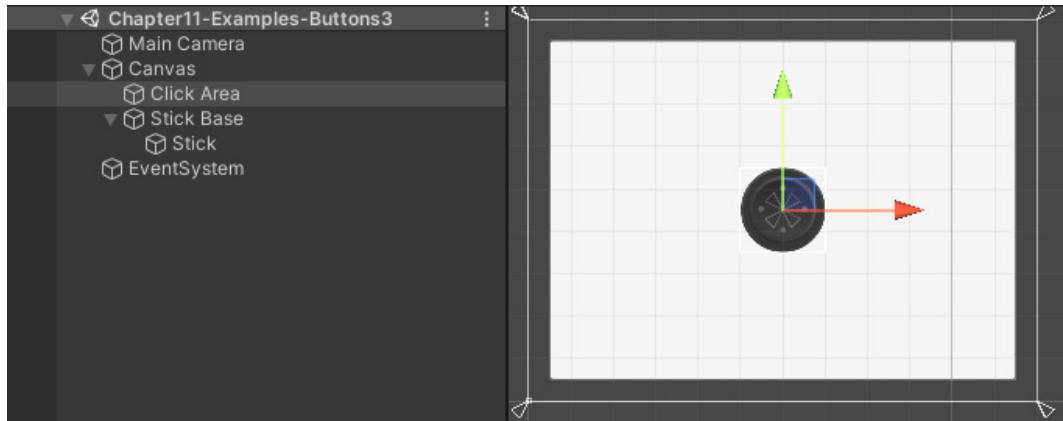


Figure 11.41: The Hierarchy showing Click Area and Stick Base

6. Open up your `FloatingAnalogStick` code so that we can add some functionality to it.
7. To make the position of our analog stick more easily hook to the position of the mouse on the screen, we should move our stick base so that it is centered at the lower-left corner of the `Canvas`s. Set the anchor and position to the **bottom left** anchor preset.
8. Now, set the **X** and **Y Pivot** properties to `0.5`.
9. Set the **Pos X** and **Pos Y** properties to `0`. This should place the analog stick at the lower-left corner of the `Canvas` (or screen) with its pivot point set to its center:

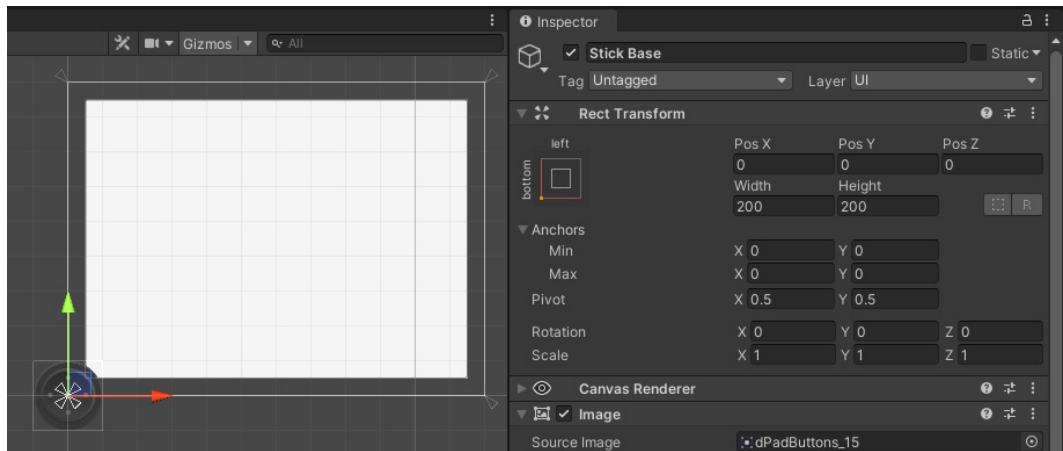


Figure 11.42: The Hierarchy showing Click Area and Stick Base

10. We need two more variables now to get the analog stick to appear where we want it to in the scene. Add the following variable to your script to assign the stick and track if it has been added:

```
[SerializeField] private RectTransform theBase;
[SerializeField] private bool stickAdded = false;
```

11. Now, create the following function to add the stick to the scene:

```
public void AddTheStick()
{
    theBase.anchoredPosition = Input.mousePosition;
    theStick.anchoredPosition = Vector2.zero;
    mouseStartPosition = Input.mousePosition;
    stickAdded = true;
}
```

12. Add the following `Update()` function to determine when the stick will appear:

```
void Update()
{
    if (stickAdded == true)
    {
        Dragging();

        if (Input.GetMouseButtonUp(0))
        {
            // ToggleBaseCanvasGroup(false); // This line is
            // commented out as ToggleBaseCanvasGroup is not defined in the
            // provided code
            stickAdded = false;
            StoppedDrag();
        }
    }
}
```

13. Add `Stick Base` to the `The Base` slot:

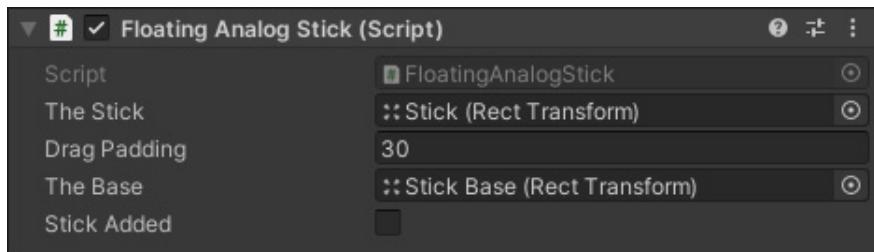


Figure 11.43: The Floating Analog Stick component

14. Add the **Event Trigger** component to the Click Area.
15. Add the following Pointer Down Event to the Click Area:

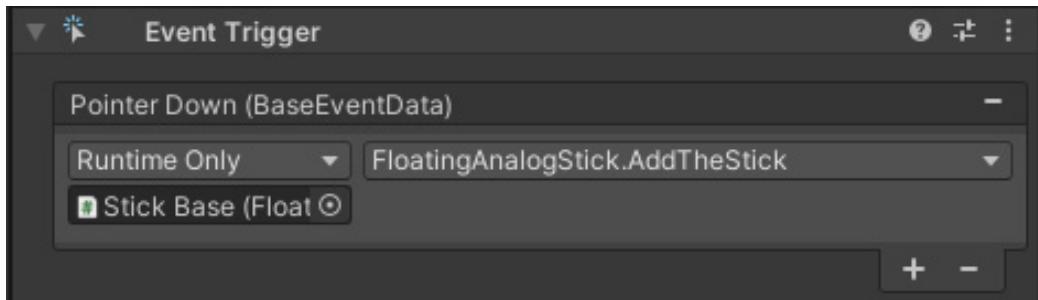


Figure 11.44: The Event Trigger component

Playing the game now will have the analog stick appear where you click and move around with your dragging.

16. Now, let's make it so that the analog stick is only visible when the player is touching the screen. Add a **Canvas Group** component to Stick Base.
17. Set **Alpha** to 0, **Interactable** to false, and **Blocks Raycast** to false.
18. Add the following variable to your FloatingAnalogStick script to keep track of the CanvasGroup:

```
private CanvasGroup theBaseVisibility;
```

19. Add the following Awake () function to initialize the theBaseVisibility variable:

```
void Awake()
{
    theBaseVisibility = theBase.GetComponent<CanvasGroup>();
}
```

20. Create a new function called ToggleBaseCanvasGroup () to toggle the CanvasGroup's properties on and off:

```
public void ToggleBaseCanvasGroup(bool visible)
{
    theBaseVisibility.alpha = Convert.ToInt32(visible);
    theBaseVisibility.interactable = visible;
    theBaseVisibility.blocksRaycasts = visible;
}
```

21. Add the following to the `AddTheStick()` function to turn on the `CanvasGroup`:

```
ToggleBaseCanvasGroup(true);
```

22. Add the following to the innermost `if` statement within the `Update()` function to turn off the `CanvasGroup`:

```
ToggleBaseCanvasGroup(false);
```

Now, the analog stick will appear when the player presses down, move in the direction of their finger, and disappear when the player lifts their finger.

Summary

UI Images are one of the core components of the Unity UI system and manipulating them is essential to creating visually interactive user interfaces. This chapter culminated all the skills we have learned in the preceding chapters by letting us create interesting interfaces that utilized Events, Buttons, and Images.

In the next chapter, we'll look at how to create masks and scroll views so that we can hold even more child objects within our Panel containers.

12

Using Masks, Scrollbars, and Scroll Views

We've learned how to make UI that has all of its components visible on the screen at once, but often you will have UI elements that are off-screen or off-menu and not visible until you navigate to them or reveal them.

In this chapter, we will discuss the following topics:

- How to use masks to hide portions of UI Images
- Using Scrollbars and accessing their properties via code
- Utilizing the UI Scroll View to create scrollable menus
- Creating a settings menu with a mask and scrolling text

Note

All the examples shown in this chapter can be found within the Unity project provided in the code bundle. They can be found within the scene labeled **Chapter12**.

Each example image has a caption stating the example number within the scene.

In the scene, each example is on its own Canvas, and some of the Canvases are deactivated. To view an example on a deactivated Canvas, simply select the checkbox next to the Canvas' name in the **Inspector**. Each Canvas is also given its own Event System. This will cause errors if you have more than one Canvas activated at a time.

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2012>

Using masks

Masks affect the visibility of objects within their shape. If an object is affected by a mask, any part of it outside the mask's restricted area will be invisible. The visible area of a mask can either be determined by an Image with the **Mask** component or a Rect Transform with the **Rect Mask 2D** component.

With UI, masking can be used for scrolling menus, so items that exist outside of the menu's area will not be visible. It is also used to *cut out* images. For example, the following image shows a cat's image being cut out by a circular mask:



Figure 12.1: Circular Mask Example in the Chapter12 scene

You may note that the edges of the cat with the mask don't look great. To avoid this, ensure that you use sprites with appropriate image resolutions and try different filtering modes on the Sprite's **Import Settings**.

The Mask component

The **Mask** component can be added to any UI object with an **Image** component. If it is added to a UI object that doesn't have an **Image** component, it won't function since it needs an **Image** to determine the restricted area.

The **Mask** component can be added to a UI object by selecting **Add Component** | **UI** | **Mask** within the **Inspector**:

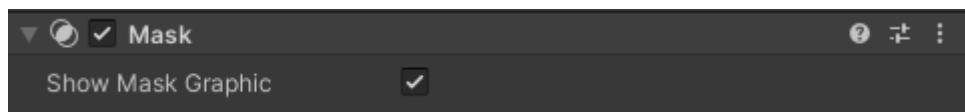


Figure 12.2: The Mask component

Any children of the UI object containing the **Mask** component will then have their visibility restricted to only the area within the opaque area of the **Source Image** on its **Image** component.

In the following image, the object named **Mask** is a UI Image with a **Mask** component added to it. As you can see, the purple triangle on the left of the Panel is only partially visible, while the green triangle on the right is fully visible. The green triangle is fully visible because it is not a child of the UI Image that contains a **Mask** component:

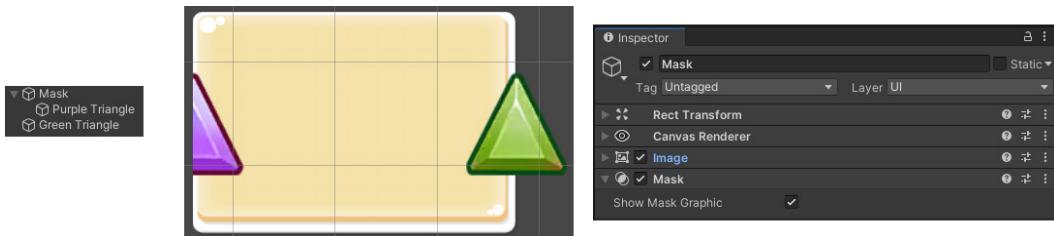


Figure 12.3: The Mask Component Example in the Chapter12 scene

You have the option to hide the **Source Image** that defines the **Mask** component's visibility area. If you deselect **Show Mask Graphic**, the parent's **Source Image** will not be visible. It's important to note that changing the opacity on the **Source Image** does not affect the **Mask** component's functionality.

Rect Mask 2D component

Using the **Mask** component allows you to restrict the visible area to a non-rectangular shape. However, if you want to restrict the visible area to a rectangular shape and don't want to use an image to restrict the visible area, you can use a **Rect Mask 2D** component.

The Rect Mask 2D component can be added to a UI Object by selecting **Add Component** | **UI** | **Rect Mask 2D** within the **Inspector**, as shown in the following:

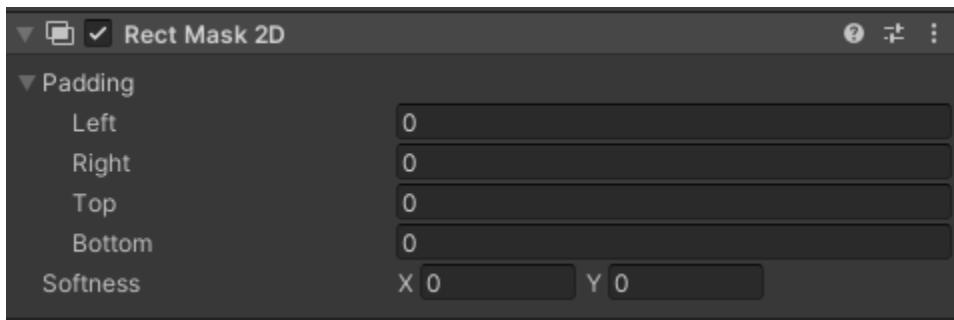


Figure 12.4: A Rect Mask 2D component

Notice that the component allows you to adjust **Padding** and **Softness**.

When a **Rect Mask 2D** component is added to a GameObject, the visibility of its children will be affected by the shape of its Rect Transform. An **Image** component is not required on the parent object for the **Rect Mask 2D** component to function properly.

In the following image, an Empty UI GameObject is created and a **Rect Mask 2D** component is added to it. It is then given a child UI Image. As you can see, the triangle is masked by the Rect Transform area:

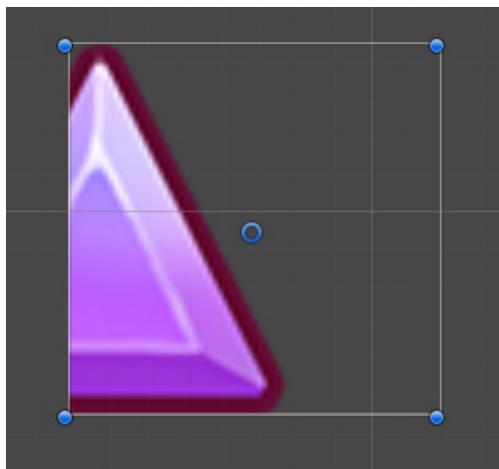


Figure 12.5: A Rect Mask 2D Component Example in the Chapter12 scene

If you want to apply a mask to a rectangular shape, I highly recommend that you use the **Rect Mask 2D** component rather than the standard **Mask** component, as it is more performant.

It's important to note that, as implied by the name **Rect Mask 2D**, this mask will only work on 2D objects. You can read more about its limitations at <https://docs.unity3d.com/Manual/script-RectMask2D.html>.

As I stated earlier, one common use for a mask is to create a menu with objects that *spill out* of the visible area. Creating these types of menus requires scrollbars and scroll views, so let's look at those components now.

Implementing UI Scrollbars

The **UI Scrollbar** object allows the user to drag a handle along a path. The position of the handle on the path affects the position of an image or object within a usable area.

If you're having trouble seeing what a scroll bar is from the preceding description, here's an easier explanation with context. It is most commonly used in video games with menus that have a lot of information within a viewable area that is smaller than the area that all the information takes up.

To create a UI Scrollbar, select + | UI | **Scrollbar**. By default, a UI Scrollbar has a child named **Sliding Area**. The **Sliding Area** also has a child named **Handle**.

The **Sliding Area** child is an empty GameObject. Its purpose is to ensure that its child, the **Handle**, is correctly positioned and aligned. The **Handle** is a UI Image. It represents the interactable area of the Scrollbar.

If you want to change the appearance of the Scrollbar's background and Handle, you need to change the **Source Images** of the **Image** components on the Scrollbar parent and Handle child, respectively.

The Scrollbar component

The parent Scrollbar object has a **Scrollbar** component. It has all the properties common to the interactable UI objects, along with a few that are exclusive to Scrollbars:

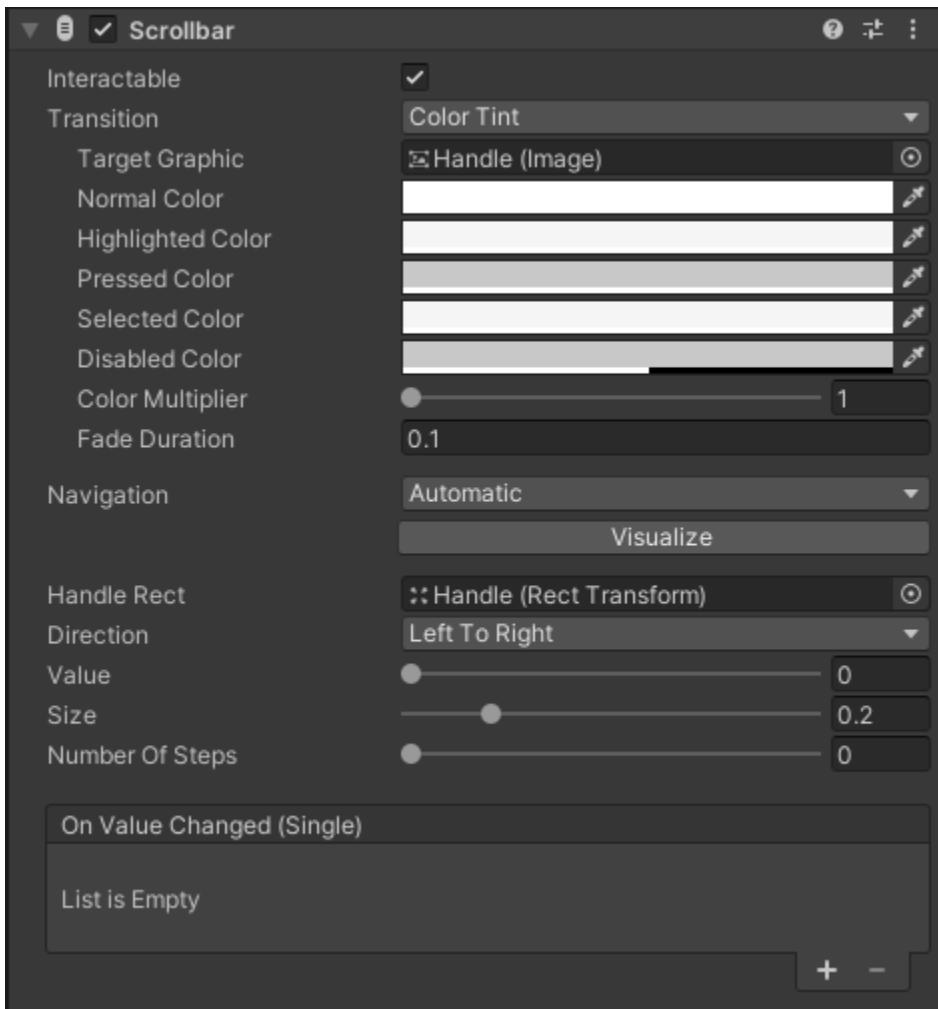


Figure 12.6: The properties of the Scrollbar component

A Scrollbar's **Value** is always restricted to being between 0 and 1. Scrollbars are used to move objects that take up more room than the viewable space. Due to this, the Scrollbar's position should easily translate to a percentage or a value between 0 and 1.

The **Handle Rect** property assigns the Rect Transform of the object that displays the handle's image. By default, the Rect Transform of Handle is assigned to this property. You'll note that the Rect Transform component on the Handle GameObject has the **Some values driven by Scrollbar** message, since the position of Handle is affected by the Scrollbar. The position of the Scrollbar's Handle is tied to the **Value** property of the **Scrollbar** component.

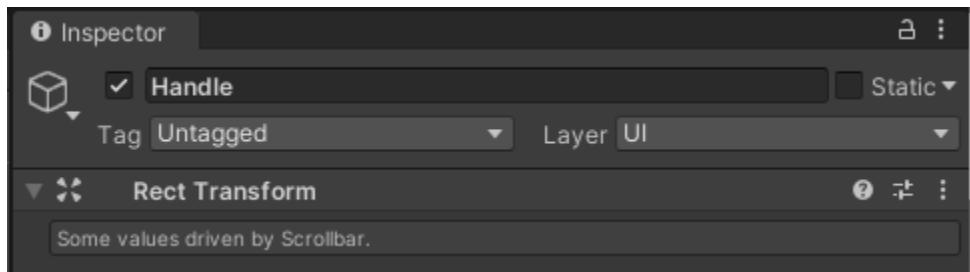


Figure 12.7: The Rect Transform component on the Handle GameObject

The **Direction** property allows you to select the orientation of the Scrollbar. The available options are the same as with a Slider and translate accordingly: **Left To Right**, **Right To Left**, **Bottom To Top**, and **Top To Bottom**.

The **Size** property determines the percentage of the Scrollbar's **Sliding Area** taken up by the Scrollbar's Handle. This can be any `float` value from 0 to 1. I recommend that you choose a value relative to the size of the objects being scrolled so that the Scrollbar's movement feels more intuitive. This means that the larger the scrollable area is, the smaller the Scrollbar Handle becomes.

The **Number of Of Steps** property is used if you want the Scrollbar to have staggered, discrete steps and don't want it to have continuously controlled movement. This is used if you want your scrollbar to move the scrolling objects to specific locations.

Often, you will see a scroll area represented by dots (like the dots that identify which home screen you are viewing on an iOS device). This can be achieved using a **Number Of Steps** property greater than 0. Setting this value to 0 will allow for continuously controlled movement rather than staggered discrete steps.

Examples of creating continuous and discrete Scrollbars are provided in the *Examples* section at the end of this chapter.

Scrollbar default event – On Value Changed (Single)

The Scrollbar component's default event is the **On Value Changed** event, as seen in the **On Value Changed (Single)** section of the Scrollbar component. This event will trigger whenever the Scrollbar's Handle is moved. It can accept a `float` argument.

When a public function has a `float` parameter, it will appear twice within the function's dropdown list of **On Value Changed (Single)** events: once within a **Static Parameters** list and again within the **Dynamic float** list, as shown in the following screenshot:

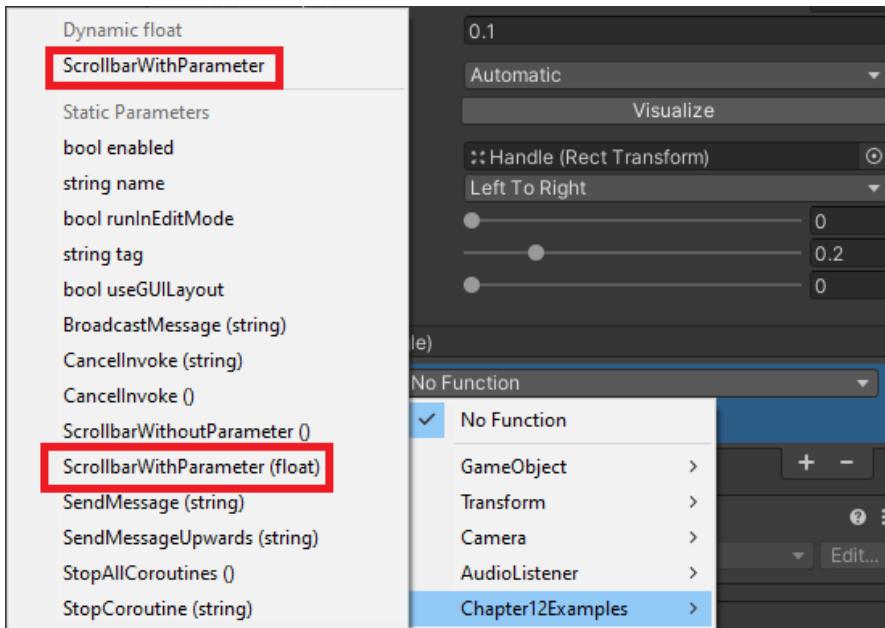


Figure 12.8: Static Parameters and Dynamic float methods

If the function is selected from the **Static Parameters** list, a box will appear that allows you to enter a `float` as an argument within the event. The event will then only send the value within that box. In the example shown in the following screenshot, the only value that will ever be sent to the `ScrollbarWithParameter()` function will be 0.

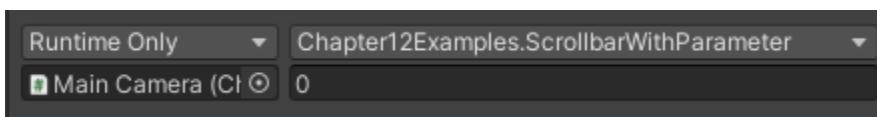


Figure 12.9: Example of a Static Parameters method in the Inspector

If you want the Scrollbar's value to be sent as an argument to a function that has a parameter, you must select the function from the **Dynamic float** list.

The following functions and image represent the Scrollbar example found in the Chapter12 scene that triggers events that call functions with and without parameters:

```
public void ScrollbarWithoutParameter() {
    Debug.Log("changed");
```

```

    }

    public void ScrollbarWithParameter(float value){
        Debug.Log(value);
    }
}

```

In the following screenshot, the third option shows the function chosen from the **Dynamic float** list and will send the value of the **Value** property as an argument to the function:

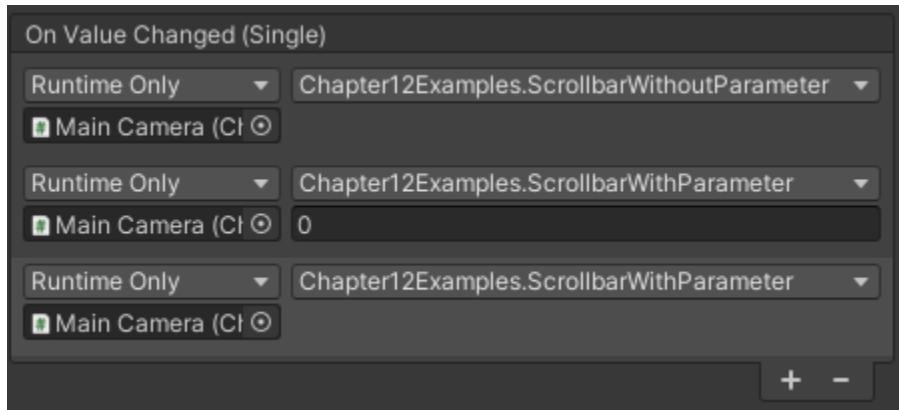


Figure 12.10: Events on Scrollbar Example in the Chapter12 scene

Now that we've reviewed how to implement a UI Scrollbar in Unity, let's look at UI Scroll Views.

Implementing UI Scroll View

The **UI Scroll View** object creates a scrollable area along with two child UI Scrollbars. The scrollable area can be scrolled using the scrollbars, by dragging the area within the Scroll View, or using the scroll wheel on the mouse.

To create a UI Scroll View, select + | UI | **Scroll View**. By default, a UI Scroll View has three children: **Viewport**, **Scrollbar Horizontal**, and **Scrollbar Vertical**. The **Viewport** object also has a child named **Content**. The **Scrollbar Horizontal** and **Scrollbar Vertical** children have the same parent/child relationship as default UI Scrollbars, as discussed in the previous section.

Even though the UI Scroll View comes with two **Scrollbar** children by default, you don't have to use both Scrollbars with your **Scroll View**. In fact, you don't have to use Scrollbars at all! Refer to the *Scroll Rect component* section for further details.

The Viewport child object is a UI Image with a **Mask** component. The **Mask** component of Viewport has the **Show Mask Graphic** property turned off, by default. As you can see from the Rect Transform in the following screenshot, the Viewport applies a mask to an area within the Scroll View:

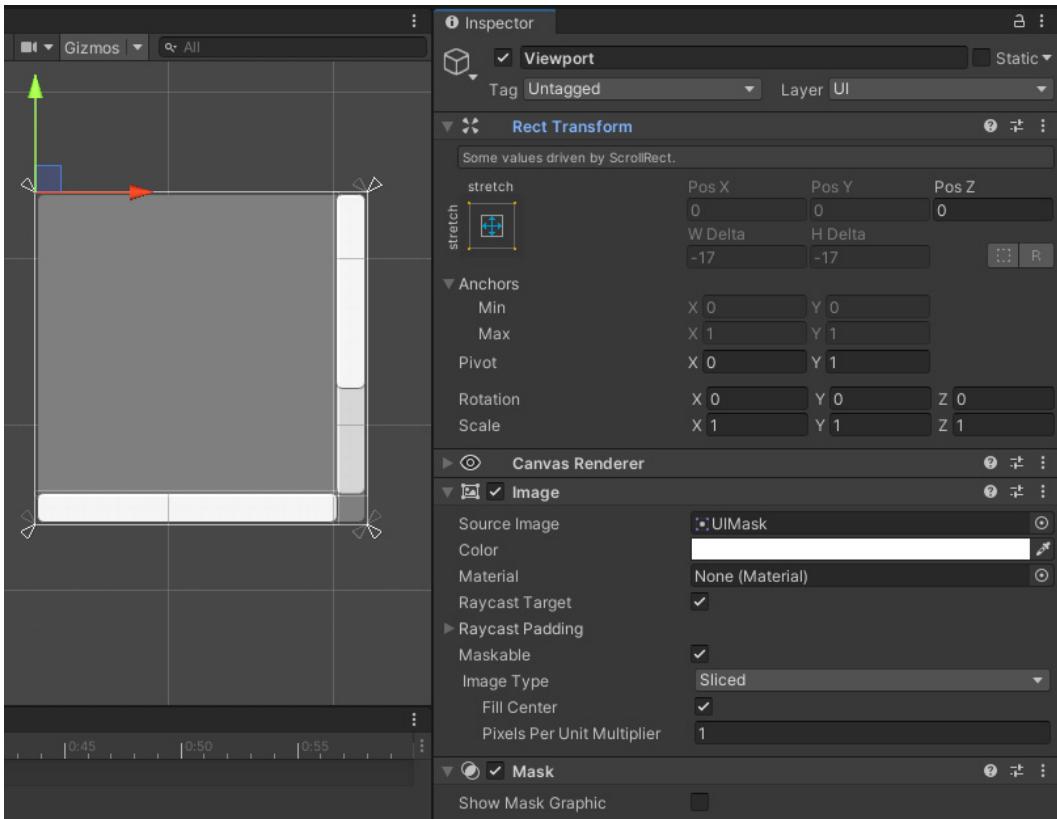


Figure 12.11: A UI Scroll View Viewport

This will make the items within the **Scroll View** only viewable within the defined area (between the Scrollbars). You cannot change most of the **Rect Transform** properties of the Viewport, because this area is set based on how you have the settings of your **Scroll View** set in its **Scroll Rect** component (refer to the *Implementing UI Scrollbar* section for more details).

The child of **Viewport** is an empty **Rect Transform** named **Content**. This will act as the holder of all the items you wish to place within **Scroll View**. You can think of the **Content** as the things that will be moving around within the **Scroll View**. As you can see from the following image, the **Rect Transform** of **Content** is larger than the viewable area defined by **Viewport**, since the objective of a **Scroll View** is to have items outside of the viewable area.

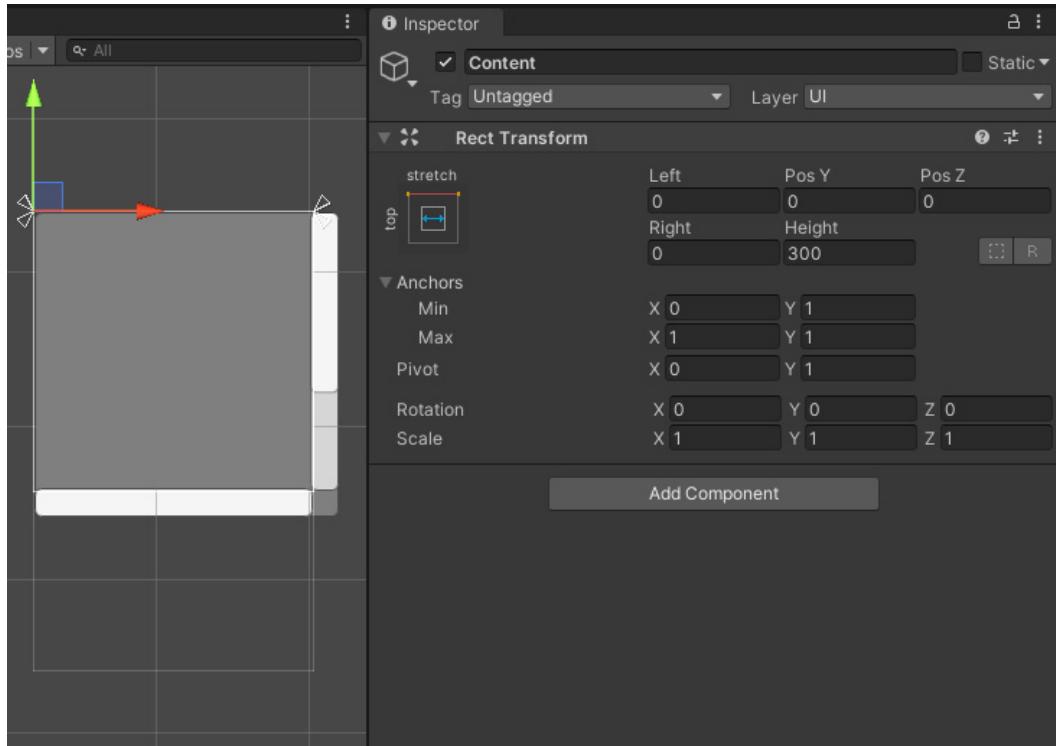


Figure 12.12: A UI Scroll View Content

To add items to the `ScrollView`, you simply add children to the `Content` object. Since `Content` is a child of `Viewport`, any of its children will also be affected by the `Mask` component on the `Viewport`.

The following example shows four images added as children of `Content`. `Content` has also been given a **Vertical Layout Group** component:

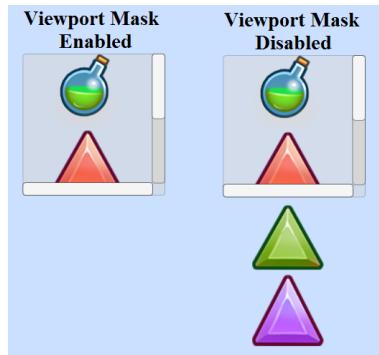


Figure 12.13: A Scroll View example in the Chapter13 scene

In the preceding image, you can see that when the **Mask** component is disabled from the **Viewport**, all items are visible. When you are setting up your **Scroll View**, I highly recommend that you disable the **Mask** on the **Viewport** so that you can see the general layout of the items you are placing and re-enable it when you are done laying out all the items.

You should also adjust the **Rect Transform** area of **Content** to enclose all of its child items or use a **Content Size Fitter** component so that you can more easily predict the behavior of **ScrollView**. If the area of the **Rect Transform** **Content** does not fully encompass all the items, scrolling the **ScrollView** may not show the items outside of its viewable area.

ScrollView contains an **Image** component. If you want to change the background of the **ScrollView** (the gray rectangle that encapsulates everything), change the **Source Image** of the **Image** component on the **ScrollView**. You can change the appearance of the **Scrollbar** **Horizontal** and **Scrollbar** **Vertical** children, as described in the *Implementing UI Scrollbar* section.

Now that we understand the intent and setup of a **Scroll Rect**, let's look at the individual properties of the **Scroll Rect** component.

Scroll Rect component

The behavior of the **ScrollView** is determined by the **Scroll Rect** component on the **ScrollView** parent object:

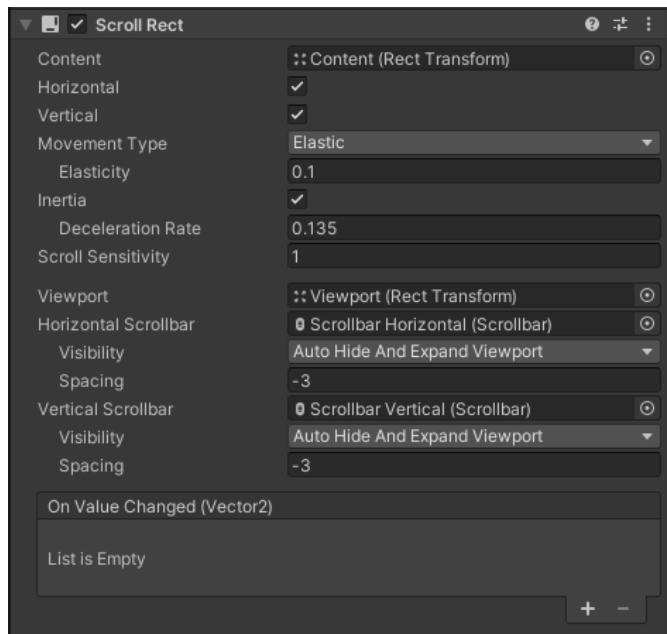


Figure 12.14: The Scroll Rect component

Note that the Scroll Rect component doesn't have the **Interactable**, **Transition**, or **Navigation** properties that all the other UI components in this chapter (and previous chapters) have!

I'll discuss the properties slightly out of order to make them easier to discuss.

The **Content** property is assigned the Rect Transform of the UI element that will scroll. When using the UI Scroll View object, this is set to the Rect Transform component of **Content** by default. The **Viewport** property is assigned the Rect Transform that is the parent of the item assigned to the **Content** property. When using the UI Scroll View object, this is set to the Rect Transform component of **Viewport** by default.

If you are creating a scrollable area without using the UI Scroll View, the **Viewport** and **Content** properties must be assigned and the Rect Transform assigned to **Viewport** must be a parent of the Rect Transform assigned to **Content**, for the **Scroll Rect** component to function properly.

Movement properties

There are three properties related to how the **Content** in the **Scroll View** will move. The **Horizontal** and **Vertical** properties enable and disable scrolling in the horizontal and vertical directions, respectively. By default, they are both enabled. The **Movement Type** property determines how the **Scroll View** moves at its boundaries.

There are three **Movement Type** options: **Unrestricted**, **Elastic**, and **Clamped**. These **Movement Types** only affect the way the scrollable area reacts to being dragged by the scrollable area and to the scroll wheel on the mouse. You will not note a difference between these three **Movement Types** when you are using the Scrollbars. All **Movement Types** will behave exactly like the **Clamped Movement Type** when the Scrollbars are used to move the **Content**.

When **Unrestricted** is selected as the **Movement Type** the player can drag the scrollable area endlessly without restriction to the Rect Transform of **Content**. The **Elastic** and **Clamped Movement Types** will stop moving the **Content** once the edges of Rect Transform of **Content** have been reached. However, when **Unrestricted** is selected as a **Movement Type**, the player can continue to drag or use the scroll wheel. If there is a mask on the **Viewport**, this can result in the player dragging all content outside of the viewable space. If the player moves the content with the scrollbars, however, they will be restricted to the bounds of **Content**.

When **Elastic** is selected as the **Movement Type**, if the **Content** is dragged past its boundary, it will bounce into place once the player stops dragging. This will also bounce if the scroll wheel is used. When **Elastic** is selected, the subproperty **Elasticity** becomes accessible. The **Elasticity** property determines the intensity of the bounce.

When **Clamped** is selected as the **Movement Type**, the **Content** will not be draggable past its boundary and no bounce will occur.

Properties concerning scrolling speed

Selecting the **Inertia** property will make the Content continue to move after the player has stopped dragging. **Inertia** is only apparent when the scroll area is dragged and doesn't affect Content movement initialized by the Scrollbars or mouse scroll wheel. When **Inertia** is selected, the **Deceleration Rate** subproperty becomes accessible. The **Deceleration Rate** property determines when the Content will stop moving after the player has ceased dragging. A **Deceleration Rate** of 0 will stop the Content the instant the player stops dragging, and a **Deceleration Rate** of 1 will never stop. By default, **Deceleration Rate** is set to 0.135.

The **Scroll Sensitivity** property determines how far the Content will move with each turn of the scroll wheel. The higher the number, the further the content will move with a turn, making it appear to move more quickly.

If you want to disable the use of the mouse scroll wheel for the Scroll View, set **Scroll Sensitivity** to 0.

Properties of the Scrollbars

You can set properties for the way your horizontal and vertical scrollbars react separately. The **Horizontal Scrollbar** and **Vertical Scrollbar** properties assign the **Scrollbar** components of the UI Scrollbars you wish to use for each. By default, these are assigned the **Scrollbar Horizontal** child and **Scrollbar Vertical** child, respectively.

If you only want to use drag on the Scroll View area and don't want scrollbars, you can simply set the **Horizontal Scrollbar** and **Vertical Scrollbar** properties to None or delete the **Scrollbar Horizontal** and **Scrollbar Vertical** objects from the scene.

Under each Scrollbar assignment, you can set the **Visibility** and **Spacing** properties of the respective Scrollbar.

The **Visibility** property has three options: **Permanent**, **Auto Hide**, and **Auto Hide And Expand Viewport**. When **Permanent** is selected for the **Visibility** property, the respective Scrollbar will remain visible, even if it is not needed, if its corresponding movement is allowed. For example, as shown in the following screenshot, if **Horizontal** movement is allowed, and the **Horizontal Scrollbar's** **Visibility** is set to **Permanent**, the respective scrollbar will be visible, even though it is not necessary (no horizontal movement can be achieved):

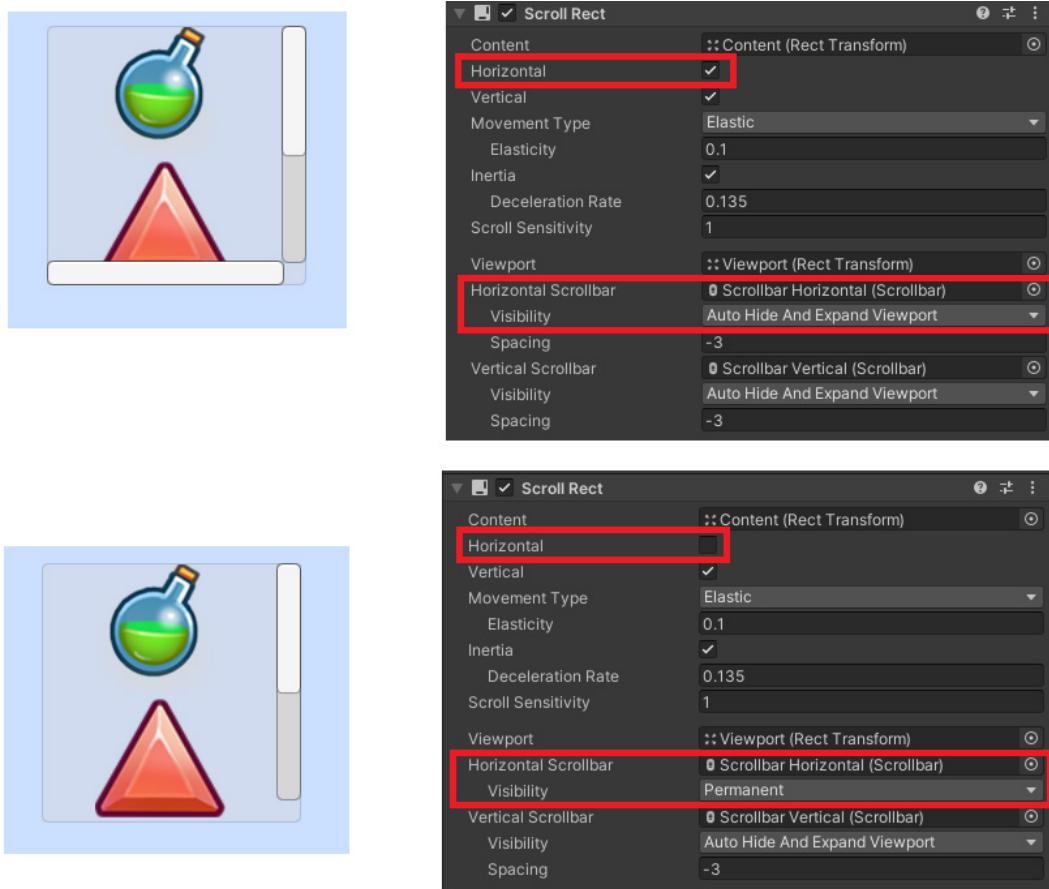


Figure 12.15: Adjusting the Scrollbars in the Scroll Rect

However, referencing the same image, you can see that if **Horizontal** movement is deactivated, setting the **Horizontal Scrollbar's Visibility** to **Permanent** removes it from the Scroll View entirely. It is also deactivated in the **Hierarchy**.

When **Auto Hide** is selected for the **Visibility** property, the respective Scrollbar will become invisible and deactivate in the **Hierarchy** when the game is played if it is not needed (meaning that there is no movement in that direction required) or if the respective axis movement is disabled.

The **Auto Hide And Expand Viewport** property works in the same way as **Auto Hide**, but it will also expand the area of the Rect Transform assigned to **Viewport**. If the **Viewport** object has a **Mask** component, this will cause the mask's area to expand to cover the area that was initially being taken up by the Scrollbar.

The **Spacing** property determines the space between the **Viewport**'s Rect Transform and the **Scrollbar**'s Rect Transform. By default, this value is set to -3, which means the two Rect Transforms overlap slightly. If you want to change the position of the **Viewport**, you have to do so with this property, since the properties related to the **Viewport**'s position are disabled in its Rect Transform component.

Scroll Rect default event – On Value Changed (Vector2)

The Scroll Rect component's default event is the **On Value Changed** event, as seen in the **On Value Changed (Vector2)** section of the Scroll Rect component. This event will trigger whenever the Content area of the Scroll View is moved by dragging, scrolling with the mouse scroll wheel, or scrolling with one of the Scrollbars. It accepts a Vector2 position as an argument and, as with the other events discussed in this chapter, you can choose to pass no argument, a static argument, or a dynamic argument.

If you want to send the Vector2 position of Content to a function, you'd send it to a function with a Vector2 parameter from the **Dynamic Vector2** list. The position sent is essentially a percentage, with the starting position having a value of 1 . 0 in the corresponding coordinate and the last position having a value of 0 . 0 in the corresponding coordinate.

Let's consider the following function:

```
public void ScrollViewWithParameter(Vector2 value)
{
    Debug.Log(value);
}
```

If the previous function is selected from the **Dynamic Vector2** list, the following image shows the Vector2 values printed in the **Console**:

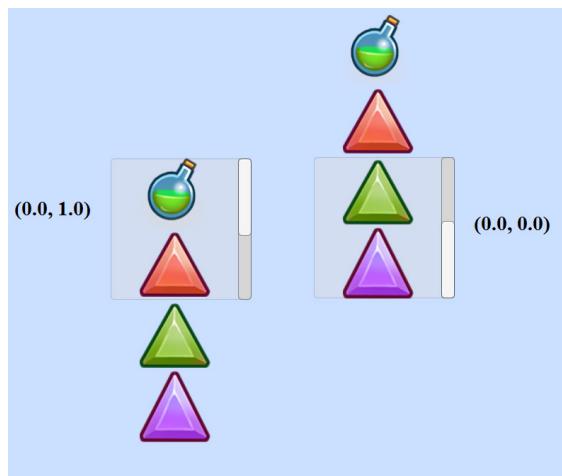


Figure 12.16: Values of the Scroll Rect based on position

Now that we've looked at the various UI elements, let's look at some examples of implementing them.

Examples

I mentioned earlier in this chapter that a common usage of masks and scroll views is a menu with multiple items in it. So, for this chapter, we'll only create one example, a scroll view from a pre-existing menu, by mimicking the layout of the Scroll View UI object.

Making a scroll view from a pre-existing menu

To help organize the project, duplicate the `Chapter11-Examples` scene that you created in the last chapter; rename it `Chapter12-Examples`. Open `Chapter12-Examples` and complete the following example within that scene.

I want to be able to add more items to my `Inventory Panel` and allow the player to scroll through the items. I could create a new UI Scroll View item and update it to look like my current `Inventory Panel`, but that would be more hassle than it's worth. So, instead, I will convert the current `Inventory Panel` to a scrollable view. After it is complete, it will look like the following figure:



Figure 12.17: The scrollable menu we will build

This Panel can have its view adjusted by dragging the area beside the food items. I could have added a vertical scrollbar to control the movement, but I wanted to show you how simple it is to make a draggable area without one. It also just looks a little nicer without a scrollbar.

To make the `Inventory` Panel scrollable, complete the following steps:

1. Right now, the `Pause` Panel is in the way. Let's disable it by deselecting the checkbox in its `Inspector` so that we can see the `Inventory` Panel more easily.
2. For the scrollable view to really work, we need more items in our inventory. Select all the children of `Inventory Holder` named `Item Holder` and duplicate them with `Ctrl + D`. Now, with all the duplicates selected, rename them `Item Holder`, so we don't have the numbered suffix added to each of them. You should now have twice as many items in your `Inventory Holder`, and you should see the following:

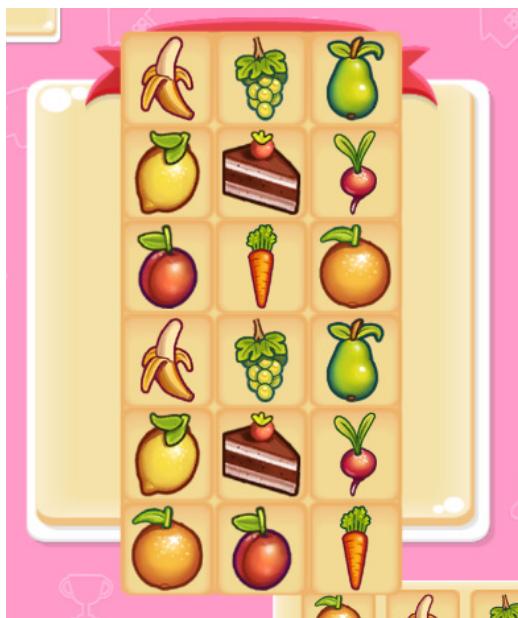


Figure 12.18: The result after Step 2

3. To get a scrollable view, we need to add a few more items to the scene that can be parents of the content we want to scroll. There is a very specific parent/child relationship you want to follow with each item having the specific components to create a scrollable view, as demonstrated in the following diagram:

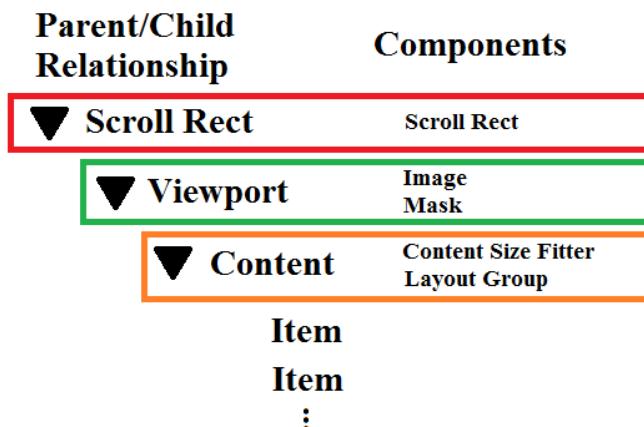


Figure 12.19: The Hierarchy layout to create a scrollable menu

Let's start with the object that will hold the **Scroll Rect** component. Select **Inventory Panel**, right-click, and select **Create Empty**. This will create an empty object that is a child of our **Inventory Panel**.

4. Rename the new item **Scroll Rect** and reposition it in the **Hierarchy** so that it is below **Inventory Banner**.
5. Give the **Scroll Rect** GameObject the **Scroll Rect** component, with **Add Component | UI | Scroll Rect** in its Inspector.
6. Now, let's create the item that will hold the **Mask** component. Remember that the **Mask** component must be on an object that also has an **Image** component; so, let's create a UI Image. Right-click on **Scroll Rect** and select **UI | Image** to create a child of **Scroll Rect**.
7. Rename the new Image **Viewport**.
8. **Inventory Holder** is visually in the way right now, so disable it momentarily.
9. Add the **uiElements_38** image to the **Image** component of **Viewport**.
10. Adjust the Rect Transform of **Scroll Rect** so that it stretches to fill its parent's Rect Transform.

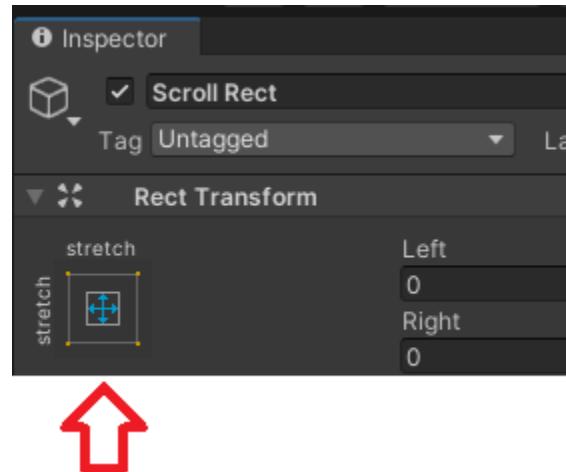


Figure 12.20: The Rect Transform of the Scroll Rect object

11. Adjust the Rect Transform properties of Viewport, as follows:

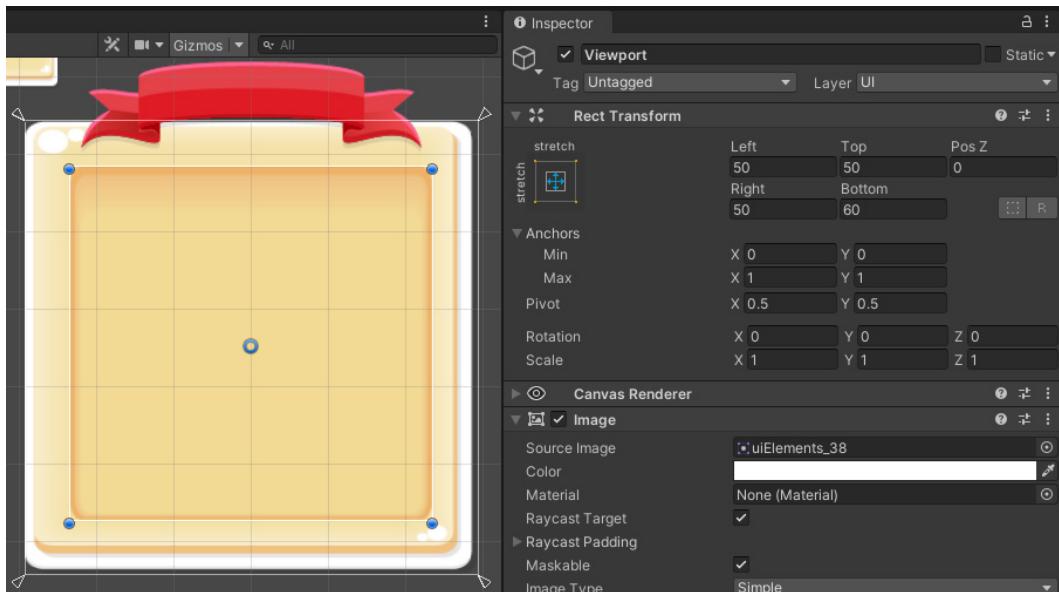


Figure 12.21: The Rect Transform of the Scroll Rect object

12. Give Viewport the Mask component, with **Add Component** | **UI** | **Mask**.
13. Re-enable Inventory Holder and rename it Content.
14. Set the alpha of the **Color** property on Content to 0 so that the background is invisible. You should see the following:

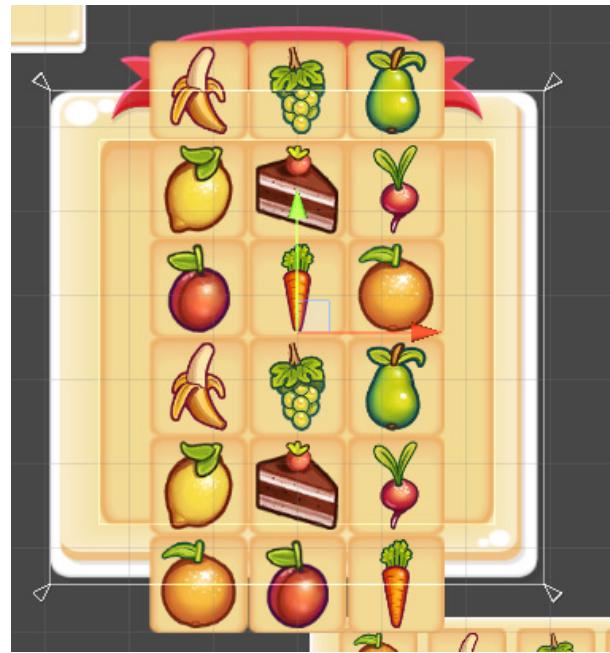


Figure 12.22: The results of Step 11

15. Now, drag Content in the **Hierarchy** so that it is a child of Viewport. Doing so should make the mask instantly apply to all of the children of Content:



Figure 12.23: The results of Step 12

16. Position Content so that the food at the top of the list appears fully within the Mask area:



Figure 12.24: How to position the Content

17. Now, all that is left to do is to have a properly functioning scrollable area to set up the properties on the **Scroll Rect** component. Select **Scroll Rect** and drag **Content** and **Viewport** into their appropriate slots on the **Scroll Rect** component.
18. Disable **Horizontal** movement because we only want the menu to move vertically. Set **Movement Type** to **Clamped** so that the menu doesn't stretch and bounce and stays within the appropriate bounds. Your **Scroll Rect** component should now look as follows:

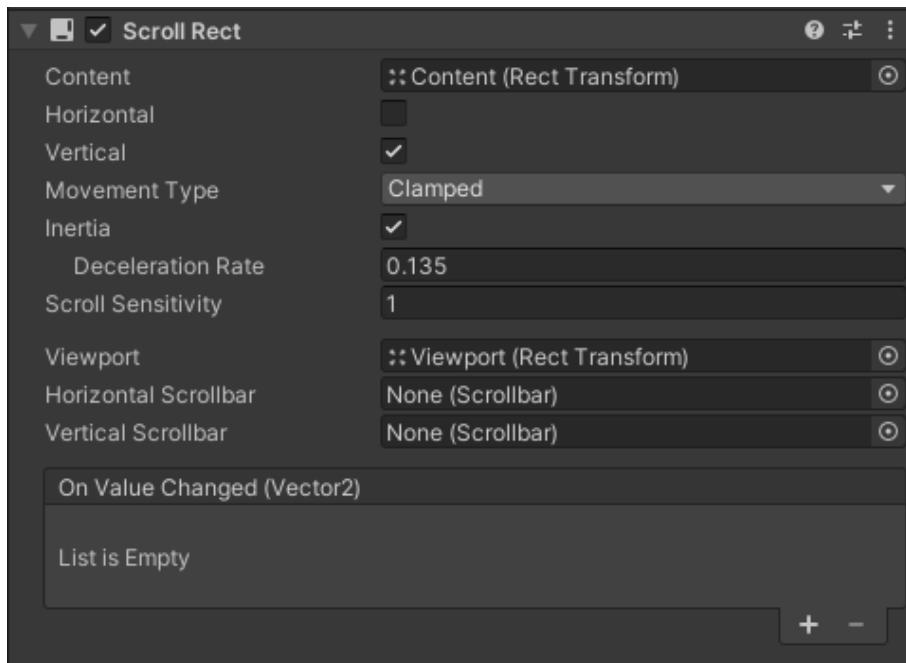


Figure 12.25: Assigning the Content and Viewport

19. Re-enable the `Pause Menu` object so that our `Pause Menu` will function in the game.

If you play the game, the `Inventory Panel` should now have items that scroll when you drag beside them. Remember that you bring up the `Inventory Panel` by pressing the `I` key on your keyboard. As we have drag and drop functionality on our individual items, to scroll, we have to drag the area of `Content` that does not have a food item on it.

Summary

This chapter covered how to hide the visibility of UI elements as well as use Scroll Views to create containers that hold more items than the screen can hold at one time. There are still quite a few more interactable UI elements, however. In the next chapter, we will look at the remaining interactive UI elements provided by the Unity UI system.

13

Other Interactable UI Components

The most popular interactable UI components are Buttons. However, there are multiple types of interactable UI elements other than buttons. If you think of an online form you've filled out recently, you've probably interacted with buttons, text fields, and possibly a radio button or checkbox. While technically all of these interactable items can be developed with UI Buttons, UI Text, and some custom code, you don't have to build them yourself! Unity has included, within the uGUI system, multiple commonly used interactable UI items both as GameObjects that you can edit and as components that you can add to pre-existing GameObjects.

This chapter will review all the other pre-built UI items that come with the uGUI system. After having reviewed the chapters on buttons and text, most of these objects' properties will be familiar to you, but each interactable item has a few properties exclusive to that UI item type, so we'll focus on those properties.

In this chapter, we will discuss the following topics:

- Using UI Toggles
- Using UI Sliders
- Using UI Dropdowns and Dropdown – TextMeshPros
- Using UI Input Fields and Input Field – TextMeshPros
- Creating a dropdown menu with images

Note

All the examples shown in this section can be found within the Unity project provided in the code bundle. They can be found within the scene labeled Chapter13.

Each example image has a caption stating the example number within the scene.

In the scene, each example is on its own Canvas, and some of the Canvases are deactivated. To view an example on a deactivated Canvas, simply select the checkbox next to the Canvas' name in the **Inspector**. Each Canvas is also given its own Event System. This will cause errors if you have more than one Canvas activated at a time.

Technical requirements

You can find the code for this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2013>

Using UI Toggle

The **UI Toggle** object is an interactable checkbox with a label. To create a UI Toggle, select + | **UI | Toggle**.

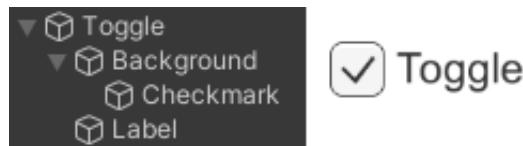


Figure 13.1: UI Toggle GameObject and children

By default, a UI Toggle has two children: a **Background** and a **Label**. The **Background** also has a child, a **Checkmark**.

The **Background** child is a UI Image that represents the “box” in which the **Checkmark** UI Image appears. The **Label** is a UI Text object.

If you want to change the appearance of the box and checkmark, you change the source images of the Image components on the **Background** and **Checkmark** children, respectively.

Toggle component

The parent Toggle object has a **Toggle** component. The Toggle component looks very similar to the Button component and has many of the same properties. As you'll see in this chapter, the first few properties of all interactive UI objects are the same. The properties at the bottom of the component are the ones that are exclusive to the UI Toggle object (*Figure 13.2*):

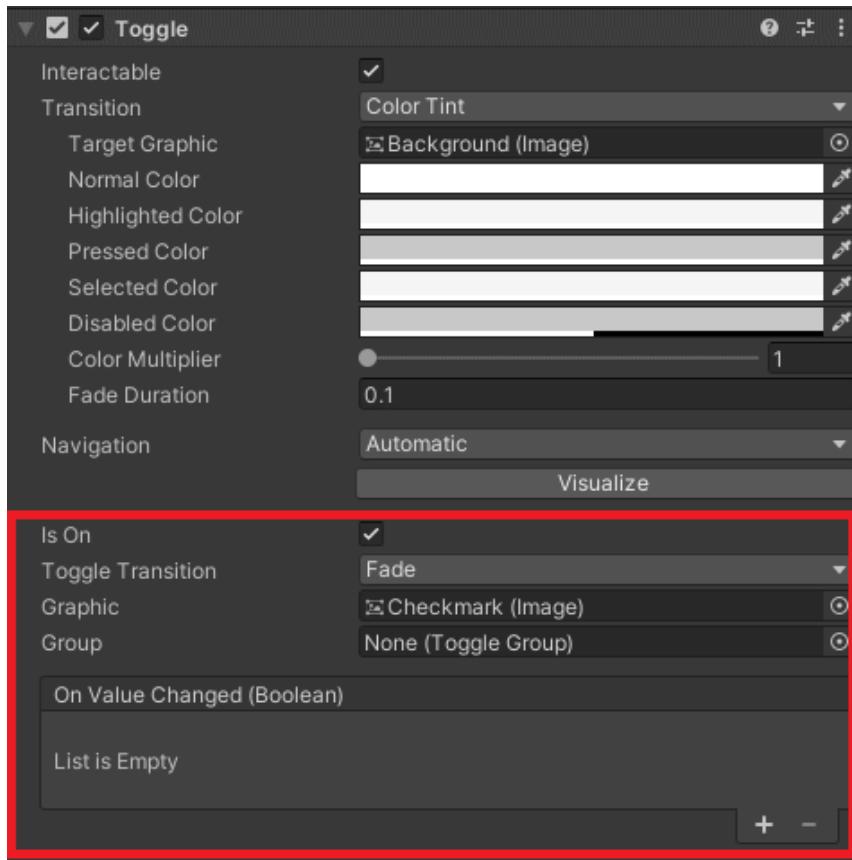


Figure 13.2: The Toggle component unique properties

The **Is On** property determines whether the Toggle is checked or not when it is initialized in the scene.

The **Toggle Transition** property determines what happens when the toggle transitions between on and off or checked and not checked. The two options are **None** and **Fade**. The **None** transition will instantaneously toggle between the checkmark Image being visible and not visible while the **Fade** transition will have the checkmark Image fade in and out.

The **Graphic** property assigns the **Image** component that will display the checkmark. The Checkmark child's Image component is automatically assigned to this property, but you can change it if you so desire.

The last property, **Group**, assigns the **Toggle Group** component that will define which Toggle Group the Toggle belongs to (if any).

The **Toggle** component's default Event is the On Value Changed Event, as seen in the **On Value Changed (Boolean)** section.

Toggle default event – On Value Changed (Boolean)

The **Toggle** component's default event is the **On Value Changed Event**, as seen in the **On Value Changed (Boolean)** section of the **Toggle** component. This event will trigger whenever the toggle is selected or deselected. It can accept a Boolean argument.

When a public function has a Boolean parameter, it will appear twice within the function's dropdown list of **On Value Changed (Boolean)** events: once within a **Static Parameter** list and again within the **Dynamic bool** list, as shown in the following screenshot:

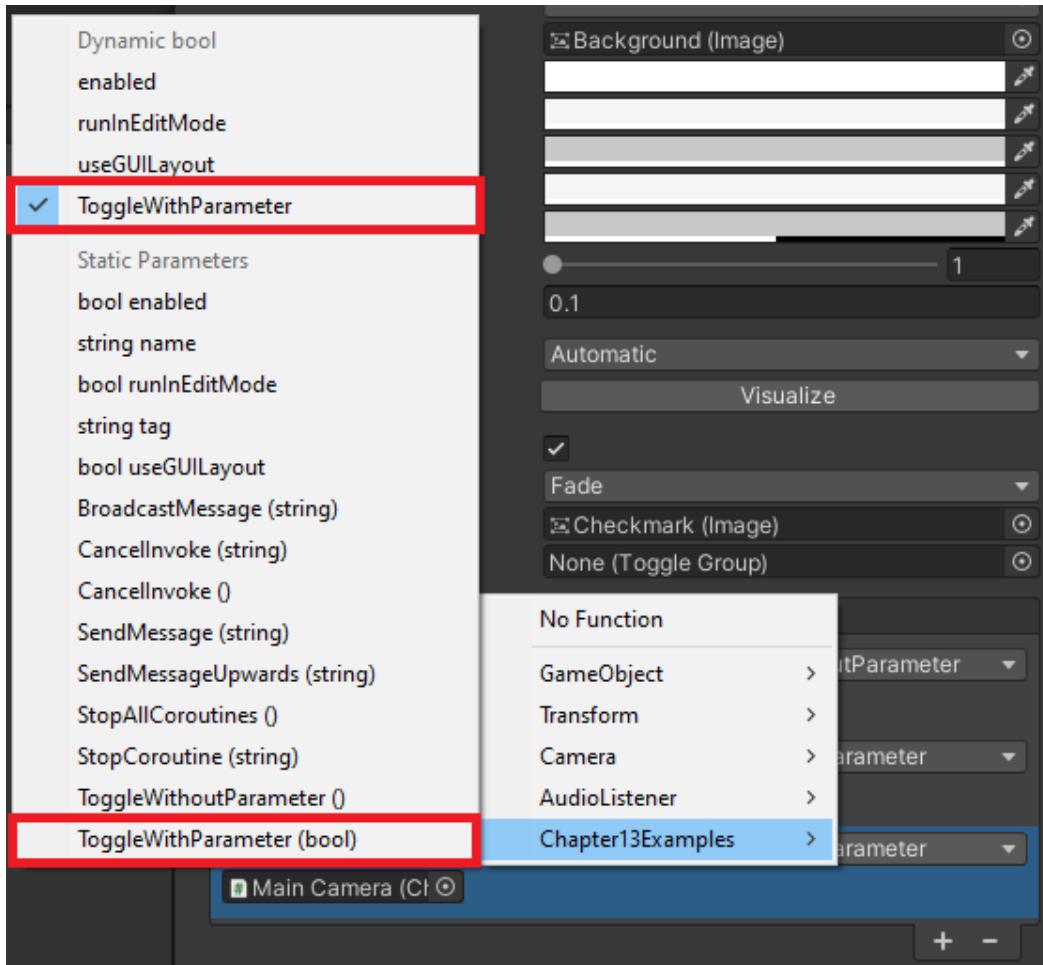


Figure 13.3: The static and dynamic versions of the `ToggleWithParameter` method

If the function is selected from the **Static Parameters** list, a checkbox will appear as an argument within the Event. The event will then only send the value of that checkbox. In the example shown in

in the preceding screenshot, the only value that will ever be sent to the `ToggleWithParameter()` function will be false (since the checkbox is deselected).

If you want to pass the `.isOn` value of the **Toggle** to the script, the function must be chosen from the **Dynamic bool** (not **Static Parameters**) list in the function dropdown of the event.

To demonstrate how **On Value Changed (Boolean)** events work, let's see how the following two functions respond when called:

```
public void ToggleWithoutParameter() {
    Debug.Log("changed");
}

public void ToggleWithParameter(bool value) {
    Debug.Log(value);
}
```

The following events are added to a **Toggle** in the Chapter13 scene:

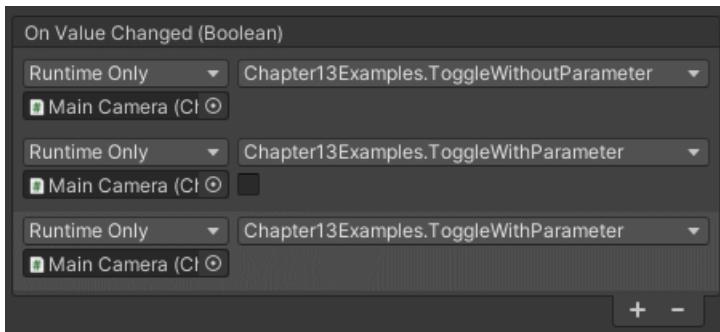


Figure 13.4: Events on the Toggle Event Example in the Chapter13 scene

When the **Toggle** within the scene is deselected, the following will print in the **Console**:

```
changed
False
False
```

When the **Toggle** is selected, the following will print in the **Console**:

```
changed
False
True
```

Since the function called from the first event does not have a parameter, it will always execute when the value of the **Toggle** changes, regardless of what the value of the **Toggle** is when executed.

The second event will always print the value of `False`, because the function has a parameter and, since the event was chosen from the **Static Parameters** list, the argument being sent is represented by the checkbox, which is set to `False`. So, the value `False` is always sent to the function.

The third event's function was chosen from the **Dynamic bool** list, so the argument sent to the function will be the `.isOn` value to which the Toggle changes.

Toggle Group component

The **Toggle Group** component allows you to have many UI Toggles that work together, where only one can be selected or *on* at a time. When Toggles are in the same Toggle Group, selecting one Toggle will turn *off* all others. For the Toggle Group to work properly at the start, you should either set all the Toggles within the Toggle group's **Is On** property to `False` or set only a single Toggle's **Is On** property to `True`.

The **Toggle Group** component does not create a renderable UI object, so attaching it to an empty GameObject will not create any visible element.

Once the **Toggle Group** component is attached to a GameObject, the GameObject it is attached to must be dragged into the **Group** property of each of the Toggles that will be contained within the group:

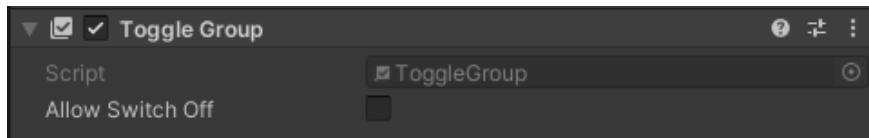


Figure 13.5: The Toggle Group component properties

There is only one property on the **Toggle Group** component: **Allow Switch Off**. The **Allow Switch Off** property allows the Toggles to be turned off if they are selected when already in the on state. Remember that the Toggle Group component forces at most one Toggle on at a time. So, the **Allow Switch Off** property being turned off forces there to be at least one Toggle selected at all times.

My suggestion when using this component is to use an empty GameObject that acts as the parent for all the Toggles you wish to group together. This empty GameObject will then contain the Toggle Group component (as demonstrated in the **Toggle Group Example** in the Chapter13 scene). The object containing the Toggle Group component must then be assigned to the **Group** property on the **Toggle** component of each of the Toggle children.

UI Slider

The UI Slider object allows the user to drag a handle along a path. The position on the path corresponds to a range of values.

To create a UI Slider, select **+** | **UI** | **Slider**. By default, a UI Slider has three children: a **Background**, a **Fill Area**, and a **Handle Slide Area**. The Fill Area also has a child, **Fill**, and the Handle Slide Area has a child, **Handle**.

The **Background** child is a UI Image that represents the full area that the Slider's Handle can traverse. In the default Slider example, this is the darker gray background area that gets filled.

The **Fill Area** child is an empty GameObject. Its main purpose is to ensure that its child, the **Fill**, is correctly aligned. The **Fill** is a UI Image that stretches across the Fill Area based on the Slider's value. In the default Slider example, this is the light gray area that trails behind the handle and fills in the **Background**.

The **Handle Slide Area** child is also an empty GameObject. Its purpose is to ensure that its child, the **Handle**, is correctly positioned and aligned. The **Handle** is also a UI Image. The **Handle** represents the interactable area of the Slider.

If you want to change the appearance of the Slider, you change the **Source Image** of the **Image** components on the **Background**, **Fill**, and **Handle** children.

Slider component

The parent Slider object has a **Slider** component. It has all the properties common to the interactable UI objects along with a few that are exclusive to Sliders, as highlighted in the following figure:

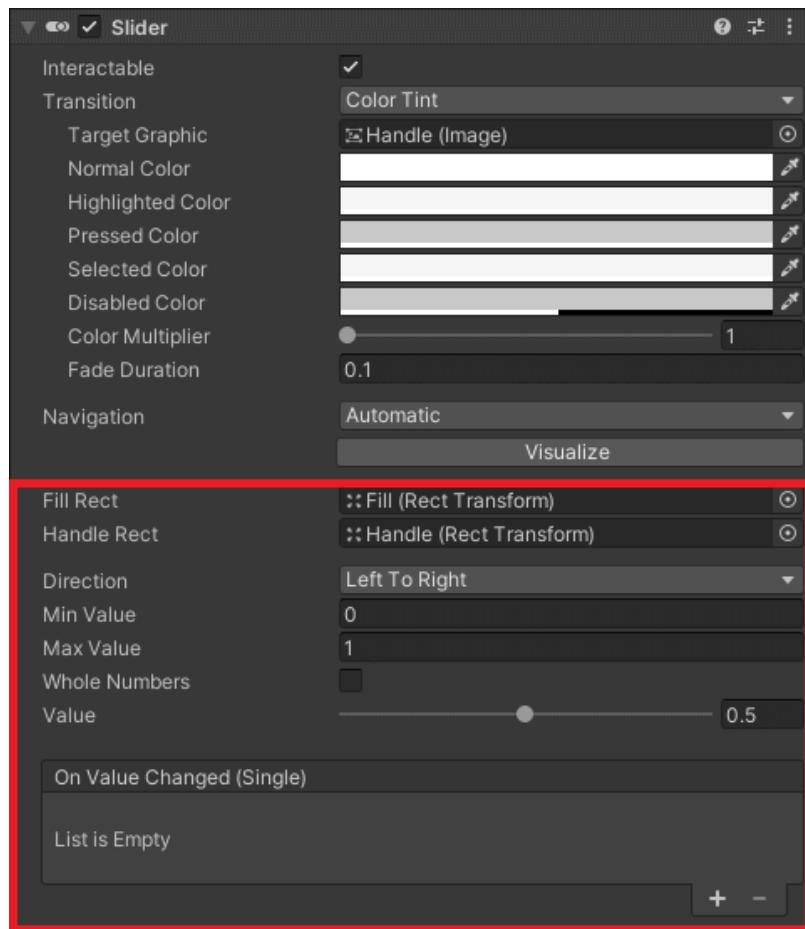


Figure 13.6: The unique properties of the Slider component

The **Fill Rect** property assigns the Rect Transform of the object that displays the Image of the filled area. By default, this is the Fill GameObject's Transform component. You'll note that on the Rect Transform component of the Fill, a message stating **Some values driven by Slider** is displayed. This indicates that the values are changed based on the **Slider** component. While playing the scene, if you move the Handle of the Slider, you will not see the Rect Transform properties of the Fill update. However, if you make the Scene view visible while moving the Handle in the **Game** view, you will see the Rect Transform area of the Fill change as you affect the slider.

The **Handle Rect** property assigns the Rect Transform of the object that displays the handle's image. By default, the Rect Transform of Handle is assigned to this property. You'll note that the Rect Transform component on the Handle GameObject also has the **Some values driven by Slider** message since the position of the Handle is affected by the Slider.

The range of values that the Slider represents is determined by the **Min Value** and **Max Value** properties. You can assign any value to the **Min Value** and **Max Value** properties, even negative numbers. While the **Inspector** allows you to define the **Min Value** as a number larger than the **Max Value**, the Slider will not work properly if you do so.

The **Direction** property allows you to select the orientation of the Slider. The available options are **Left To Right**, **Right To Left**, **Bottom To Top**, and **Top to Bottom**. The order of the positions in each direction represents the first position (or **Min Value**) and then the last position (or **Max Value**) of the Slider's value range.

If the **Whole Numbers** property is selected, the range of values the Slider can represent will be restricted to integer (non-decimal) values.

Note

As I am a math teacher, I feel the need to point this out. In math, the term Whole Numbers represents all non-negative Integers (0 through infinity). Here, in the Slider component, the term Whole Numbers represents all Integers, even negative ones. So, if you're a math nerd like me, don't let this imply to you that the Slider cannot hold negative values if the **Whole Numbers** property is selected.

The **Value** property is the value of the Slider. The position of the Slider's Handle is tied to this property. The slider in the **Inspector** next to the **Value** property is a one-to-one representation of the Slider in the scene.

Slider default event – On Value Changed (Single)

The Slider component's default event is the On Value Changed event, as seen in the **On Value Changed (Single)** section of the Slider component. This event will trigger whenever the Slider's Handle is moved. It can accept a float argument.

If you want the Slider's value to be sent as an argument to a function that has a parameter, you must select the function from the Dynamic float list (similar to selecting functions from the Toggle's Dynamic bool list).

The following functions and screenshot represent a Slider example found in the Chapter7Text scene that triggers events that call functions with and without parameters:

```
public void SliderWithoutParameter() {
    Debug.Log("changed");
}

public void SliderWithParameter(float value) {
    Debug.Log(value);
}
```

In the following screenshot, the third option shows the function chosen from the **Dynamic float** list and will send the value of the **Value** property as an argument to the function:

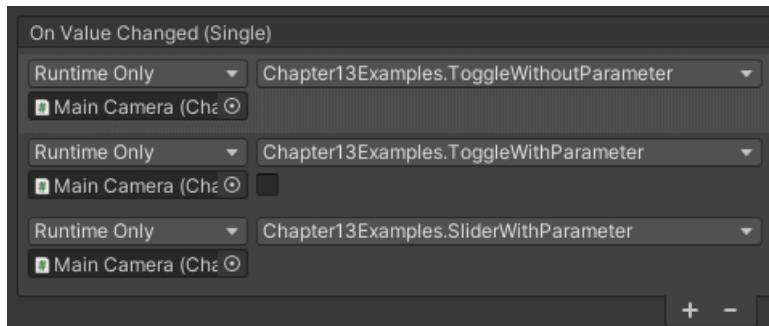


Figure 13.7: Events on Slider Example in the Chapter13 Scene

Note

It's important to note that if the **Whole Numbers** property is selected and the Slider can only hold integer values, the functions called by this event will receive those integers as float values.

Now that we've reviewed how to use Sliders, let's review how to use the two types of Dropdowns.

UI Dropdown and Dropdown – TextMeshPro

There are two Dropdown UI objects available, the UI Dropdown object packaged in Unity and the Dropdown—TextMeshPro object. Both the Dropdown objects allow the user to select from a list of options. The list becomes visible when the Dropdown is clicked on. Once an object is selected from the list, the list will collapse, making the chosen option visible within the Dropdown (if desired).

The two Dropdown options are pretty much identical in the way they work. The only difference between the two is the UI Dropdown uses UI Text objects to display text while the Dropdown—TextMeshPro uses Text - TextMeshPro objects. Due to this, I will discuss the two objects at the same time in this section. Additionally, because the two objects are identical in function, you will need to use Dropdown—TextMeshPro over the UI Dropdown if you want to include “fancy” text.

To create a UI Dropdown, select + | **UI | Dropdown**. To create a Dropdown—TextMeshPro, select + | **UI | Dropdown - TextMeshPro**. As you can see in the following screenshot (*Figure 13.8*), the two Dropdown objects have identical parent/child object relationships and names. By default, the Dropdown objects have three children: a Label, an Arrow, and a Template. The Template child is disabled by default (hence, it appears grayed out in the Hierarchy) and has multiple children.

The Template child and all of its children are discussed in the *Dropdown Template* section of this chapter.

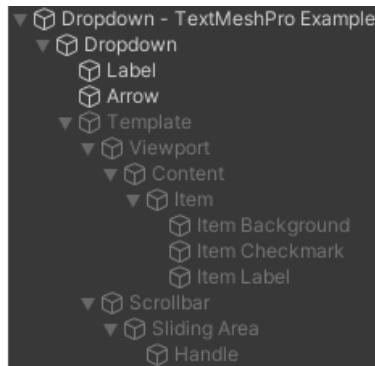
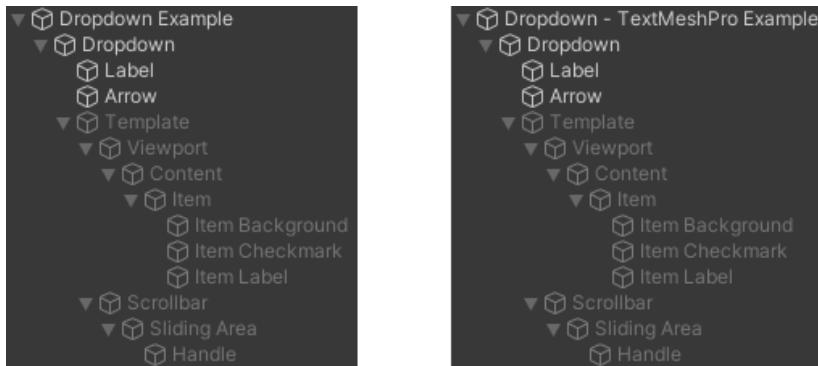


Figure 13.8: The Hierarchy of the two types of dropdowns

In the following paragraphs, I will discuss all Text objects as if they are UI Text objects. However, remember that the Dropdown—TextMeshPro uses TextMeshPro - Text objects.

The Label child is a UI Text object. By default, it displays the text within the Dropdown object that represents the selected option. As the player changes the selected option, the **Text** property of the **Text** component of Label changes to the appropriate option. To change the properties of the text that displays within the boxed area of the Dropdown, change the properties of the **Text** component on the Label. When new text replaces the text within the Label, it will automatically display based on the properties set by the Text component of the Label.

The Arrow child is a UI Image. Its only function is to hold the image for the arrow that (by default) appears at the right of the Dropdown. This arrow doesn't actually do anything and is simply an image. It doesn't accept inputs or change with the properties of the Dropdown component.

The background image of the Dropdown is on the main Dropdown parent object and not on a child named Background. Therefore, if you want to change the appearance of the Dropdown's background and Arrow, you change the **Source Images** of the Image components on the Dropdown parent and Arrow child, respectively. The image of the Dropdown only affects the rectangle that can be selected to display the dropdown menu. The background to the menu that expands outward when the player interacts with the dropdown is handled by the Template (discussed in the following *Dropdown Template* section).

Dropdown Template

Before we discuss the various properties of the Dropdown component, let's look more closely at Dropdown's **Template**.

The child of Dropdown named Template allows you to set the properties of the “items” that will appear as options in the dropdown menu. It also allows you to set the properties of the background of the menu and the Scrollbar that will appear if the list expands past the viewable area of the dropdown menu.

Remember that the Template child is disabled by default. Enabling the Template (by selecting the checkbox in its **Inspector**) will display the Template in the scene.

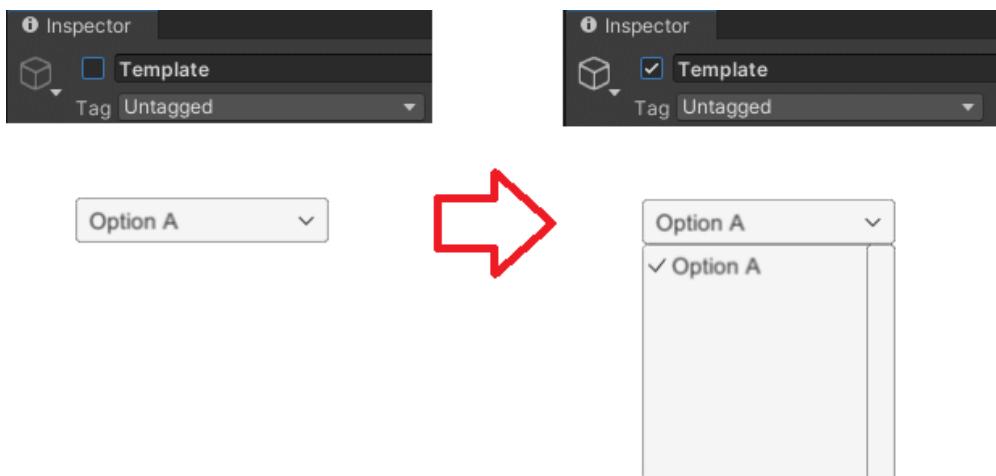
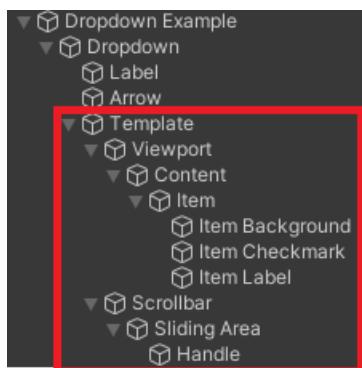


Figure 13.9: Enabling the Template GameObject of the Dropdown

You can leave this permanently enabled in your Editor because once you enter **Play** mode, it will hide as it is supposed to.

If you look closely at the parent/child relationships of the Template within the **Hierarchy**, you’ll note that it is simply a UI Scroll View object with one Scrollbar:



UI Scroll View with One Scrollbar

Figure 13.10: The UI Scroll View with One Scrollbar within the Dropdown

Viewing the **Inspector** of the Template GameObject, shows that it, in fact, is just a UI Scroll View object, as it has a **Scroll Rect** component attached to it with no **Horizontal Scrollbar** assigned:

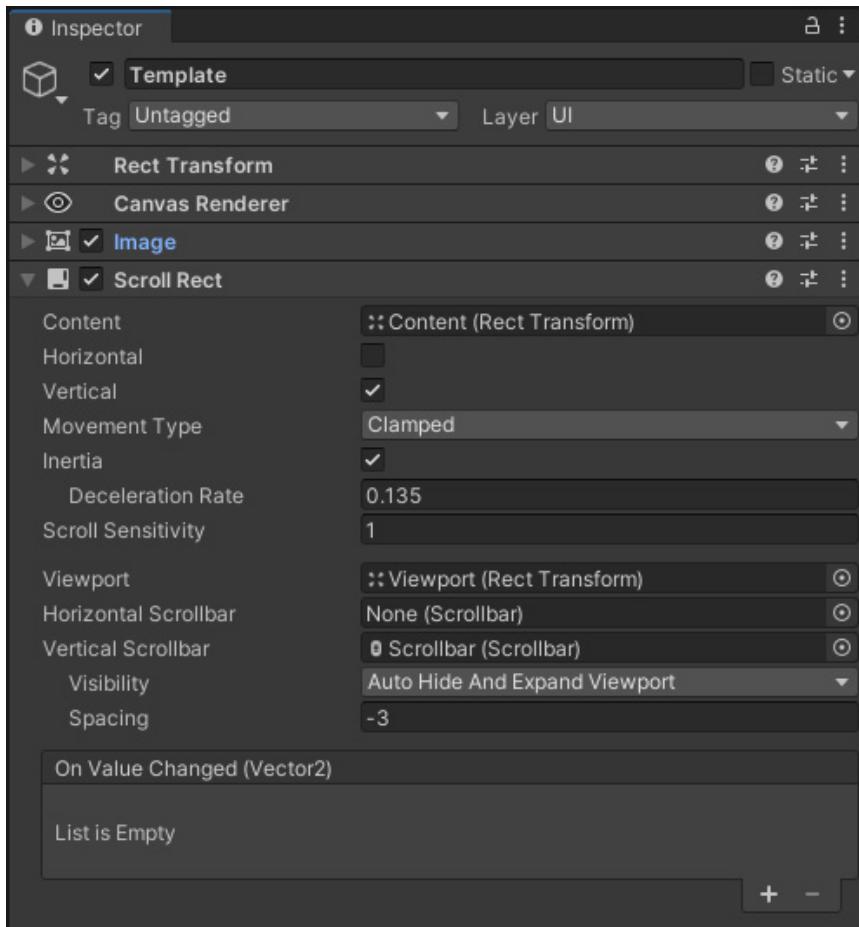


Figure 13.11: The Scroll Rect component of the Template

The **Content** of the Template Scroll View object has a single child named **Item**. **Item** has three children: **Item Background**, **Item Checkmark**, and **Item Label**. If you look at the **Inspector** of **Item**, you'll see that it is just a UI Toggle and has the same children and properties as the UI Toggles discussed at the beginning of this chapter.

So, all `Template` is a Scroll View with a single Scrollbar and with a single Toggle as its Content! It looks way more complicated initially, but after you break down what the individual pieces are, you'll realize that it's just a combination of a few of the UI items we have already discussed!

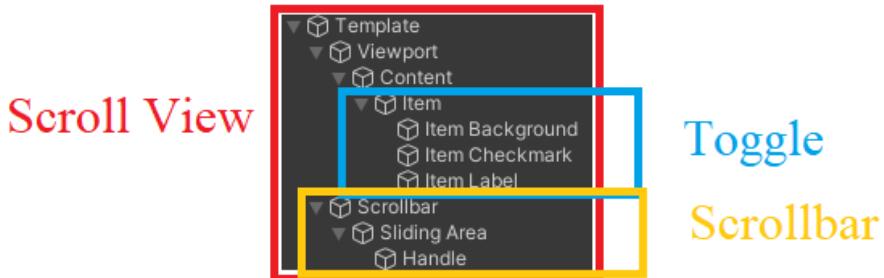


Figure 13.12: Breakdown of the Template's children

When working with the Dropdown Template, if you want to change the visual properties and the settings, just remember the breakdown shown in the preceding figure, and the prospect of editing it will seem a lot less daunting.

Every item option you set to appear within the Dropdown will follow the exact same visual properties of those you set for the Item Toggle.

The Dropdown component

Now that we've broken down the Template, we can look at the properties of the **Dropdown** component.

The parent Dropdown object has a **Dropdown** (or `Dropdown - TextMeshPro`) component. It has all the properties common to the other interactable UI objects, along with a few that are exclusive to Dropdowns:

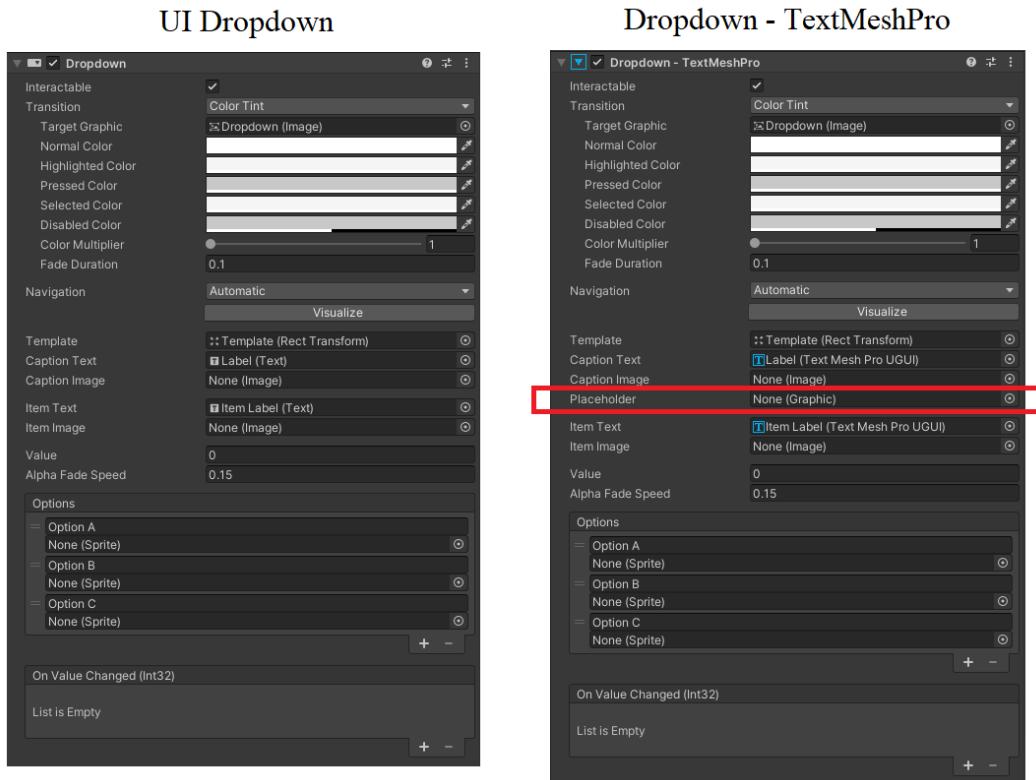


Figure 13.13: The difference between the two Dropdown components

As you can see from the preceding image, the properties of the **UI Dropdown** and **Dropdown - TextMeshPro** are nearly identical. There are only two main differences. First, **UI Dropdown** objects use **UI Text** objects, while **Dropdown - TextMeshPro** objects use **Text - TextMeshPro** objects. Second, **Dropdown - TextMeshPro** has a **Placeholder** property.

The **Dropdown** component is actually super powerful. It handles all interactions with the **Dropdown** menu and will switch text displays, open and close the dropdown, and move around the checkbox within the dropdown. It even adds a scrollbar and handle to allow the dropdown menu to have a really long list. The only thing that must be coded by you is how to interpret the player selects.

Let's review the various properties of the two **Dropdown** objects.

Caption properties

There are two properties related to the caption or the option that is currently selected (*Figure 13.3*).

The **Caption Text** property references the Text component of the GameObject that will display the currently selected option's text. By default, this is the Text component of the Label child. The **Caption Text** is optional, so if you do not want the currently selected option displayed within the Dropdown area (and only within the Dropdown list), simply change the **Caption Text** property to None (Text).

The **Caption Image** property holds the Image component of the GameObject that will display the currently selected option's image. Nothing is assigned to this property by default and, you will note that the **Dropdown** does not have a child that can hold the Image. To have an image display with the text, you will have to create a UI Image and assign it to the **Caption Image** property. It is best that the UI Image you create is created as a child of the Dropdown.

Template properties

There are three properties related to assigning the template's properties to all possible options to display in the dropdown list.

The **Template** property references the Rect Transform of the template. As stated previously, the template defines the way each option within the dropdown list will look as well as how the dropdown holder will look. By default, this property is assigned to the child Template object.

The **Item Text** property references the **Text** component of the GameObject that holds the text of the item template. By default, the **Text** component on the Item Label (child of the Item) is assigned to this property.

The **Item Image** property references the **Image** component of the GameObject that holds the image of the item template. By default, this property is unassigned, similar to the **Caption Image**. Just as with the **Caption Image**, to use this property, a UI Image will need to be created and assigned to this property. If you create one, ensure that you add it as a child of **Item** within the **Template** child to avoid confusion.

Option properties

The **Value** property represents which option is currently selected. The options are in a list, and the number in the **Value** property represents the currently selected option's index within the list. Since the options are represented by their indices, the first option has a **Value** of 0 (not 1).

The **Options** property lists out all the options within the **Dropdown** menu. Within the list, each option has a text string and sprite (optional). All strings and sprites within this list will automatically swap into the correct component properties of the children of **Dropdown**, based on the properties of the **Dropdown** component. So, you will not have to write any code to ensure that these items display appropriately when the player interacts with the **Dropdown**.

By default, the **Options** list contains three options. However, you can add or subtract options by selecting the plus and minus sign at the bottom of the list. You can also rearrange the options within the list by dragging and dropping the options' handles (two horizontal lines). Note that rearranging the options in the list will change their indices within the list and then change the **Value** they send to the **Dropdown** component.

Dropdown default event – On Value Changed (Int32)

The Dropdown component handles all interactions with the Dropdown menu itself. The only thing that has to be coded by you is how to interpret the option the player selected.

The Dropdown component's default event is the **On Value Changed** Event, as seen in the **On Value Changed (Int32)** section of the **Dropdown** component. This event will trigger whenever a new option is selected by the player. It accepts an integer as an argument and, as with the other events discussed in this chapter, you can choose to pass no argument, a static argument, or a dynamic argument.

If you want to send the index of the option selected (or the value of the **Value** property) to a function, you'd send it to a function with a Int32 parameter from the **Dynamic int** list. Refer to the *Creating a dropdown menu with images* example at the end of the text for an implementation of this.

The next interactable UI component we'll review is the UI Input Field.

UI Input Field

The UI Input Field provides a space in which the player can enter text.

To create a UI Input Field, select + | UI | **Input Field**. By default, the InputField GameObject has two children: a Placeholder and a Text object.

The Placeholder child is a UI Text object that represents the text displayed before the player has input any text. Once the player begins entering text, the Text component on the Placeholder GameObject deactivates, making the text no longer visible. By default, the text displayed by the Placeholder is **Enter text...**, but the text being displayed as well as its properties are easily changed by affecting the properties on the **Text** component of the Placeholder.

The Text child is a UI Text object that displays the text the player inputs. Setting the properties on the Text object's **Text** component will change the display of the text entered by the player.

InputField contains an **Image** component. If you want to change the appearance of the input box, change the **Source Image** of the **Image** component on the InputField.

Input Field component

The parent InputField object has an **Input Field** component. It has all the properties common to the interactable UI objects along with a few that are exclusive to Input Fields:

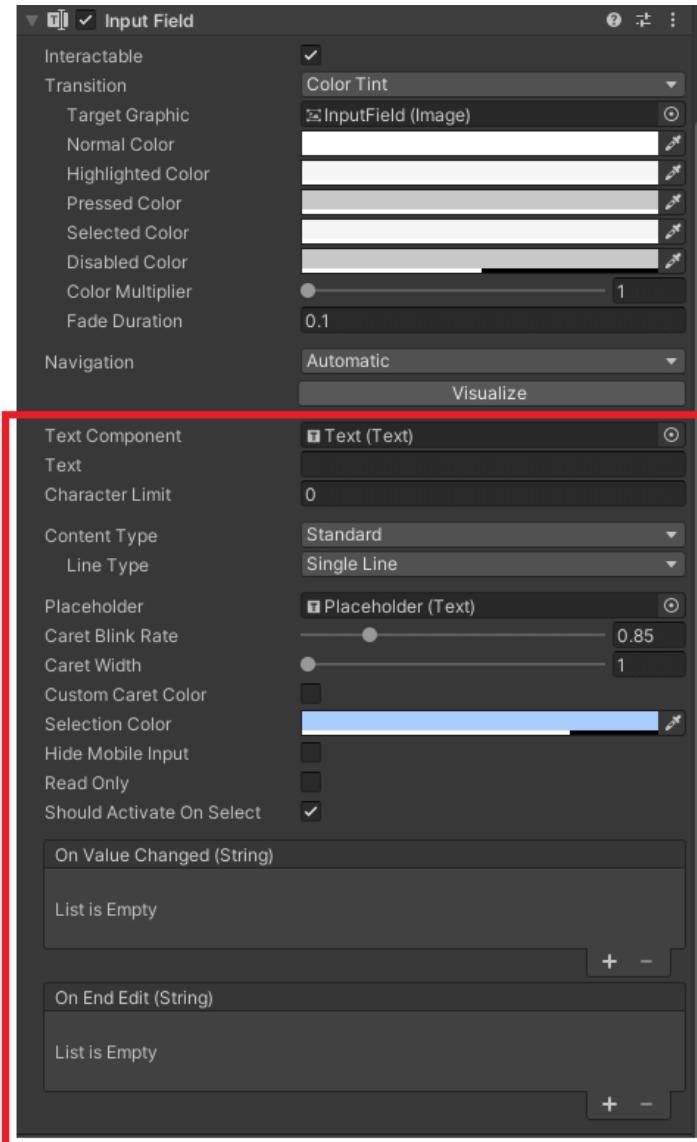


Figure 13.14: The unique properties of the Input Field component

Let's look at the various properties of the UI Input Field.

Properties of entered text and onscreen keyboards

Many of the properties within the **Input Field** component affect the text that displays within the Input Field. Due to some of the properties having a lot of options and information pertaining to them, I will discuss them slightly out of order.

Remember that to change the visual style of the entered text, you need to change the properties of the **Text** component on the Text GameObject.

The **Text** Component property references the Text component of the GameObject that will display the player's entered text. By default, this is the **Text** component of the Text child.

The **Text** property is the text currently entered into the Input Field. When you are attempting to retrieve the data from the Input Field, you want to get the information from this property and not from the **Text** component on the Text GameObject. The Text component on the Text GameObject will only store what is currently being displayed. So, if the text is displayed as asterisks because it's a password or has scrolled, the full and correct text will not be stored in the **Text** component of the Text GameObject.

The **Character Limit** property allows you to specify the maximum number of characters the player can enter into the field. Leaving the **Character Limit** property set to 0 allows unlimited text entry.

The **Placeholder** property references the **Text** component of the GameObject that displays the text when the player has either not entered anything or has cleared all entered text. By default, this is the **Text** component of the **Placeholder** child.

The **Hide Mobile Input** property allows you to override the default mobile keyboard that pops up when a text **Input Field** is selected. You will select this option if you have your own keyboard that you want the player to use. Currently, this only works for iOS devices.

If you wanted to use your own keyboard on an Android device, your best bet would be to create your own custom input field script. The script would show a keyboard when the input box is selected and then change the text within the box based on the custom keyboard key presses.

The **Read Only** property makes the text within the **Input Field** static and uneditable by the player. The player can still select the text to copy and paste it when this property is activated.

When the **Read Only** property is selected, the text displayed by the Input Field can still be edited via code by accessing the **Text** property on the Input Field component. However, changing the **Text** property on the **Text** component of the Text GameObject, will not change the displayed text.

Content Types

The **Content Type** property determines the types of characters that will be accepted by the Input Field. On devices that display keyboards on screen, it also affects the keyboard that is displayed by the device when the input field is selected. If the desired keyboard is not available, the default keyboard will be displayed. For example, if the device does not have a numbers-only keyboard, it will display

the default keyboard. For more detailed explanations of each keyboard and character validations that come with these Content Types, refer to the *Keyboard Types* and *Character Validation Options* sections.

The possible options are **Standard**, **Autocorrected**, **Integer Number**, **Decimal Number**, **Alphanumeric**, **Name**, **Email Address**, **Password**, **Pin**, and **Custom**.

The **Standard** option allows any character to be entered. Note, however, that characters not available for the entered text's font will not display.

The **Autocorrected** option works like the **Standard** option, but on devices with onscreen keyboards (particularly touchscreen keyboards), it allows the device's autocorrect functionality to automatically override words based on its own autocorrecting algorithms.

The **Integer Number** option allows only integer values (positive and negative numbers without decimals). The player will be restricted from entering more than one negative symbol. The **Decimal Number** option works similarly, except that it also accepts decimal points. The player will be restricted from entering more than one decimal point. On devices with onscreen keyboards (particularly mobile devices), the numeric keyboard will appear rather than the standard keyboard with these two options.

The **Alphanumeric** option only allows letters (uppercase and lowercase) along with numbers and input. Mathematical symbols and punctuation, including negative numbers and decimal points (periods), are not accepted.

The **Name** option will automatically capitalize each new word entered within the field. The player has the option to lowercase the first letter of a word by deleting the letter and re-entering it in lowercase.

The **Email Address** option will allow the player to enter an email address. It will also restrict the player from entering more than one @ symbol or two consecutive periods (dots/decimals).

The **Password** option allows letters, numbers, spaces, and symbols entered in the field. When the player enters text into a **Password** Input Field, the entered text will be hidden from view and displayed as asterisks (*).

The **Pin** option allows only integer numbers (no decimals) to be entered. Negative numbers are accepted. The text entered by the player in a field with the **Pin Content Type** will be hidden in the same way the **Password** option hides the player input. On an onscreen keyboard device, the numeric keyboard will be displayed with the **Pin** option.

The final option, **Custom**, gives you the most control of the type of input. When selected, new properties appear in the **Inspector** allowing you to select the **Line Type**, **Input Type**, **Keyboard Type**, and **Character Validation**.

Line Types

The **Line Type** option is available with the **Standard**, **Autocorrect**, and **Custom** options for **Content Type**. There are three **Line Type** options: **Single Line**, **Multi Line Submit**, and **Multi Line New Line**. All other **Content Types** are automatically restricted to **Single Line** types. If the player is allowed to enter more text than the Input Field's visible area can display (meaning the **Character Limit** property does not restrict it to the visible space), the text will scroll based on the **Line Type** selected.

The **Single Line** option only allows the entered text to be displayed on one line. If the text exceeds the visible horizontal space, the text will scroll horizontally. If the player hits the *Enter* key, the Input Field acts as if the text has been submitted.

The **Multi Line Submit** and **Multi Line New Line** options allow the text to overflow vertically if it exceeds the visible horizontal space and scroll vertically if the text exceeds the visible vertical space. The difference between the two options is what happens when the *Enter* key is hit: **Multi Line Submit** will submit the text and **Multi Line New Line** will start a new line.

Input Types

When the **Custom Content Type** is selected, you have the option to select from three Input Types: **Standard**, **Autocorrect**, and **Password**.

Selecting these various **Input Types** does not change the keyboard or provide any validation, like the similarly named **Content Types**. For example, the **Password Input Type** will accept the *Enter* key as a new line with **Multi Line New Line** and display it as an asterisk in the field but accept it as a new line in the actual data stored in the **Text** property.

The **Standard** option does not put any special circumstances on the type of input.

The **Autocorrect** option applies to platforms with onscreen keyboards that have built-in autocorrect functionality. This option allows the device's autocorrect to change the text as it sees fit.

The **Password** option will display the text as asterisks.

Keyboard Types

When the **Custom Content Type** is selected, you have the option to select **Keyboard Types**. On devices with onscreen keyboards, this property allows you to select which keyboard will display when the Input Field is selected.

The possible options are **Default**, **ASCII Capable**, **Numbers And Punctuation**, **URL**, **Number Pad**, **Phone Pad**, **Name Phone Pad**, **Email Address**, **Social**, **Search**, **Decimal Pad**, and **One Time Code**.

If the keyboard selected is not available on the target device, the device's default keyboard will be displayed.

The **Default** option displays the device’s default (letters) keyboard. On most devices, this keyboard only displays letters, the *Space* key, *Backspace* key, and *Return (Enter)* key. When this option is selected, the player will have the ability to switch to the keyboard with numbers and punctuation keys.

For example, the iOS English default keyboard and numbers and punctuation keyboard can easily be switched between, as shown in the following figure:

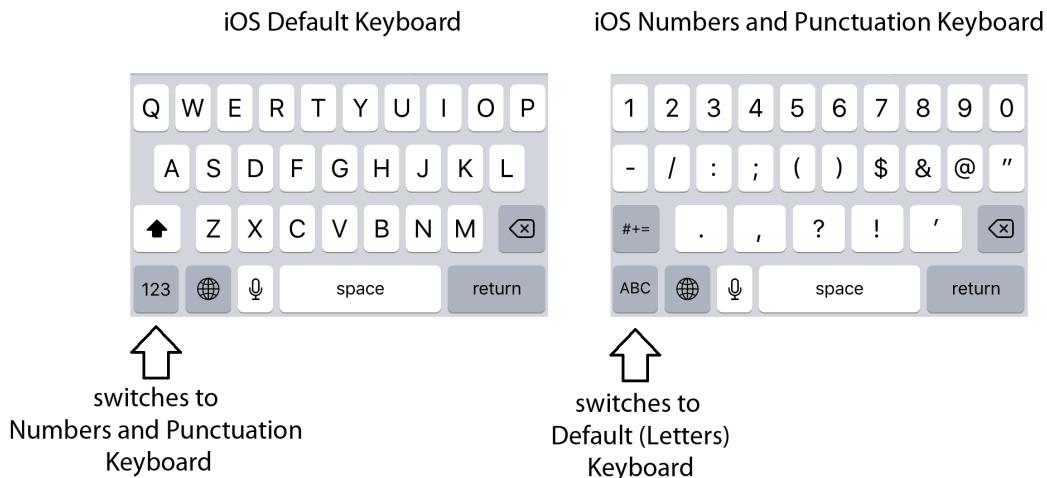


Figure 13.15: The iOS English default keyboard and numbers and punctuation keyboard

The **ASCII Capable** option displays the device’s keyboard with standard ASCII keys. This option is available to restrict the keyboard to those of English and similar language keyboards. This keyboard is also a letters keyboard, and the option to switch to the numbers and punctuation keyboard is available. For example, the iOS ASCII keyboard is shown in the preceding diagram, as it is the same as the default English keyboard.

The **Numbers And Punctuation** option opens the device’s numbers and punctuation keyboard with the option to switch to the “letters” keyboard. For example, the iOS numbers and punctuation keyboard is shown in the preceding diagram.

The **URL Keyboard** option brings up the device’s URL keyboard. This keyboard has a period (.) key, forward-slash (/) key, and .com key in place of the **Space** Key. For example, the following image shows the iOS URL keyboard and its numbers/punctuation form. Note that the URL keyboard’s numbers/punctuation form is not the same as the numbers and punctuation form that accompanies the default/ASCII keyboard:



Figure 13.16: The iOS URL keyboard and its numbers/punctuation form

The **Number Pad** option displays the device's keyboard with numbers (0-9) and (usually) a **Backspace** key. This keyboard is used for PINs, so it does not allow alternate characters. For example, the following image shows the iOS number pad keyboard:



Figure 13.17: The number pad keyboard

The **Phone Pad** option displays the device's keyboard with the same keys as the number pad keyboard but also includes keys for the asterisk and hash sign (pound sign). For example, the following image shows the iOS phone pad keyboard and its symbol display:



Figure 13.18: The iOS phone pad keyboard and its symbol display

The **Name Phone Pad** option displays the device’s “letters” keyboard and can switch to the phone pad keyboard. For example, the following image shows the iOS name phone pad keyboard’s two views:



Figure 13.19: The iOS name phone pad keyboard’s two views

The **Email Address** option shows the device’s email keyboard. The email keyboard prominently displays the @ key and the period (.) key as well as other common email address symbols. For example, the following image shows the iOS email keyboard and its numbers/punctuation form:



Figure 13.20: The iOS email keyboard and its numbers/punctuation form

The **Social Keyboard** option displays the device’s social keyboard. This keyboard prominently displays common social networking keys such as the @ key and the # key. For example, the following image shows the iOS “Twitter” keyboard and its numbers/punctuation form. On the iOS device, this keyboard is specifically called the *Twitter keyboard*, but it displays on other social networking apps such as Instagram:

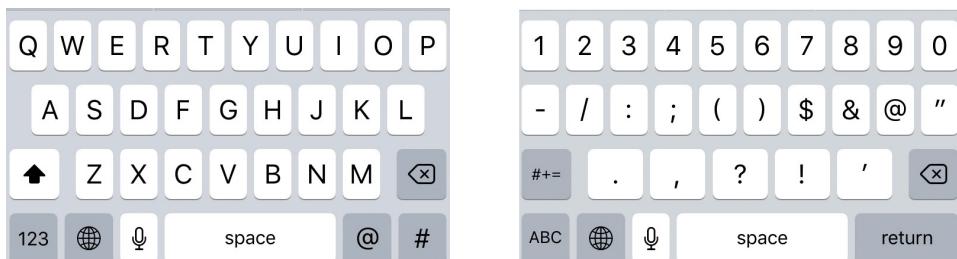


Figure 13.21: The iOS Twitter keyboard

The **Search** option displays the web search keyboard. This keyboard prominently displays the space and period keys. For example, the following image shows the iOS web search keyboard and its numbers/punctuation form:



Figure 13.22: The iOS web search keyboard and its numbers/punctuation form

Note

You can view a list of all the keyboard types available on iOS at <https://developer.apple.com/documentation/uikit/uikeyboardtype>.

You can view a list of all input types (not just keyboard, but they are included in the list) available on Android at https://developer.android.com/reference/android/widget/TextView.html#attr_android:inputType.

Character Validation options

When the **Custom Content Type** is selected, you have the option to select which type of **Character Validation** you would like to use. This option restricts the type of characters that can be entered in the Input Field. If the player attempts to enter a character that does not meet the restrictions, no character will be inserted in the Input Field.

The possible options are **None**, **Integer**, **Decimal**, **URL**, **Alphanumeric**, **Name**, and **Email Address**.

Character Validation only checks each individual character being entered to see whether it is allowed within the field. It does not check the entire string to see whether the string itself is valid. For example, if **Email Address** is selected, it will not check whether it is actually in the format of an email address. That type of validation will have to be accomplished via code.

The **None** option does not perform any character validations, allowing any character to be entered into the Input Field with any formatting.

The **Integer** option allows any positive or negative integer value to be entered. This restricts the input to only allowing the digits 0 through 9 and the dash (negative symbol). The input is further restricted to allowing the negative symbol only as the first character entered.

The **Decimal** option has the same restriction as the **Integer** option, but it also allows a single decimal point to be entered.

The **Alphanumeric** option only allows English letters (a through z) and the digits 0 through 9. Capital and lowercase letters are permitted; the negative symbol and decimal point are not accepted.

The **Name** option allows characters typically found in names and provides formatting. It allows letters, spaces, and an apostrophe ('). It also enforces capitalization of the first character in the string and every character that comes after a space. A space cannot follow an apostrophe, and a space cannot follow another space. Only one apostrophe is allowed in the string. The letters are not restricted to just a-z as with the **Alphanumeric** option. Any Unicode letter is permitted.

Note

For a list of all allowable Unicode letters, check out the remarks on the `Char.IsLetter` method in .NET at [https://msdn.microsoft.com/en-us/library/system.char.isletter\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.char.isletter(v=vs.110).aspx).

The **Email Address** option allows characters that are allowed within an email address and enforces a few formatting rules. It is significantly less restrictive in the types of characters that can be entered than the other validation options. The following characters are allowed:

- Lowercase and capital English letters (a through z)
- Digits 0-9
- The following punctuation marks and special symbols:

Symbol name	Character
at sign	@
dot/period	.
question mark	?
exclamation point	!
hyphen	-
underscore	_
apostrophe	'
backtick	`
tilde	~
open and close braces	{ and }
vertical bar	
caret	^
asterisk	*

Symbol name	Character
plus sign	+
equal sign	+
forward slash	/
hash sign/pound sign	#
dollar sign	\$
percent	%
ampersand	&

Table 13.1: Permitted special characters

Spaces are not allowed, only one @ symbol is allowed in the string, and a dot cannot follow another dot.

Even though a dot as the first character of an email address is not valid, the **Email Address Character Validation** option does not restrict it from being the first character entered in the Input Field.

Properties of the caret and selection

A **Caret** (also known as a **text insertion cursor**) is a vertical bar used to represent where text will be inserted when typed. When the game is playing, a third child is automatically added to InputField named `Input.Caret`. When the Input Field is selected, the caret becomes visible.

The properties discussed in this section affect the look of the caret as well as the look of text if it is selected (or highlighted) using the caret.

The **Caret Blink Rate** property determines how quickly the caret will blink. The number assigned to this property represents how many times the caret will blink per second. The default value is 0 . 85.

The **Caret Width** property determines how thick the caret is in pixels. The default value is 1.

When the **Custom Caret Color** property is selected, a secondary property, **Caret Color**, becomes available. You then have the option to change the color of the caret. Unless **Custom Caret Color** is selected and the **Caret Color** is changed, the caret will be a dark grey color.

When the caret is dragged across characters within the Input Field, the characters will be selected (or highlighted). The **Selection Color** property determines the color of the selected text.

Input field default events – On Value Changed (String) and On End Edit (String)

The Input Field component has two default events. The first default event is the **On Value Changed** Event, as seen in the **On Value Changed (String)** section of the Input Field component. This event will trigger whenever the text within the Input Field is changed. It accepts a string as an argument, and

its use of the argument works in the same way as the **On Value Changed** events from UI components discussed earlier in this chapter. If you want to pass a parameter to the function, you can select the function from either the **Static Parameters** list or from the **Dynamic string** list, depending on how or if you want an argument passed to the function:

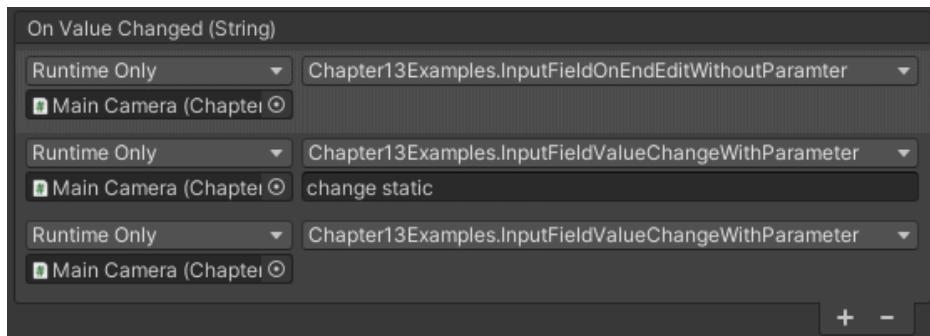


Figure 13.23: On Value Changed Events on Input Field Example in the Chapter13 scene

If you want to constantly check what the player is entering in the Input Field, you would use the third setup shown in the preceding image, which selects a function with a parameter from the **Dynamic string** list.

The second default event is the On End Edit event, as seen in the **On End Edit (String)** section of the Input Field component. This event fires whenever the player completes editing the text. This completion is confirmed by the player either clicking outside of the Input Field (so that the Input is no longer selected) or by submitting the text.

It accepts a string as an argument. As with the other events discussed in this chapter, you can choose to pass no argument, a static argument, or a dynamic argument. The following screenshot shows the setup for all three options:

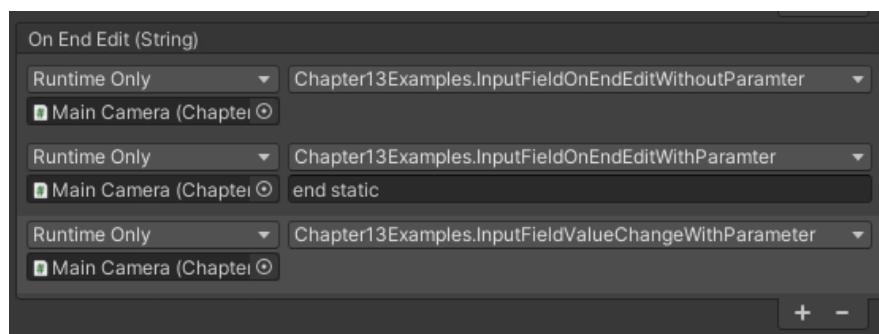


Figure 13.24: On End Edit Events on Input Field Example in the Chapter13 Scene

If you want to have the On End Edit event called when the *Enter* key is hit, use either the **Single Line** or **Multi Line Submit** options for the **Line Type**.

Now that we've reviewed the UI Input Field, let's review its counterpart, the Input Field – TextMeshPro.

Input Field - TextMeshPro

The Input Field - TextMeshPro is very similar to the UI Input Field. When added to the scene, you'll see it looks nearly identical, except that the placeholder text has a different font. The UI Input Field uses an Arial font by default, while the Input Field - TextMeshPro uses Liberation Sans.

To create a UI Input Field, select + | UI | **Input Field - TextMeshPro**. By default, Input Field - TextMeshPro GameObject has a child named Text Area, which has two children: a Placeholder and a Text object. You will observe that it is slightly different in setup than the UI Input Field.

The Text Area GameObject contains a Rect Transform component and a Rect Mask 2D component. The Text Area ensures that the text does not appear outside of a specified area, as shown by the highlighted area in the following image. If you wanted to change the size of this area, you would change the properties on the Rect Transform component:



Figure 13.25: The Text Area of the InputField - TextMeshPro

The Placeholder and Text children are simply Text - TextMeshPro objects. You can find more information about the Text - TextMeshPro objects in *Chapter 10*.

An Input Field - TextMeshPro GameObject contains an Image component. If you want to change the appearance of the input box, change the **Source Image** of the **Image** component on the InputField (TMP) parent.

TextMeshPro - Input Field component

The parent InputField (TMP) object has a **TextMeshPro – Input Field** component. It has all the properties common to the interactable UI objects, many of the same properties of the standard UI Input Field, and a few that are exclusive to Input Field - TextMeshPros. This section will not discuss the properties that Input Field - TextMeshPros share with UI Input Fields since they were discussed in the previous section, and we will only discuss those that are exclusive to it. You can see the properties in the following image:

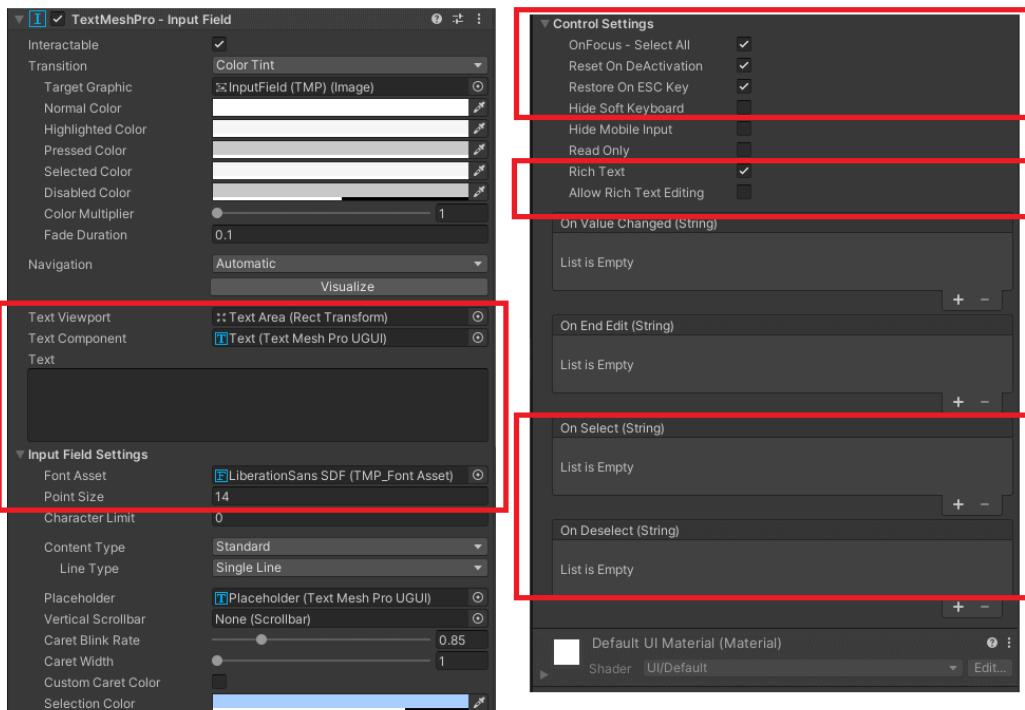


Figure 13.26: The TextMeshPro – Input Field component

The **Text Viewport** property is set to the Rect Transform of the area in which the entered text should be visible. The Rect Transform of Text Area child is assigned to this property, by default. As stated earlier, the Text Area child has a Rect Mask 2D component that stops text from becoming visible outside of the area defined by the Rect Transform component of the Text Area.

The **Text Component** property is set to the Text Mesh Pro UGUI component of the object in which the entered text should display. The TextMeshPro - Text object assigned to this property will determine the font and display settings of the entered text. The Text Mesh Pro UGUI component of the Text child is assigned to this property, by default.

The **Text Input Box** group can be expanded to display a large text input area. The **Text Input Box** property works the same way as the **Text** property on the Input Field component of UI Input Field objects. The text entered by the user will be stored here and can be accessed by code. This will store the actual text entered and not the formatted text. For example, if the text has been formatted to appear as asterisks (as with **Pin** and **Password Content Types**), the actual pin or password will be stored here rather than a string of asterisks.

Input Field settings

The **Font Asset** property determines the font of the various texts displayed within the Input Field - TextMeshPro, and the **Point Size** property determines the size of the text. You'll note that the Placeholder and Text children also have the **Font Asset** and **Point Size** properties on their Text Mesh Pro UGUI components. Changing the **Font Asset** and **Point Size** properties on the Input Field - TextMeshPro parent will also change the corresponding properties on the child objects.

The rest of the properties in this group are the ones included within the UI Input Field.

Control settings

If the **OnFocus - Select All** property is selected, when the Input Field - TextMeshPro is selected, all the text within the field will be highlighted.

If the **Reset On DeActivation** property is selected, the caret will reset to the default position at the front of the text.

If the **Restore on ESC Key** property is selected, the text will reset back to the default when the *esc* key is hit. The default will be either an empty string or whatever is entered in the **Text Input Box** when the scene starts.

The **Rich Text** property means that any rich text tags to be accepted, and the **Allow Rich Text Editing** property allows the user to enter rich text tags within the field.

Input Field - TextMeshPro default events – On Select (String) and On Deselect (String)

The Input Field - TextMeshPro has four default events: the **On Value Changed** Event, the **On End Edit** Event, the **On Select** Event, and the **On Deselect** Event, as shown in the **On Value Changed (String)**, **On End Edit (String)**, **On Select (String)**, and **On Deselect (String)** sections.

The first two events, the **On Value Changed** Event and the **On End Edit** Event are the same as those presented in the UI Input Field.

The third event is the **On Select** Event. This event fires whenever the Input Field - TextMeshPro is selected. The fourth event is the **On Deselect** Event. As you would expect, the event fires whenever the Input Field - TextMeshPro is deselected. It works similarly to the **On End Edit** event, except that it does not fire when the text is submitted.

As with the other events discussed in this chapter, you can choose to pass no argument, a static argument, or a dynamic argument to the **On Select** and **On Deselect** events.

Now that we've reviewed the various interactable components of the uGUI, let's look at some examples of how to use them.

Examples

This chapter has so many new items in it that I could spend the rest of this book just showing you examples! Sadly, I can't do that, so I will show you examples that I hope will be the most useful. Let's begin.

Creating a dropdown menu with images

Let's continue working on our scene and create a dropdown menu that will allow us to swap our player character between a cat and a dog. The final version will appear as follows:



Figure 13.27: The final version of the Paused Menu dropdown

Changing our selection will then change the image of the character that appears at the top of the screen.

The spritesheet containing the dog image is an asset that I've modified from free art assets found here:

<https://opengameart.org/content/cat-dog-free-sprites>

This is the same asset that provided us with the cat sprites.

When you want to see your UI Dropdown menu in play mode, you have to press *P* to bring up the *Pause Panel*. This can be kind of annoying when you just want to quickly check the layout. You can disable the automatic hiding of the *Pause Panel* momentarily by disabling the *ShowHidePanels.cs* script on the *Main Camera*. Just remember to turn it back on when you are done!

Laying out the dropdown with caption and item images

To create a UI Dropdown menu like the one shown in the previous image, complete the following steps:

1. Locate the *dogSprites.png* image provided in the source files of the text and bring it in to the *Assets/Sprites* folder of your project.

2. Slice the spritesheet by setting its **Sprite Mode** to **Multiple** and utilizing **Automatic** slicing. When you perform the **Automatic** slice, set the **Pivot** to **Bottom**.
3. Now, let's add a UI Dropdown to our Pause Panel. Right-click on the Pause Panel in the **Hierarchy** and select **UI | Dropdown**. Move the Dropdown upward in the **Hierarchy** so that it is listed right below Pause Banner.
4. Adjust the size and position of the **Dropdown** by setting its Rect Transform properties, as follows:

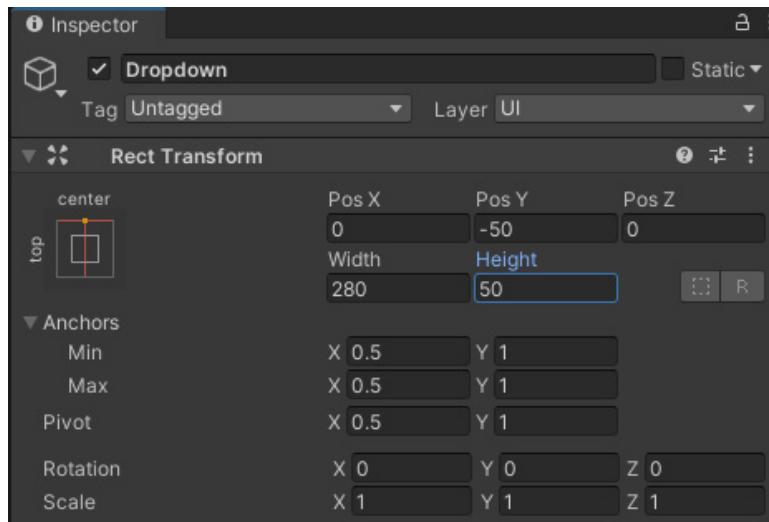


Figure 13.28: The Rect Transform of the Dropdown

Your Dropdown should now appear as follows:

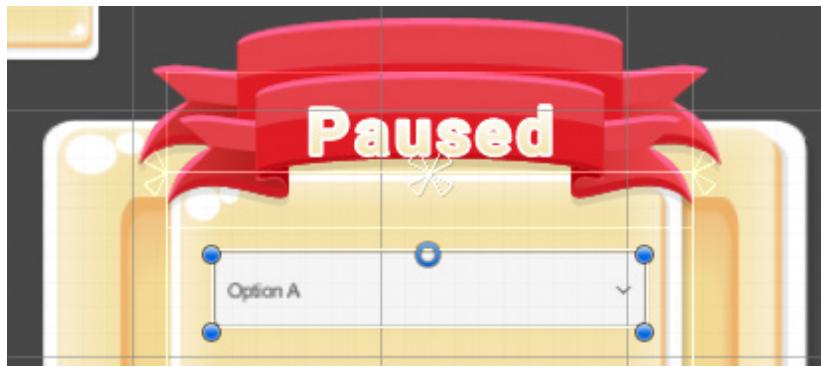


Figure 13.29: The current state of the Dropdown

5. To change the Dropdown background, we need to change the **Source Image** on the Image component of the Dropdown. Assign the uiElements_12 subsprite into the **Source Image**.

Expanding the Dropdown in the **Hierarchy** to view its children will reveal a child named Arrow. You can adjust the properties of Arrow to change its look and general position.

Give the Arrow the uiElements_132 sprite and adjust the Rect Transform, as illustrated:

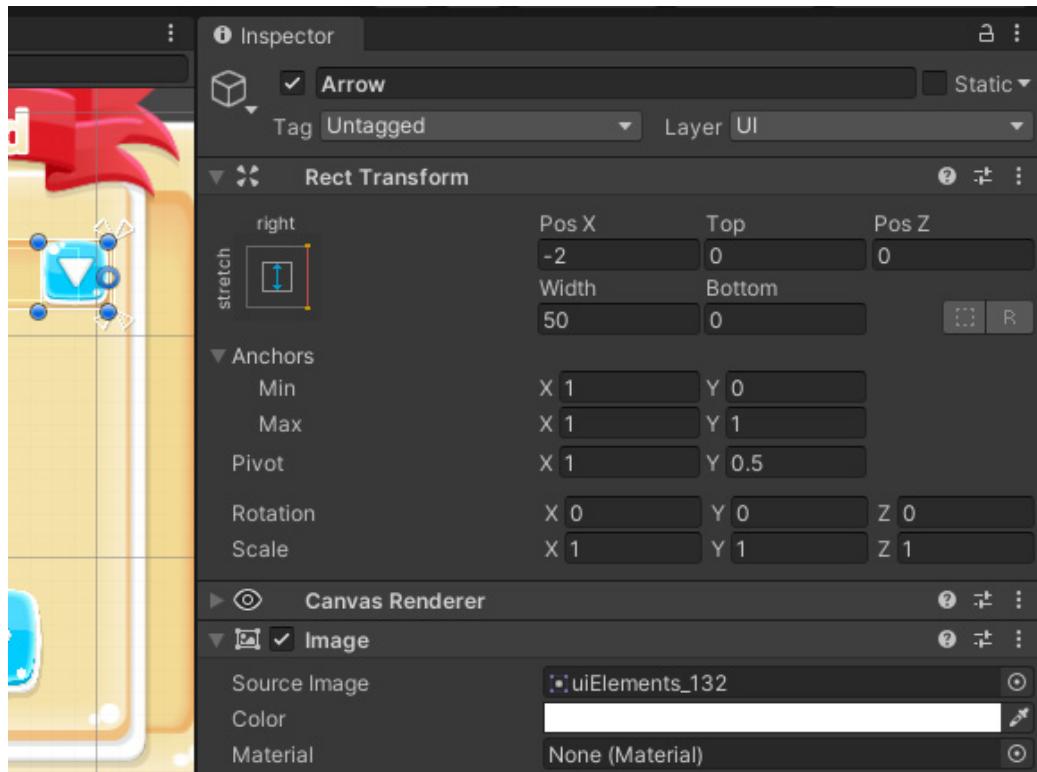


Figure 13.30: The Rect Transform of the Arrow

Note

Typing 132 in the search bar of the Project view is a quick way to find the uiElements_132 image.

6. While the Dropdown component has a variable for a caption, the UI Dropdown template does not come prebuilt with one. So, we have to manually add one in ourselves. Right-click on Dropdown in the **Hierarchy** and select **UI | Image** to give it a child Image. Move its position in the **Hierarchy** so that it is above Label1. Rename it **Caption**.
7. Give Caption the **catSprites_0** sprite and adjust its Rect Transform, as follows:

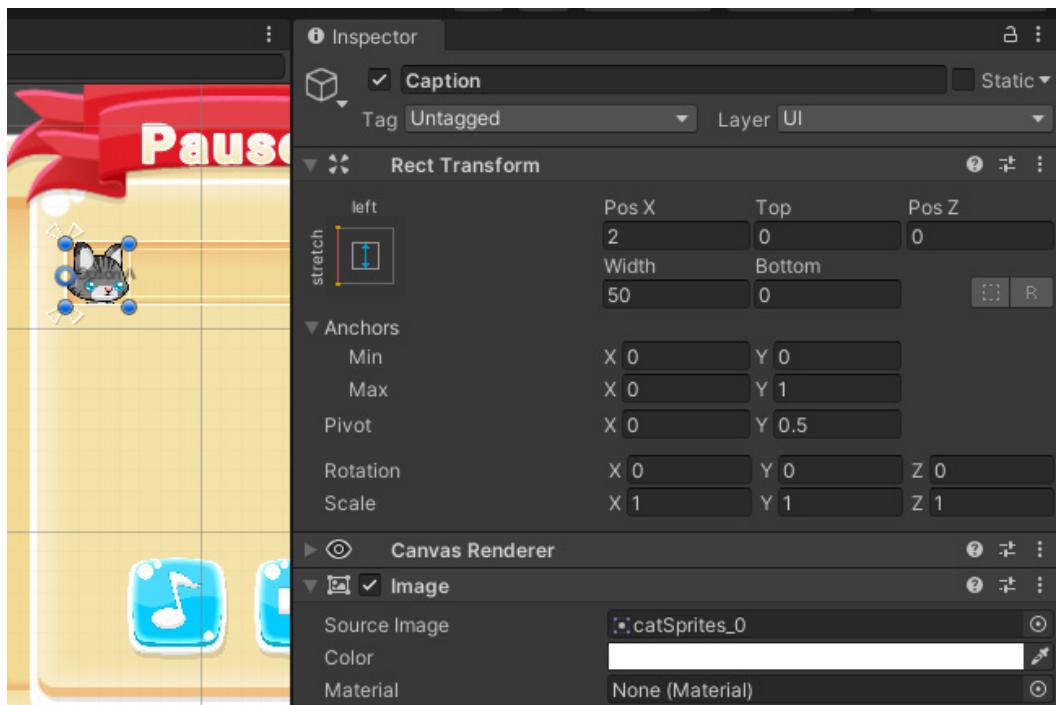


Figure 13.31: The Rect Transform of the Caption

We've set it to the image of the cat so that we can see whether it is displaying properly but remember that it will automatically change to the appropriate sprite based on the selection.

8. Now, select Label1 and adjust its **Rect Transform** and **Text** components as shown:

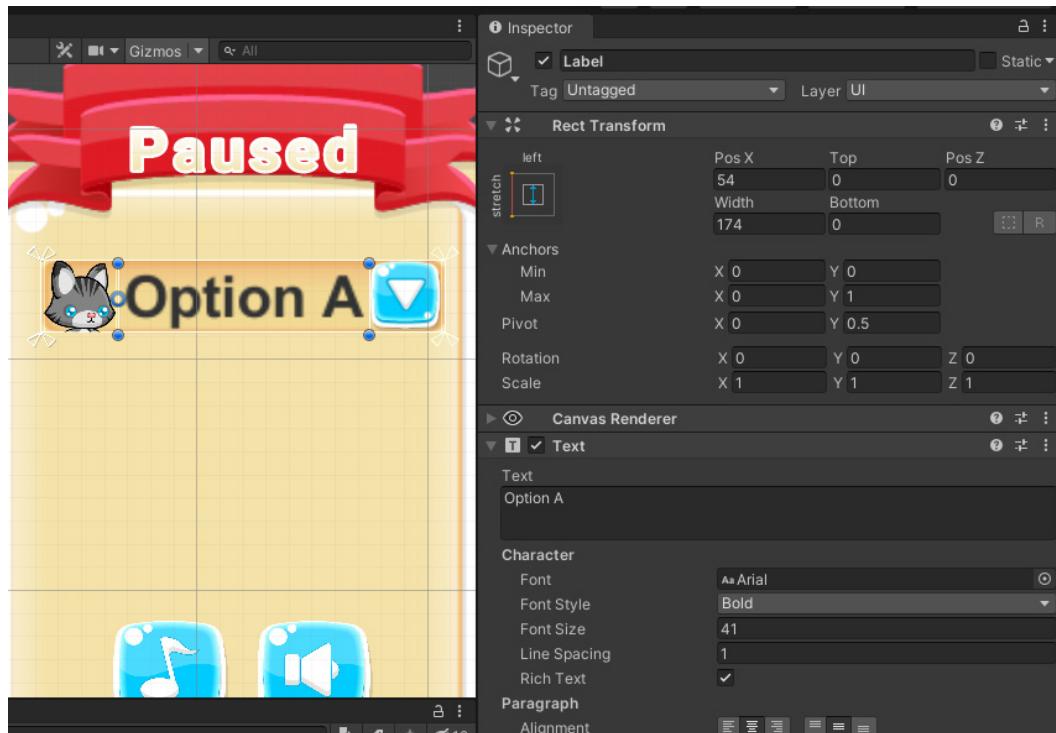


Figure 13.32: The Rect Transform of the Label

If you try to adjust the **Text**, it will revert to Option A, since this is driven by the **Dropdown** component.

9. Now that we have our caption set up the way we want it, let's work on the Template. Enable the **Template** object so that you can see it in the scene and expand it in the **Hierarchy** to view all of its children.

We won't use the **Scrollbar**, so we can leave it as it is. It will show up in the Scene view but will not be visible in Play mode, because its **Visibility** is set to **Auto Hide And Expand Viewport** in the **Scrollbar** component of **Template**.

- To change the background of the window that drops down, change the **Source Image** on **Image** component of **Template** to **uiElements_11**.
- The **Item** child shows the general format to all options that will be listed in our **Dropdown**. Any changes we make to it will be automatically applied to all options when the **Dropdown** script populates them.

Select Item and change its Rect Transform **Height** to 50.



Figure 13.33: The Rect Transform and Hierarchy of the Item

12. Content needs to fully encapsulate the Item. So, change its Rect Transform **Height** to 52. Ensure that **Pos Y** is at 0.

If you played the game, the **Pos Y** will change on Rect Transform of Content from 0 to something else. This is actually supposed to happen (option A is being set behind the caption), but it's annoying when you are trying to lay out your Item, because you won't be able to see your Item.

Therefore, after playing, change **Pos Y** back to 0 so that you can continue editing.

13. Just as we had to add a child Image to Dropdown so we could have a caption image, we also have to add a child Image to Item, so we can have an image display in the menu.

Right-click on Item in the **Hierarchy** and select **UI | Image** to give it a child Image. Rename it **Item Image** and move it between **Item Checkmark** and **Item Label** in the **Hierarchy**.

14. Give **Item Image** the **catSprites_0** sprite and adjust its Rect Transform, as shown:

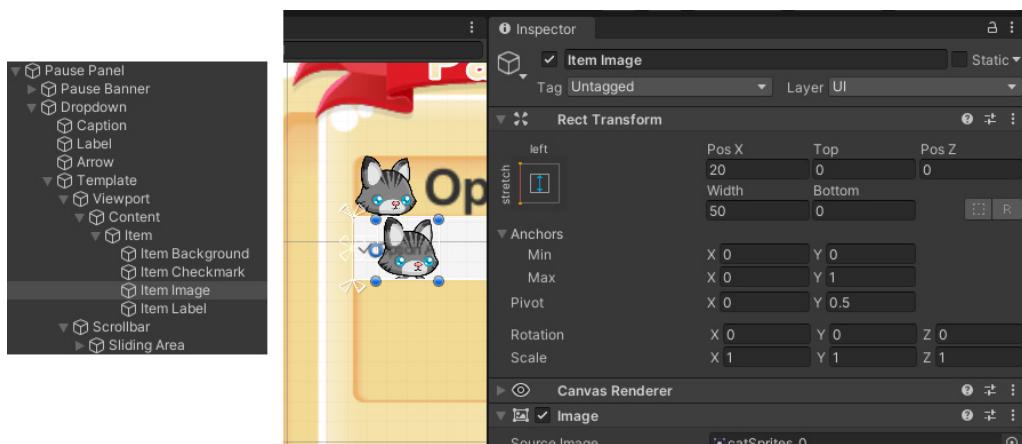


Figure 13.34: The Rect Transform and Hierarchy of the Item Image

15. Select Item Label and adjust its **Rect Transform** and **Text** components, as follows:

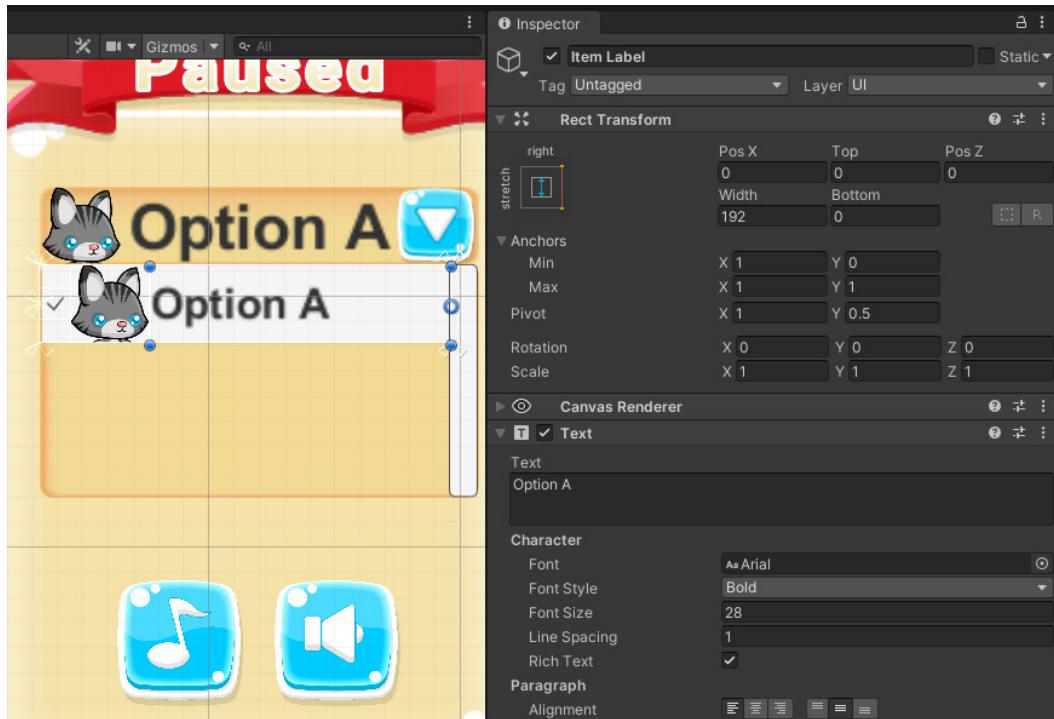


Figure 13.35: The Rect Transform of the Item Label

16. The last thing to do to Item is to remove the white background. Select Item Background and change the **alpha** on the **Color** property of the Image component to 0. Your Dropdown menu should now look as follows:



Figure 13.36: The current state of the dropdown

17. Now that we have set up our Dropdown visually, we need to set up the properties of the Dropdown component. If you play the game, you will see that our Dropdown doesn't have the correct options yet. You can't tell from playing it, but **Caption Image** and **Item Image** also aren't hooked up:

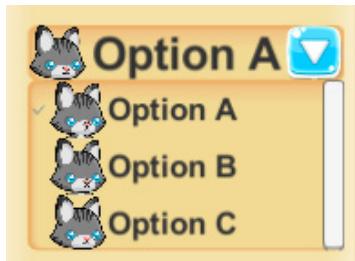


Figure 13.37: The current state of the dropdown when pressing Play

18. Let's update the Dropdown component on Dropdown. Drag the **Caption** child from the Hierarchy into the **Caption Image** property and **Item** **Image** into the **Item Image** property. When you drag **Caption** into **Caption Image**, the image of the cat will disappear from the scene. Don't worry! It will come back. It's updating to the image of **Option A**, which is set to nothing right now.
19. The last thing we need to set up is the set of options that will display in the menu that drops down. We only need two options, so select **Option C** and then hit the minus (-) button to delete it. Now, change the text **Option A** to **Cat** and the text **Option B** to **Dog**.

Drag **catSprites_0** into the sprite slot under **Cat** and **dogSprites_0** into the sprite slot under **Dog**. Your Dropdown component properties should appear, as follows:

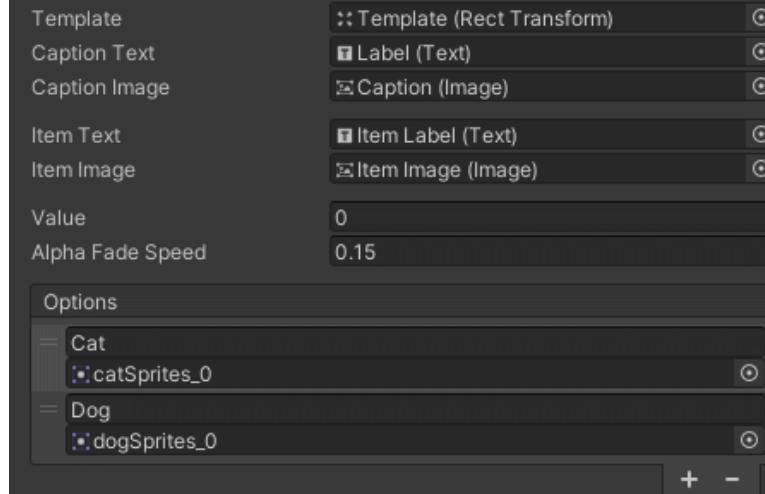


Figure 13.38: The properties of the Dropdown component

If you play the game, you will see that the dropdown now shows the appropriate list of options and the caption image and text update based on your selection:



Figure 13.39: The final visual set up of the dropdown options

Using the information from the dropdown selection

Now that our Dropdown looks the way we want it and is functioning properly, we can access the player's selection with code. We'll use the player's selection to update the player character image in the top-left corner of the screen.

To swap the player character image with the selection from the Dropdown, complete the following steps:

1. Create a new C# script in your Assets/Scripts folder named PlayerCharacterSwap.cs.
2. For us to access UI variable types, add the UnityEngine.UI namespace to the top of the script with the following line:

```
using UnityEngine.UI;
```

3. We only need two variables, one to represent the Image that will be swapped with the selection from the **Dropdown** menu and one to represent the **Dropdown** menu.

We'll attach this script to the Dropdown object, so we don't have to make the variable referencing it public. Add the following variable declarations after the namespace declarations:

```
public Image characterImage;
Dropdown dropDown;
```

4. Initialize the dropDown variable in an Awake() method with the Dropdown component attached to the object this script will be attached to:

```
void Awake () {
    dropDown = GetComponent<Dropdown> ();
}
```

5. The default event on the Dropdown component is the **On Value Changed** event, and it accepts an integer argument. Create a public function that accepts an integer parameter with the following:

```
public void DropDownSelection(int selectedIndex) {
```

6. The `DropDownSelection` method will get the integer value of the **Value** property from the `Dropdown` component. **Value** represents the index of the option currently selected within the **Options** list. If we pass the value of **Value** as an argument to the function, we can then reference it with the `selectionIndex` parameter within our script.

Add the following two lines to your `DropDownSelection` function:

```
Debug.Log("player selected " + dropDown.options[selectionIndex].text);

characterImage.sprite = dropDown.options[selectionIndex].image;
```

The first line will find the text on the option at the specified index in the options list and print it to the console.

The second line will find the sprite on the option at the specified index in the options list and change the sprite on the `characterImage` to that sprite.

7. We're now done with the script and can hook it up in the Unity Editor. Drag the `PlayerCharacterSwap.cs` script onto `Dropdown` to attach it.

Remember, the `dropDown` variable is not public, because we expected to attach this script as a component to the `Dropdown`.

8. The public variable `characterImage` needs to be assigned in the Inspector. Drag the `Character Image` from the **Hierarchy (HUD Canvas | Top Left Panel | Character Holder | Character)** to the `Character Image` slot on the `Player Character Swap` component.
9. Now we need to call the `DropDownSelection` function on the `PlayerCharacterSwap.cs` script from the **On Value Changed** event on the `Dropdown` component. Select the plus sign (+) in the **On Value Changed (Int32)** event list to add a new **On Value Changed** event. Drag `Dropdown` from the **Hierarchy** into the object slot and select the `DropDownSelection` function from the **Dynamic int** list of the `PlayerCharacterSwap` script. The **On Value Changed (Int32)** event list should appear as follows:

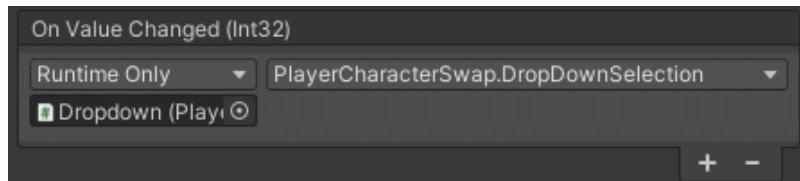


Figure 13.40: The On Value Changed (Int32) property

That's it! Now play the game and watch the player character's image swap with the image selected from the Dropdown:



Figure 13.41: The final version of the scene

Figure 13.41 shows the final version of the game.

Summary

Who knew there were so many different types of interactable UI objects? Having templates for these different UI objects is incredibly helpful. Technically, they can all be built *by hand* with Buttons, Images, and Text, but that would take a lot of effort you don't have to worry about because Unity has done it for you. In this chapter, we reviewed how to use the common UI elements: Toggle, Slider, Dropdown, and Input Field.

Next, we will cover using animations within the UI!

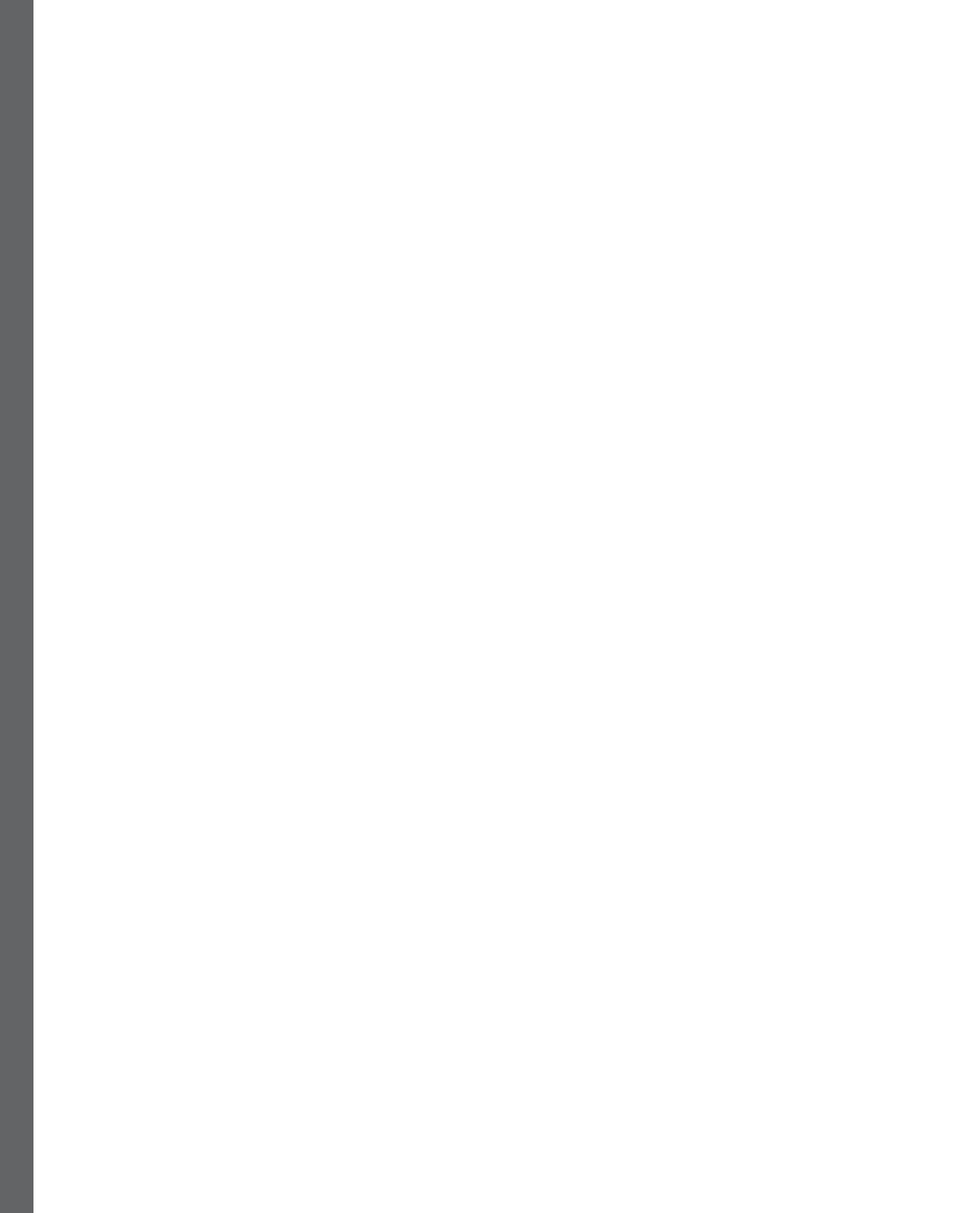
Part 4:

Unity UI Advanced Topics

In this part, you'll learn about more advanced topics related to the Unity UI system. You'll learn how to animate UI elements as well as display particles within the UI. You'll learn how to create UI that appears within the world of your game. Lastly, you'll get an overview of considerations to make sure you create an optimized UI.

This part has the following chapters:

- *Chapter 14, Animating UI Elements*
- *Chapter 15, Particles in the UI*
- *Chapter 16, Utilizing World Space UI*
- *Chapter 17, Optimizing Unity UI*



14

Animating UI Elements

Since we have already discussed how to create animation transitions for buttons, in this chapter, we'll take a look at animation transitions more thoroughly and discuss how to create animations for UI elements in a more general sense.

This chapter assumes that you have a basic understanding of Unity's Animation System and will not go into detail describing the names of the various menus and layout of the Animation Window and Animator Window. Animation Clips and Animators will be described briefly, with a focus on how they relate to UI and their implementation, which will be discussed in the examples at the end of the chapter.

In this chapter, we will discuss the following topics:

- Applying animations to the various UI elements
- Creating a pop-up window that fades in and out
- Creating a complex Animation System with a State Machine and Animation Events

Even though I am assuming that you have a basic understanding of Unity's Animation System, I do want to emphasize the difference between an Animation Clip and the Animator.

When creating animations for items in Unity, you start with **Animation Clips**. An Animation Clip should represent a single distinct action or motion. So, for example, if you had a menu that performed two separate actions, bouncing and zooming, you'd make each of those actions a separate Animation Clip. Although you can have multiple things happen in a single Animation Clip, it is very important not to put multiple actions in a single clip unless they are always going to happen at the same time.

Every GameObject can have multiple Animation Clips. The **Animator** determines how all of these Animations link together. So, a GameObject's Animator will have all of its Animation Clips within it.

I wanted to make this distinction because in the past, I have seen projects with epic Animation Clips containing multiple actions that should have been broken down into more simple motions.

Note

As with previous chapters, all of the examples shown in this section can be found within the Unity project provided in the code bundle. They can be found within the scene labeled Chapter14.

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2014>

Animation Clips

The great thing about the Unity Animation System is that you can animate nearly any property of the UI. To create an Animation Clip, simply open the Animation Window (**Window | Animation** or **Ctrl + 6**), and with the UI element you want to animate selected, select **Create**:

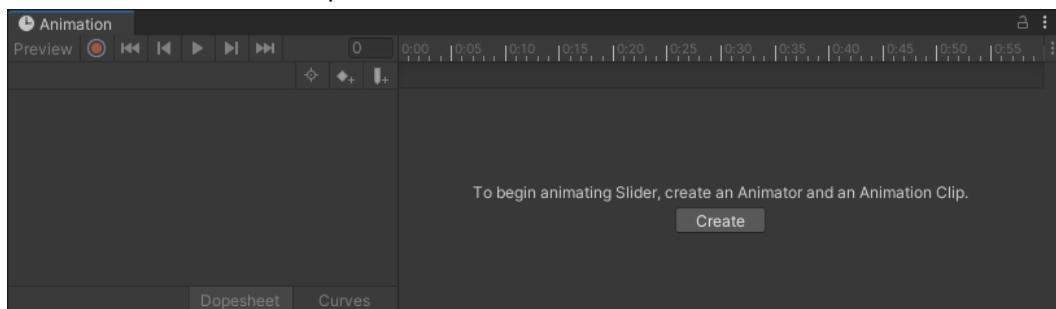


Figure 14.1: The Animation Window

Once you do so, you'll be prompted to save the Animation Clip.

After creating the Animation Clip, you can then add any property to the clip's timeline by clicking on **Add Property**:

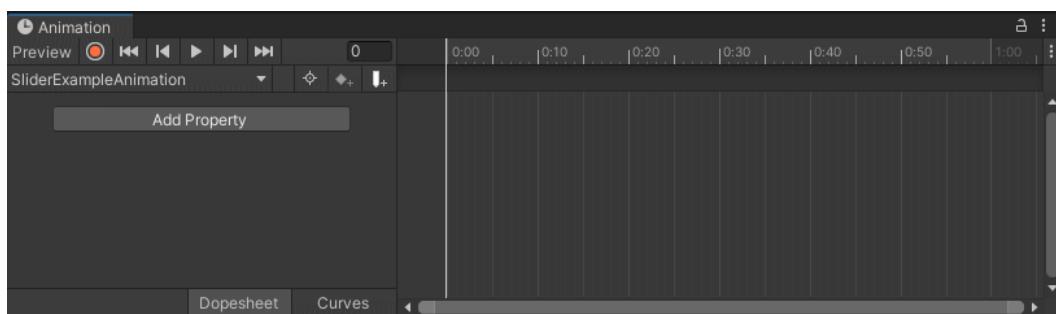


Figure 14.2: The SliderExampleAnimation timeline

Doing so will bring up every component of the object, as well as a list of all of its children:

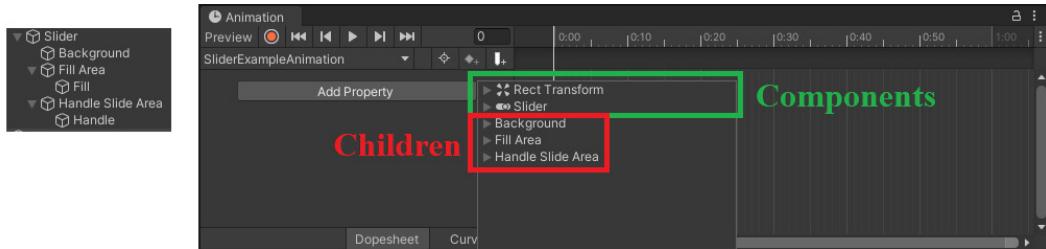


Figure 14.3: Adding an Animation property

You can also view the components and children of each child:

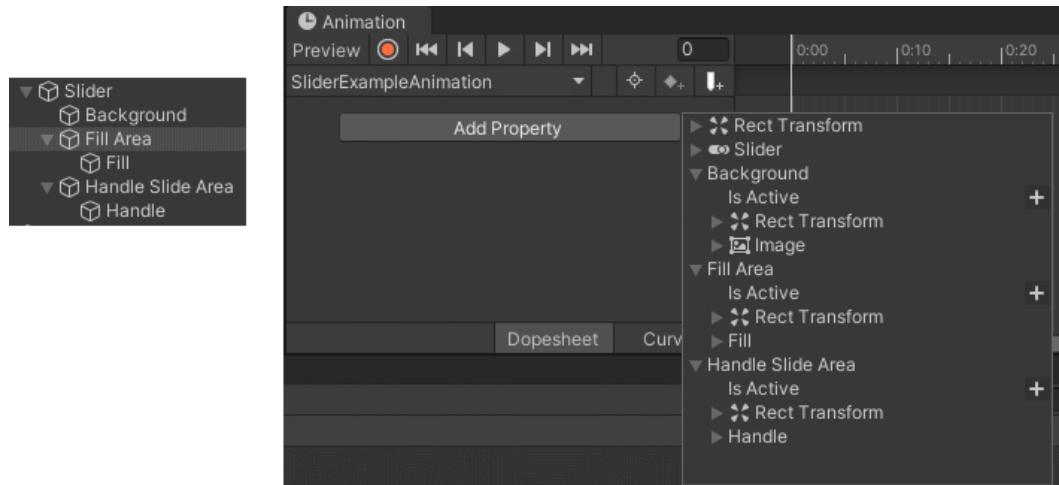


Figure 14.4: Expanding Animation components

Then, you can view the components and children of those children. You can continue in this manner until you have exhausted the list of GameObjects that are nested under the selected GameObject.

If you expand a component of a GameObject or one of its children, you will then note a list of all properties of that component that can be animated. As you will notice in the following screenshot, almost every property on the **Slider** component can be animated in this way:

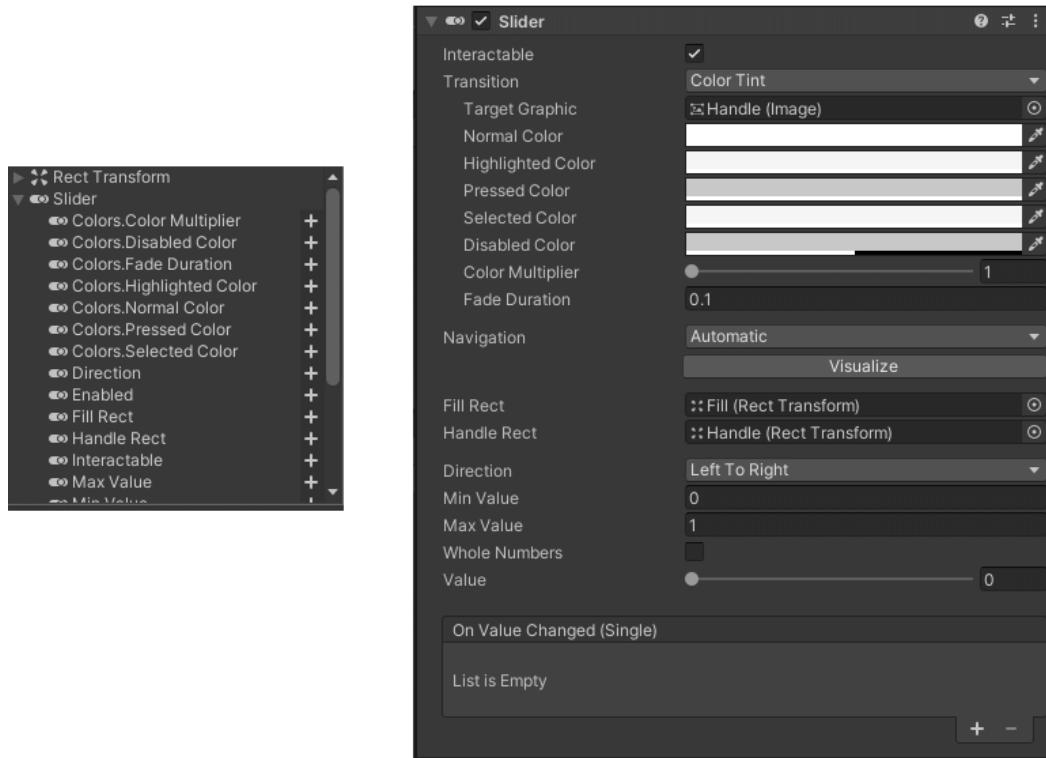


Figure 14.5: The Slider component and its Animatable properties

Only properties that have the following data types can be animated with the Animation System:

- `Vector2`, `Vector3`, and `Vector4`
- `Quaternion`
- `bool`
- `float`
- `Color`

Selecting the plus sign will add the property to the Animation timeline along with two keyframes. You can then change the values of each property at the various keyframes.

A **keyframe** is an important or *key* (hence the name) frame within an animation. It represents the start or end point of a transition within an animation.

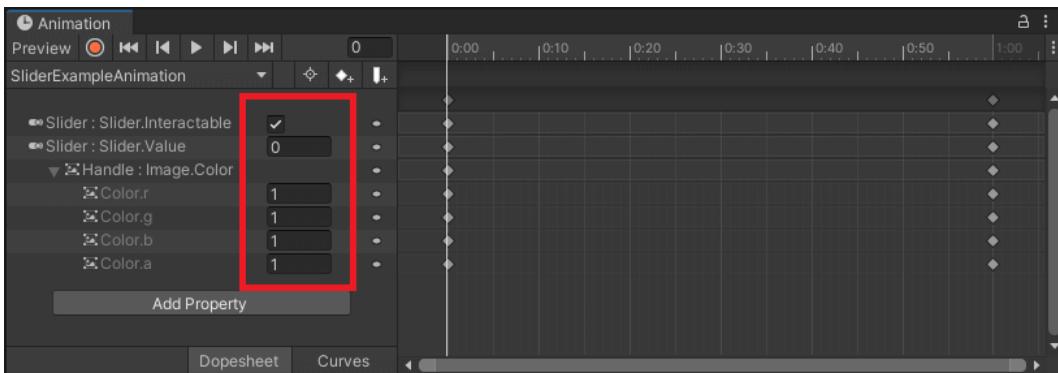


Figure 14.6: SliderExampleAnimation first keyframe

In the preceding screenshot, the values encased in the red boxes represent the value of the property at the particular frame. Boolean values are represented by checkboxes, and float values are represented by numbers. Each type can be edited directly by selecting them.

Unity will fill in the values between the keyframes so that they change (or interpolate) on a curve. You can view the interpolation curve by selecting the **Curves** tab.

You can watch the changes occur throughout all frames by playing the animation or scrubbing the playhead.

The **playhead** is the marker that indicates what frame is currently being displayed. “Scrubbing the playhead” means dragging the playhead across the timeline to view changes over individual frames.

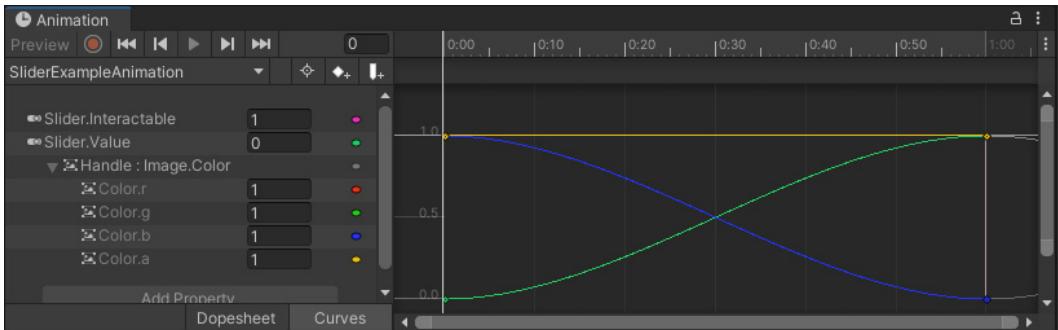


Figure 14.7: The Curves version of the timeline

Boolean properties are interpolated on a linear curve. All other properties (since they are a combination of floats) are interpolated along an ease-in-out curve. You can adjust these by adjusting the handles of the tangents at the keyframes by right-clicking on the keyframe.

Often, these ease-in-out curves will cause your UI to appear *bouncy*, and adjusting the interpolation curve can remove that bounce. For example, animating an object between two points may make the object go past the destination point momentarily due to the ease-in-out nature of the default interpolation curve.

Animation Events

One of my favorite things about Animation Clips is the ability to add Animation Events to frames on the timeline. Animation Events allow you to call functions that exist on the GameObject the Animation Clip is attached to. They are represented by white *flags* above the timeline, and hovering over them will show the name of the function called by the Animation Event.

An Animation Event can only call a function that exists somewhere on the GameObject the Animation Clip is attached to. The function can be public or private and can also have a parameter. The parameter can be of the following types:

- float
- int
- string
- An object reference
- An `AnimationEvent` object

You can add an Animation Event to the Animation Clip's timeline by right-clicking on the area above the frame in which you wish to place the Animation Event and clicking on **Add Animation Event**, or you can click the **Add Event** button. Selecting the **Add Event** button will add the Animation Event wherever the playhead currently rests.

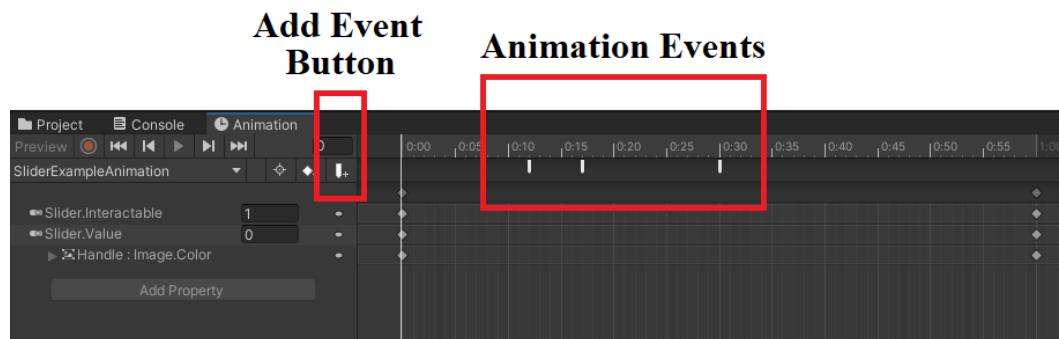


Figure 14.8: Animation Events

You can delete the Animation Event by selecting the white flag and clicking on **Delete**, and you can move it by clicking on it and dragging it.

The appearance of the Animation Event's Inspector depends on whether the Animation Clip is attached to a GameObject and whether the GameObject is currently selected. The two appearances are shown in the following screenshot:

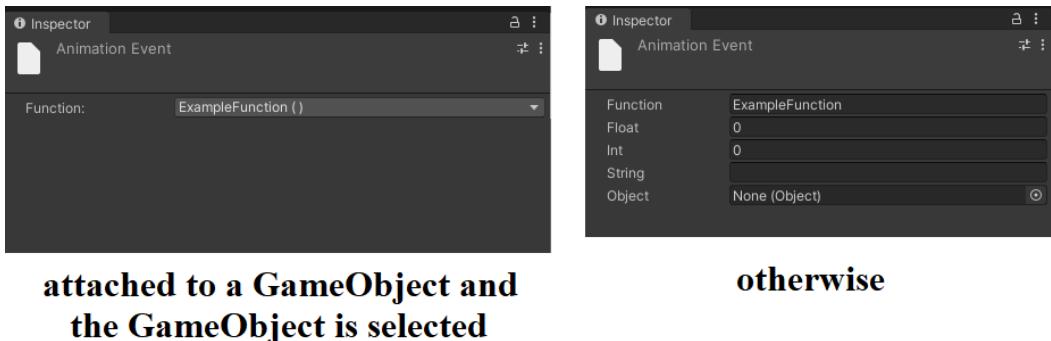


Figure 14.9: The Inspector of the Animation Event

If the Animation Clip is attached to a GameObject and the GameObject is selected, a dropdown menu will appear with a list of all available functions. If the selected function has a parameter, then options concerning the parameter will be made available (take a look at the preceding screenshot). Otherwise, the name of the function and the parameters to pass will have to be entered manually.

If you play the Chapter14 scene with the `Slider Animation Example` GameObject enabled, you will see an animation with events playing out.

Now that we've looked at Animation Clips, let's look at Animation Controllers.

Animator Controller

Whenever an Animation Clip is created for an object, an **Animator** is automatically created for it (if one does not already exist). An Animator component is also automatically added to the object. When I created the `SliderExampleAnimation` Animation Clip on the `Slider` GameObject in the preceding section, an Animator named `Slider` was created and the Animator component was attached to the `Slider` GameObject:

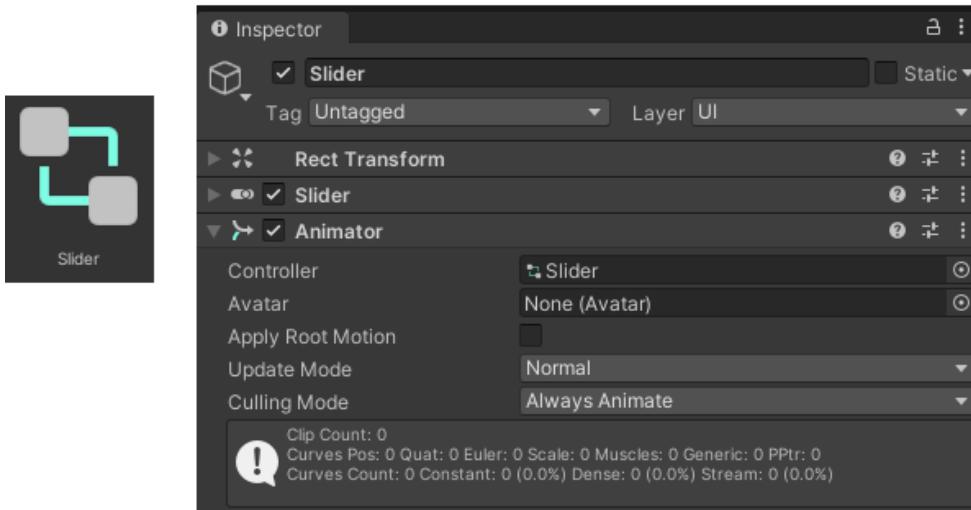


Figure 14.10: The Slider Animator's Inspector

An Animator is needed to play Animation Clips because it determines when Animation Clips are played.

An Animator is a type of decision tree known as a *state machine*. It holds a collection of *states*. States are essentially *statuses at a moment in time*. The *current state* of a state machine would be a representation of what is happening at this moment. So, for example, if there were a state machine describing my actions and behaviors, my *current state* would be *typing on the keyboard*. My state machine would have other states that I could eventually transition to, such as *sleeping* or *crying about approaching deadlines* if certain conditions are met.

States in the Animator are represented by rectangles called *nodes*. States are connected by transitions that are represented by arrow lines. These transitions occur after either a predetermined time or a set of conditions have been met. The current state will have a blue, animated status bar on it telling you the percentage of the state that has been completed. If the current state is waiting for a transition to occur, this status bar may loop or stop in the full position until the conditions of the transition are met:

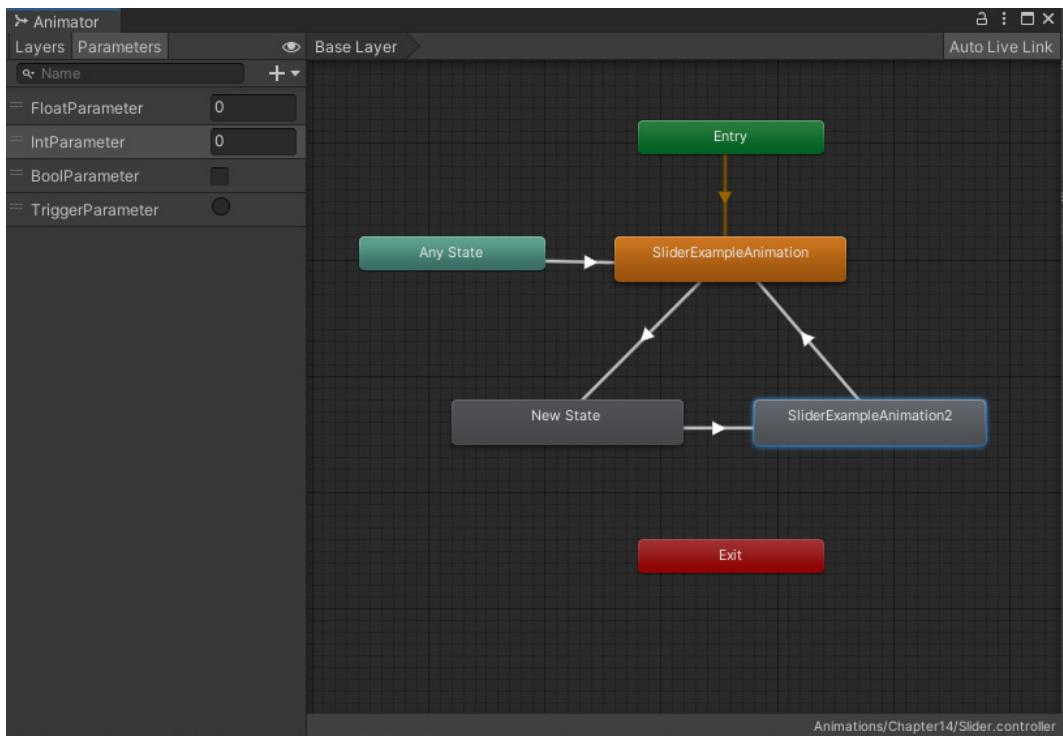


Figure 14.11: The Slider Animator

The bottom-right corner of the Animator Window displays the name of the current Animator controller as well as its folder location. This can be very helpful when you have many different Animators, as it tells you which Animator you are working with.

States can be empty or represent Animation Clips. If a state represents an Animation Clip, it will have an Animation Clip set to its **Motion** property in its **Inspector**, as in the following screenshot:

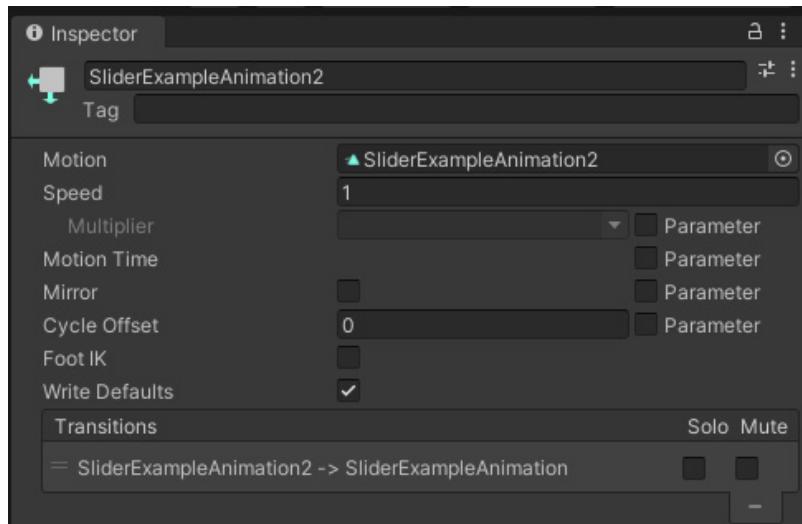


Figure 14.12: The Motion property of an Animation State

Most states will be colored gray, but those colored otherwise will represent special states. Every Animator will have an **Entry** node (green), an **Exit** node (red), and an **Any State** node (blue) within it. The first state you add to an Animator will be assigned the Default Layer State node (orange). You can change the state of the Default Layer State at any time. Note that Animator Layers are discussed in a later section.

The **Entry** and **Exit** nodes essentially work as *gates* between state machines. You can have state machines within state machines, and these *gates* decide what happens after the state machine is entered and exited, respectively. So, the **Entry** node represents the instance the state machine starts, and the **Exit** node represents the instance it stops.

The **Entry** node always transitions to the Default Layer State, and you cannot define the conditions of the transition, so the transition will always happen automatically and instantly. Therefore, you can think of the Default Layer State as the *first* state that will occur when the state machine begins.

The **Any State** node is an *all-encompassing* state. You use this state when you want a transition to happen, regardless of the current state. You can only transition away from the **Any State** node. Continuing with the example of a state machine that describes my behavior, I would have a transition from **Any State** to the state of *crying about approaching deadlines*, because no matter what I am currently doing, I could burst into tears if the condition “deadline is within 24 hours” is met.

As stated before, the Animator will stay in the current state until a specified amount of time has passed or a set of conditions have been met. Selecting a transition arrow will display the conditions of transition:

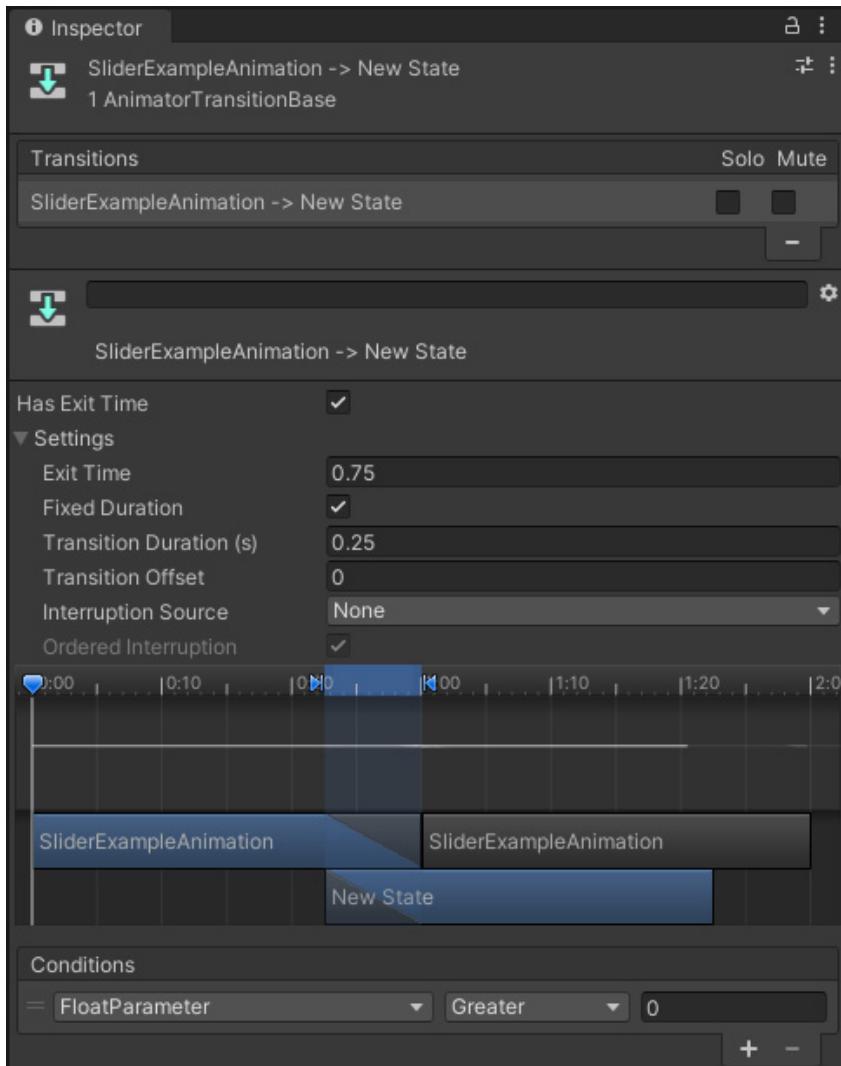


Figure 14.13: Expanded properties of an Animation

The transition from the preceding screenshot requires both a specified amount of time and a condition. The transition also isn't instantaneous and takes 0.25 s to complete.

The **Conditions** that must be met for a transition to occur are set by the Animator's **Parameters**, which can be found and created in the top-left corner of the Animator Window:

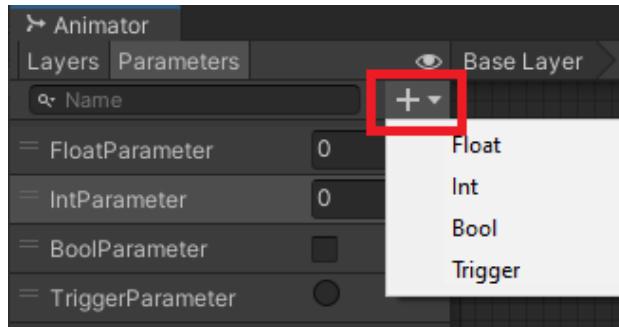


Figure 14.14: Adding an Animator Parameter

The values of these parameters can be set from scripts. There are four types of Parameters: **Float**, **Int**, **Bool**, and **Trigger**. The first three are named for their value type, but a **Trigger** is a little less obvious. A Trigger is a Bool Parameter that instantly resets itself to `False` after it is used by a transition. Trigger Parameters are helpful for creating flood-gate-type actions where the animation has to stop and wait before it can proceed to the next state. These are preferred over Bool Parameters in instances where the states and transitions form a loop, because a Bool Parameter would have to be manually reset before the state looped back around.

When you look at the list of **Parameters**, you can tell which type they are by the value on the right. Float Parameters have decimal numbers, Int Parameters have integers, Bool Parameters have square checkboxes, and Trigger Parameters have circular radio buttons.

Since Animators are state machines, they can be used to accomplish much more than animations. Animators can be used to keep track of complex game logic. For example, I created the following state machine for a match-three RPG to keep track of what was currently happening in the game. Using it to keep track of the current state of the game allowed me to restrict what the player could do based on what was happening in the game.

For example, if the enemy character was attacking, the player would not interact with the pieces on the board:

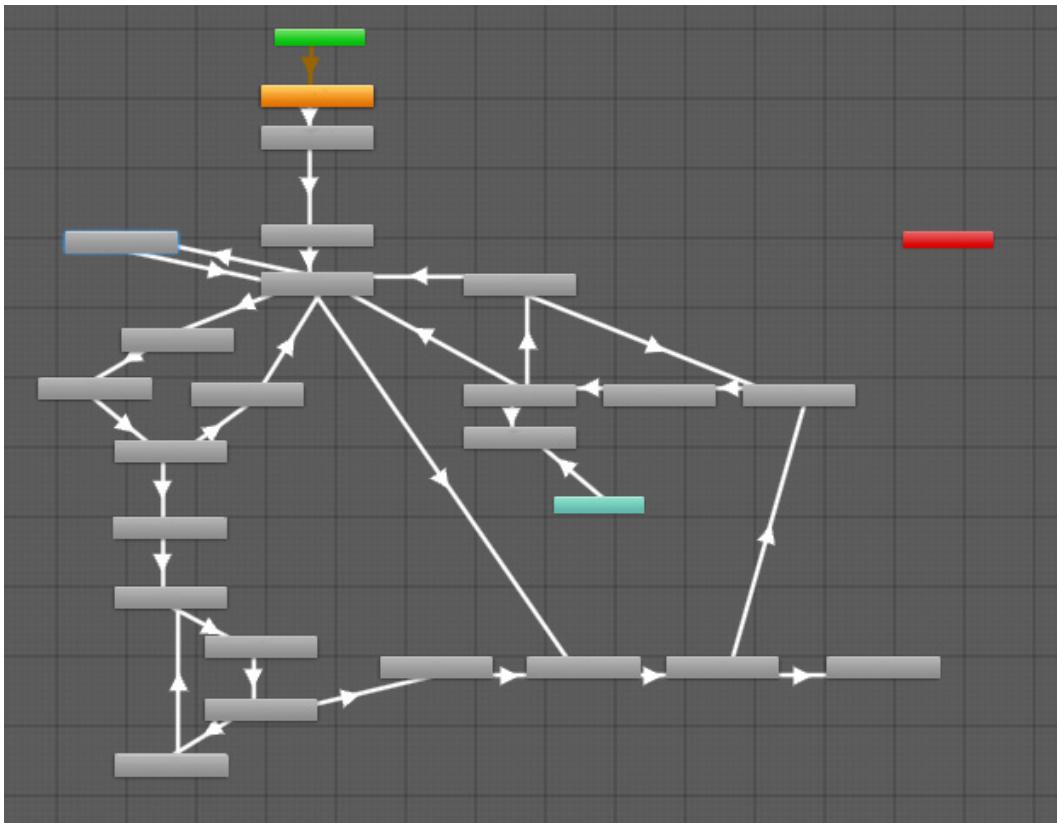


Figure 14.15: Example of a Animation State Machine used for logic

Now that we've reviewed the properties of the Animator Controller, let's look at some of its uses.

The Animator of Transition Animations

In *Chapter 9*, we took a look at Button Animation transitions and created a simple animation for a Button. We let the Button component automatically generate the Animator for us, but never looked at the Animator or did anything with it. Now that we've discussed Animators, let's look at the Animator of the Play Button saved in Assets/Animations:

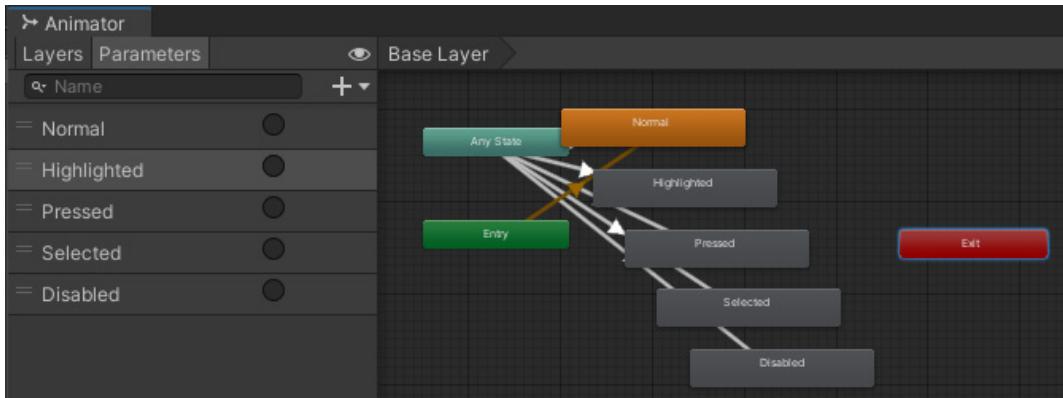


Figure 14.16: The Animator of a Button

As you can see from the preceding screenshot, the Animator that was automatically generated for us isn't particularly complicated and its setup is self-explanatory. It contains states to hold the following five Animation Clips: Normal, Highlighted, Pressed, Selected, and Disabled. All of the Animation Clips transition from Any State. Also, there are five Trigger Parameters: Normal, Highlighted, Pressed, Selected, and Disabled.

Automatically generating an Animator for any of the UI elements that allow for transition animations will result in the exact same setup. Even though this Animator is preset for you, you are free to adjust it however you see fit.

Animator layers

When using the Animator, if you have a state with transitions to multiple nodes, only one transition can occur. For example, in the following screenshot, the ChooseAState state can only transition to one of the other states at once, even if the transition conditions for all are met; this is true regardless of the type of Parameter that you use:

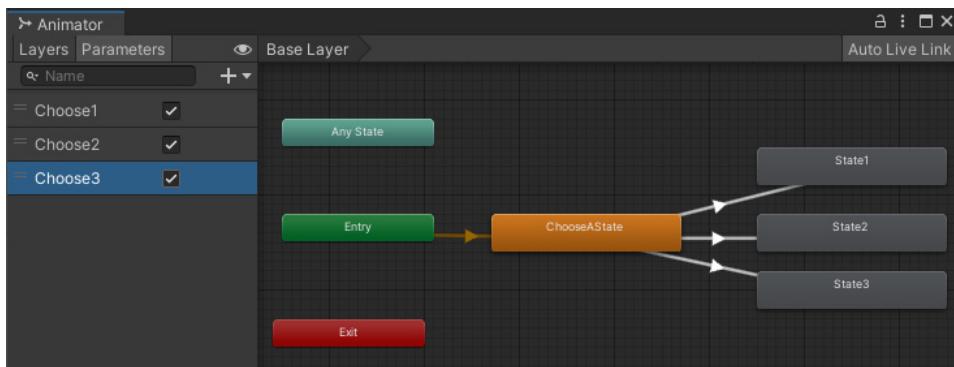


Figure 14.17: A forking animation example in the Chapter14 scene

If you want multiple animations to trigger at once, you can use Animation Layers. The following layer setup will have all three states running simultaneously:

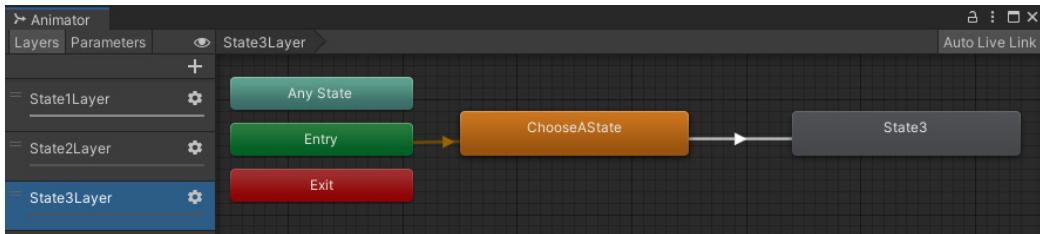


Figure 14.18: An Animation Layers Example in the Chapter14 scene

I've found that the most common need for something like this is when you have an object made of multiple sprite sheets and you want multiple sprite sheet animations to trigger at the same time and putting them all on the same Animation Clip doesn't make sense. For example, I've worked on a game where a 2D character had multiple interchangeable parts, and each part had its own sprite sheet animation. It was necessary to have the idle animation for each part start all at the same time. Since the parts could be swapped out, there were multiple combinations of parts that could be achieved, and it would not have made sense to make all the different possible idle animation combinations. It also wouldn't have made sense to give each possible part its own Animator. So, I made a layer for each body part and was able to have the individual sprite animations all play at the same time.

Setting Animation Parameters in scripts

You can set the values of the Animator Parameters via scripts using the `SetFloat()`, `SetInteger()`, `SetBool()`, `SetTrigger()`, and `ResetTrigger()` functions of the `Animator` class. You reference the Animator Parameter variables by the string names assigned to them within the Animator.

To set the Animation Parameters, you first get the Animator on which the Parameters were defined; you can do this with either a public Animator variable or using `GetComponent<Animator>()`. Then, you call the necessary function on the Animator.

Let's look at an example that would set the following **Parameters**:

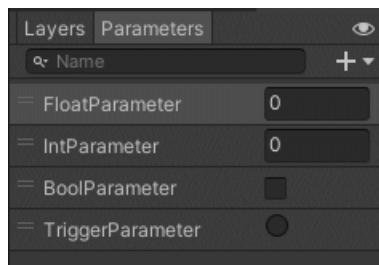


Figure 14.19: Different types of Animation Parameters

The following script would set the Animator Parameters defined in the previous screenshot:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Chapter14Examples : MonoBehaviour
{
    Animator theAnimator;

    void Awake()
    {
        theAnimator = GetComponent<Animator>();
    }

    public void SetAnimatorParameters()
    {
        theAnimator.SetFloat("FloatParameter", 1.0f);
        theAnimator.SetInteger("IntParameter", 1);
        theAnimator.SetBool("BoolParameter", true);
        theAnimator.SetTrigger("TriggerParameter"); // sets to true
        theAnimator.ResetTrigger("TriggerParameter"); // sets to false
    }

    public void ExampleFunction()
    {
        Debug.Log("The Animation Event is not sending an argument");
    }

    public void ExampleParameterFunction(int value)
    {
        Debug.Log("The Animation Event sends the following value: " +
value);
    }
}
```

The benefit of using a Trigger is that you usually don't have to reset it as it instantly resets the moment a transition uses it. However, if you set a Trigger and the transition is never reached, you will need to reset it using `ResetTrigger()`.

Animator Behaviours

If you want to write code that fires at specific points within a state, you can use a unique class of scripts known as State Machine Behaviours. State Machine Behaviours can be added to any state node you create within the Animator. I specify *you create* because you cannot add them to the **Entry** node, **Exit** node, or **Any State** node.

You can create a new State Machine Behaviour by selecting a state and clicking on **Add Behaviour**:

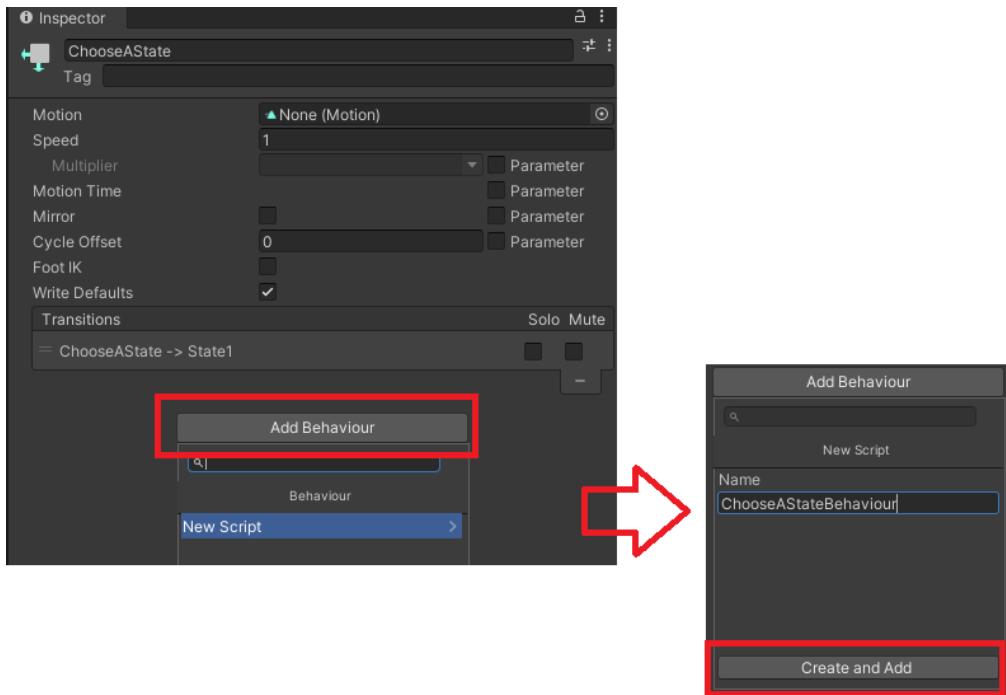


Figure 14.20: How to add a Behaviour to an Animation State

All new State Machine Behaviours created in this way are saved in the **Assets** folder.

When you open the script, it will be automatically populated with the following code:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ChooseAStateBehaviour : StateMachineBehaviour
{
    // OnStateEnter is called when a transition starts and the state
    machine starts to evaluate this state
```

```
//override
public override void OnStateEnter(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex)
{
    // Implement state enter logic here
}

// OnStateUpdate is called on each Update frame between
// OnStateEnter and OnStateExit callbacks
//override
public override void OnStateUpdate(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex)
{
    // Implement state update logic here
}

// OnStateExit is called when a transition ends and the state
// machine finishes evaluating this state
//override
public override void OnStateExit(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex)
{
    // Implement state exit logic here
}

// OnStateMove is called right after Animator.OnAnimatorMove()
//override
public override void OnStateMove(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex)
{
    // Implement code that processes and affects root motion
}

// OnStateIK is called right after Animator.OnAnimatorIK()
//override
public override void OnStateIK(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex)
{
    // Implement code that sets up animation IK (inverse
    // kinematics)
}
```

Note that this class is derived from `StateMachineBehaviour` rather than `MonoBehaviour`, like the scripts we attach to `GameObjects`.

There are a few functions prewritten in the script for you, along with descriptions of how to use them. Just as `Awake()`, `Start()`, and `Update()` are predefined functions for `MonoBehaviour`, `OnStateEnter()`, `OnStateUpdate()`, `OnStateExit()`, `OnStateIK()`, and `OnStateMove()` are predefined functions that call at specific times. You can delete whichever functions you don't want to use. You can also write other functions within this script, and they are not restricted to these predefined ones.

These functions can do whatever you want them to do, even set the Animator Parameters of your Animator.

I find State Machine Behaviours to be incredibly helpful because I use state machines extensively to control the logic of my games. Earlier, in this section, I showed you a state machine I created for a match-three RPG. I used multiple State Machine Behaviours to let my other scripts know when the states had changed, call functions from other scripts at specified times, and so on.

Now that we've reviewed using Animation clips and Animator Controllers, let's look at some examples of how to implement them in our project.

Examples

The main focus of this chapter is to provide examples of how to create common UI animations and effects, so let's get to it. In these examples, I will show you a basic re-usable animation for fading UI Panels in and out, as well as a more complex animation of a loot box dependent on player interaction.

Animating pop-up windows to fade in and out

With our first example, we will continue to work on our main scene.

Currently, we have a `Pause Panel` and `Inventory Panel` that instantly appear when the `P` or `I` key is pressed. That's not terribly interesting, so let's add some animations to have the Panels *pop* in and out with fade and scale animations.

It can get rather tedious having to expand all of the parents to see their children every time you go to a new scene. A shortcut to open all parents is to select everything in the Hierarchy and then press the right arrow key on the keyboard. You can do this multiple times if you have multiple nestings. The left arrow key will collapse the parents.

Our workflow to set up these animations and their functionality will be to create Animation Clips, set up our Animator, and then write code that sets the Animator's parameters at the appropriate times. To make the steps easier to digest, I've broken them into sections. The first section covers all of the steps involved with setting up the Animation Clips and Animator and the second section covers the sets involved with writing code.

Setting up the animations

To create a pop in and out animation on Pause Panel and Inventory Panel, perform the following steps:

1. We'll start by adding an Animation Clip to Pause Panel that will cause it to scale and fade in. Open the Animation Window and select Pause Panel from the Hierarchy. Select **Create** to add a new Animation Clip. Save the new Animation Clip in the Assets/Animation folder and name it **FadeAndScale**.
2. We want to control four properties of this Panel: its scale, its alpha value, its ability to be interacted with, and its raycast blocking. Let's start with scale. We can adjust this property from the Rect Transform component. Select **Add Property** and click on the plus sign next to **Scale** under **Rect Transform**, as follows:

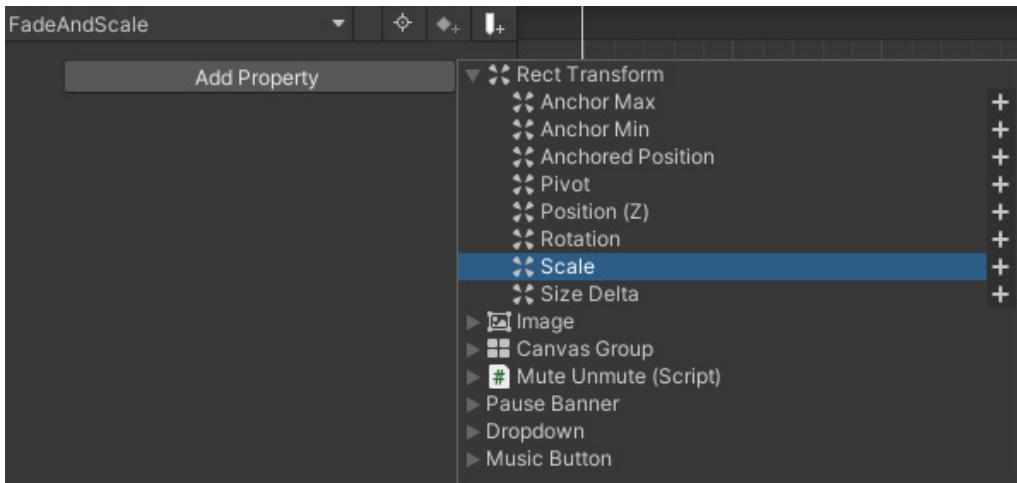


Figure 14.21: Adding the Scale property

3. Two keyframes are initialized for the **Scale** property at **0:00** and **1:00**. This means that the animation will last one second, which is a bit long for a pop-in animation. Select the keyframe at **1:00** and drag it to **0:30** to make the animation half a second long:

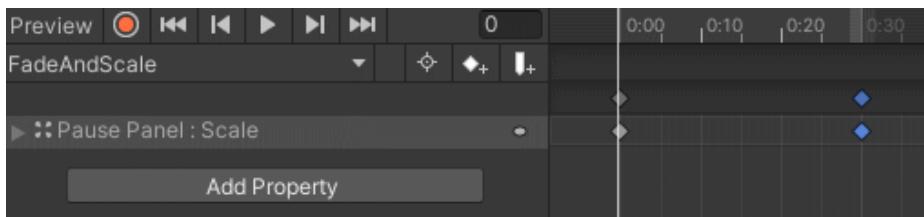


Figure 14.22: Adjusting the length of the animation

- We want the Panel to start off small and then get big. To achieve this, we will need to adjust the *x* and *y* scales. Expand the **Scale** property (select the arrow next to **Pause Panel : Scale**) to view the *x*, *y*, and *z* properties of **Scale**. Currently, the scale for all three coordinates is set to 1 at both keyframes. Since we want the scale to start small and then enlarge to its normal size, we want it to scale from 0 to 1. We only need to adjust the *x* and *y* scales since the *z* scale doesn't really affect a 2D object. So, at keyframe **0:00**, change the **Scale.x** and **Scale.y** properties to 0 by typing 0 in their property boxes, as follows:

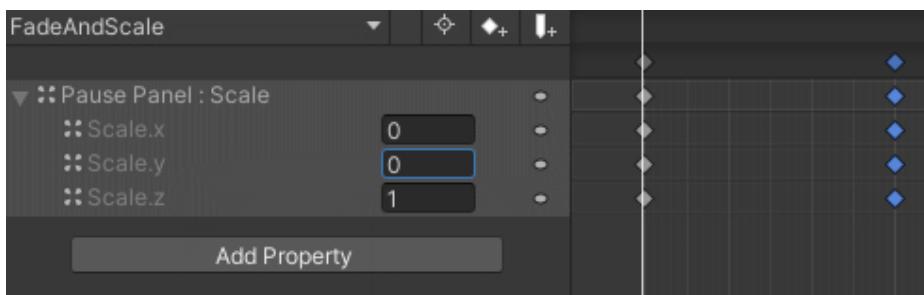


Figure 14.23: Adjusting the Scale properties

If you play the animation, you'll note that **Pause Panel** is quickly scaling in. Since parent/child relationships scale children with their parents, we don't have to animate the scale of all the **Pause Panel** children separately.

- Now, let's control the alpha of the Panel to fade it in. We don't want to animate the alpha on the **Color** property of the **Image** component of **Pause Panel**. Doing so would only affect the alpha of the Panel's background Image and would not affect the children. Instead, we want to animate the alpha value on the **Canvas Group** component. This will make the alpha of the **Pause Panel** and all its children work in unison. Remember that this was the whole reason we used a **Canvas Group** component to begin with. Let's select **Add Property | Canvas Group | Alpha**:

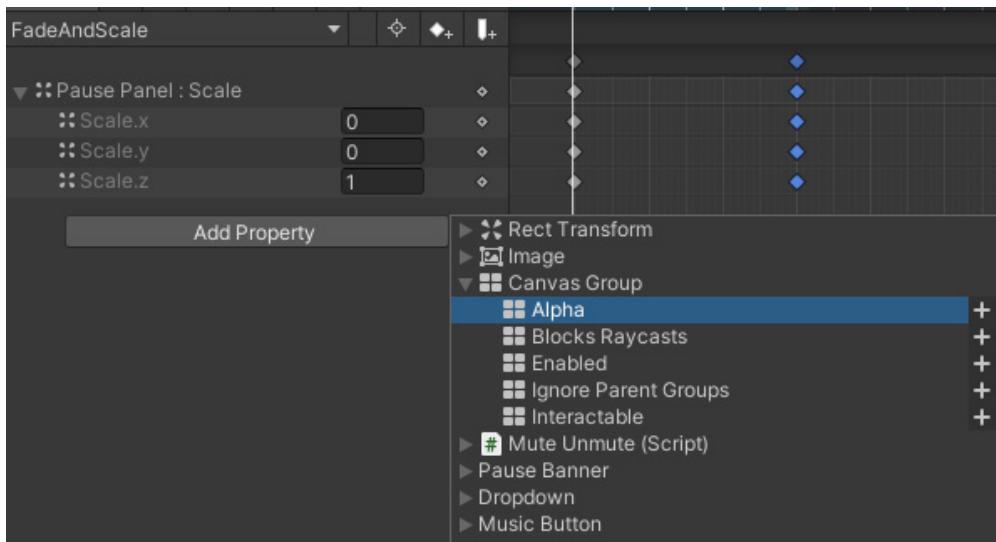


Figure 14.24: Adding the Alpha property

6. To fade in, the Panel should start completely transparent and end completely opaque. Both keyframes already have the **CanvasGroup.Alpha** property set to 1, so change the **CanvasGroup.Alpha** property to 0 at the first keyframe. Playing the animation (or scrubbing the playhead) will show the Panel and all of its children fading in.
7. We will need to animate whether or not the Panel can be interacted with. We don't want the player to be able to interact with the dropdown menu or mute buttons when the Panel is still popping in. This too is controlled by the Canvas Group component—select **Add Property | Canvas Group | Interactable**.
8. The Pause Panel and its children should be interactable only after it has fully popped into the scene. So, deselect the checkbox next to **CanvasGroup.Interactable** at the first keyframe to make the Canvas Group animate from interactable to not interactable. When you scrub the playhead, you'll see that the property does not turn back on until the very last frame.
9. The last item we will need to animate is its raycast blocking. Select **Add Property | Canvas Group | Blocks Raycast**. Animate it from `false` to `true` as you did with the **Interactable** property. Your **FadeAndScale** Animation Clip's timeline should now appear, as follows:

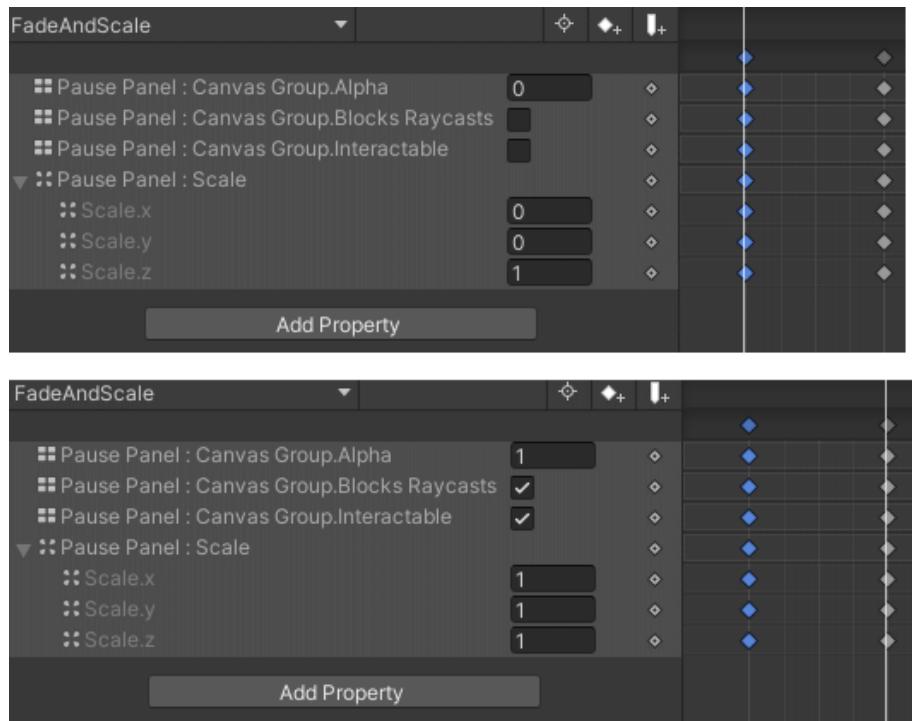


Figure 14.25: The Timeline of the FadeAndScale Animation Clip

10. Now that we have our Animation Clip set up, we can start working inside the Animator. When we created the `FadeAndScale` Animation Clip, an Animator named `Pause Panel` was automatically created. An Animator component was also added to the `Pause Panel` GameObject, with the `Pause Panel` Animator assigned to it.

With `Pause Panel` selected, open the Animator Window. You should see something similar to the following:

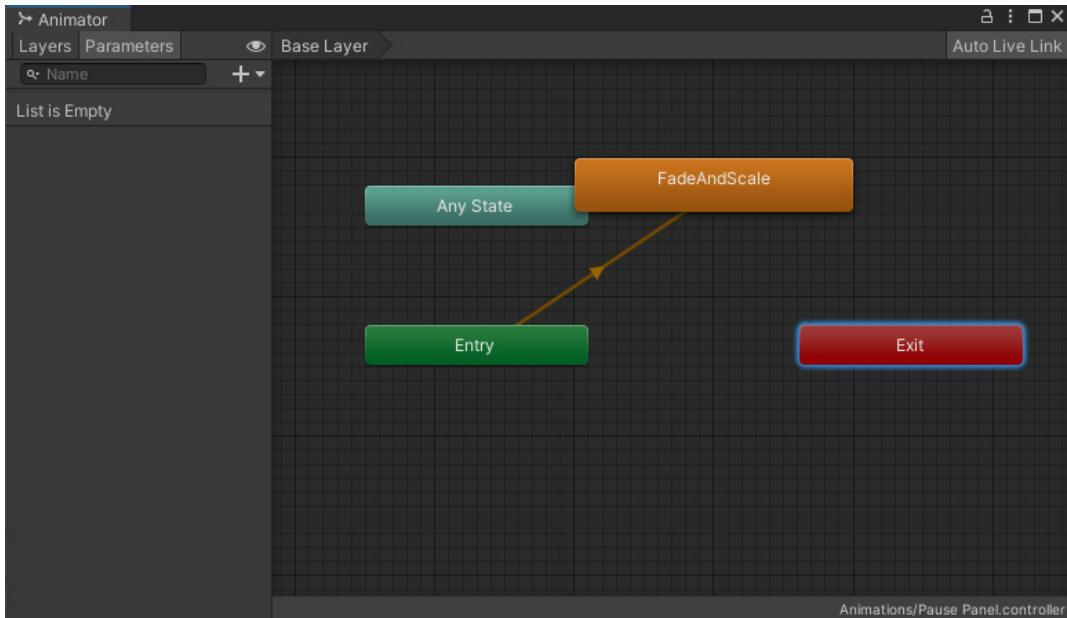


Figure 14.26: The Animator of Pause Panel

You will see a state named **FadeAndScale**. This state uses the **FadeAndScale Animation Clip** as its **Motion**, which you can view in its Inspector.

11. Currently, since the **FadeAndScale** state is connected to the **Entry** node and set as the Layer Default State, when we play the game, the **FadeAndScale** animation will play instantly. It will also play on a loop. That's not at all what we want, obviously. Let's stop it from playing when the game starts by creating an empty state as the Layer Default State. Right-click anywhere within the Animator and select **Create State | Empty**:

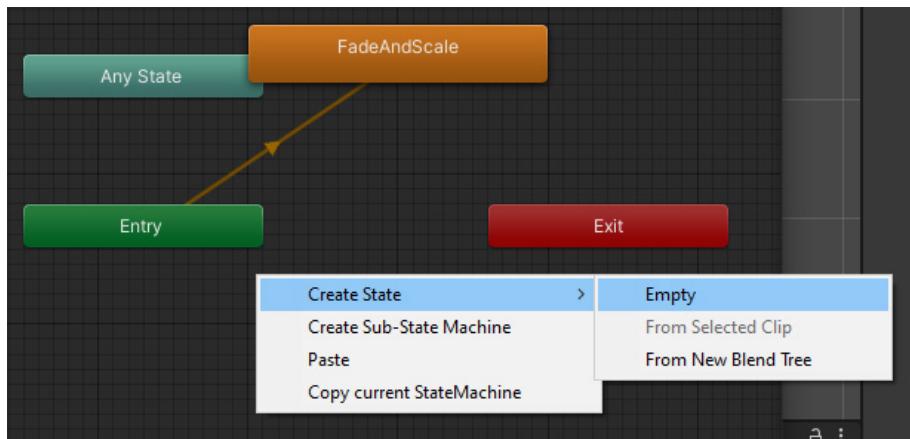


Figure 14.27: Creating an Empty State

This will add a gray-colored state named `New State` to the Animator.

12. Rename `New State` to `Empty State` by changing its name in its Inspector.
13. Set `Empty State` to the Layer Default State by right-clicking on it and selecting **Set as Layer Default State**:

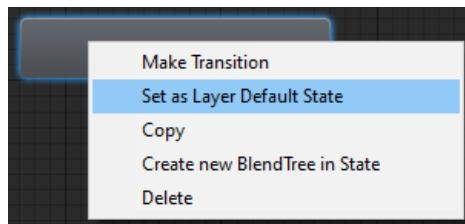


Figure 14.28: Setting a state as the Layer Default

It will now be connected to `Entry` via a transition. `FadeAndScale` will also now no longer be the Layer Default State and will not have any transitions connected to it:

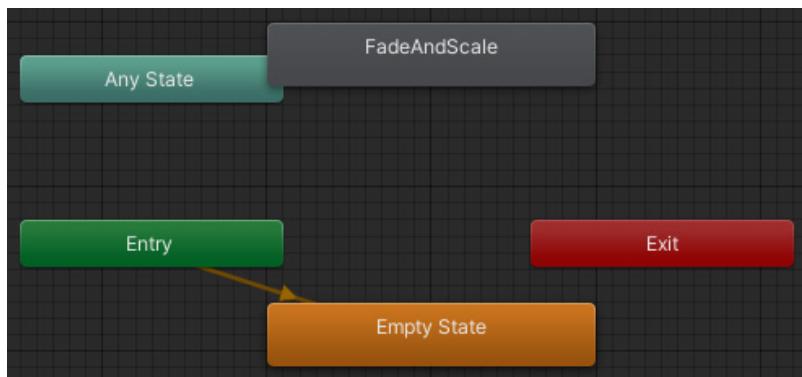


Figure 14.29: Empty State set as the Layer Default

We used an empty state as our Layer Default State because we want the Panel to do nothing, while it waits for us to tell it to start animating.

14. Rearrange the items to a more viewable layout. I personally like the following layout for this Animator:

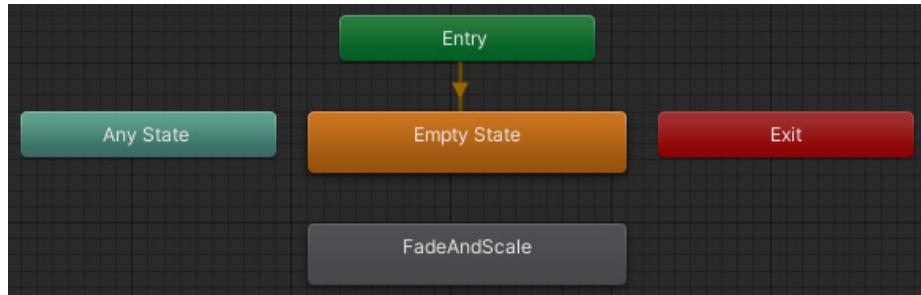


Figure 14.30: Rearranging the States

15. The **FadeAndScale** Animation Clip now no longer instantly plays when the game starts. It is still set to have its animation loop, however. To fix this, select the **FadeAndScale** Animation Clip from your **Project** folder view. This will bring up its Inspector.

Deselect the **Loop Time** property to disable looping on the animation:

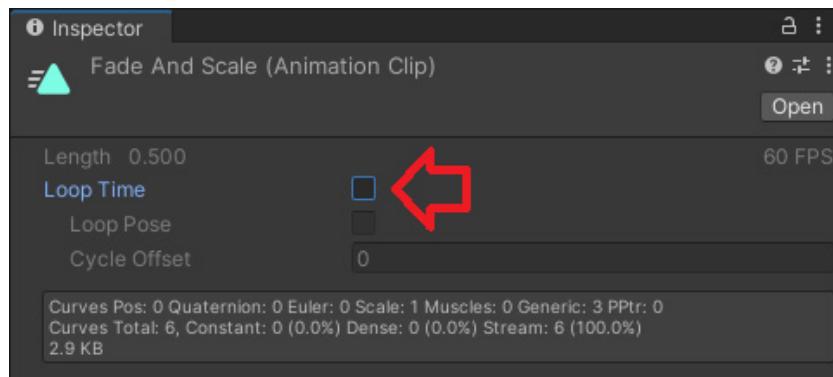


Figure 14.31: Deselecting Loop Time

16. We want the **Pause Panel** to be able to fade in and out on command, but we don't have an Animation Clip for fading out, only one for fading in. We actually don't need to create a whole Animation Clip to achieve this motion. We can simply play the **FadeAndScale** Animation Clip backward. Duplicate the **FadeAndScale** state by selecting it and pressing **Ctrl + D**. This will give you a new state named **FadeAndScale 0** that has the **FadeAndScale** Animation Clip set as its **Motion**:

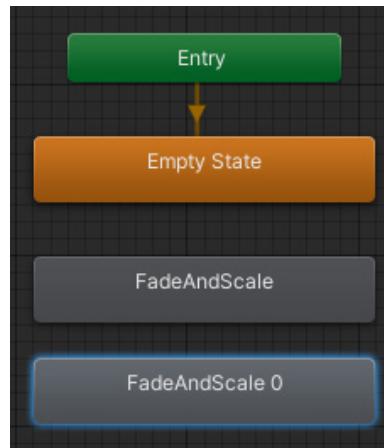


Figure 14.32: Duplicating a State

17. Rename the `FadeAndScale` state to `FadeAndScaleIn` and the `FadeAndScale 0` state to `FadeAndScaleOut`.
18. To set the `FadeAndScaleOut` state to play the `FadeAndScale` Animation Clip backward, we simply have to change its **Speed** to `-1` in its Inspector:

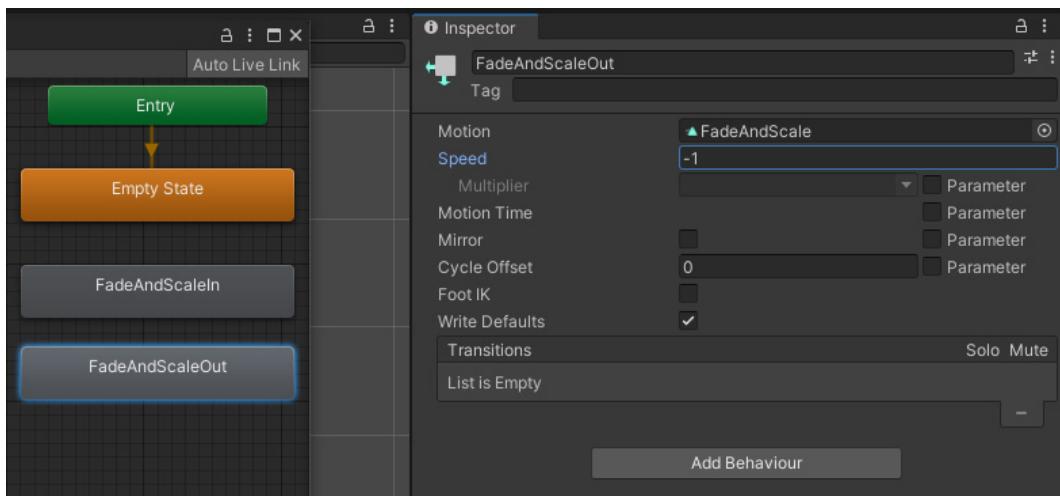


Figure 14.33: Setting an animation to play in reverse

19. Now that we have all of our states set up properly, we can create the transitions between them. Start by creating two Trigger Parameters named `FadeIn` and `FadeOut`:

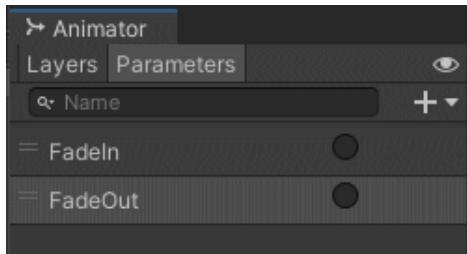


Figure 14.34: The Animator Parameters

We are using Trigger Parameters here because we want these values to instantly reset after we've used them. That way, we can create an animation cycle without having to write code that resets the parameter values.

20. You create transitions between states by right-clicking on the first state, selecting **Make Transition**, and then clicking on the second state:

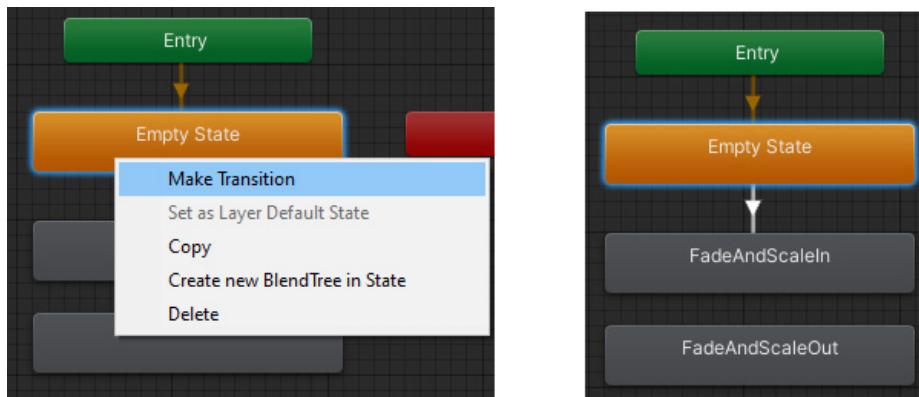


Figure 14.35: Making transitions

Create transitions between the states in the following manner:

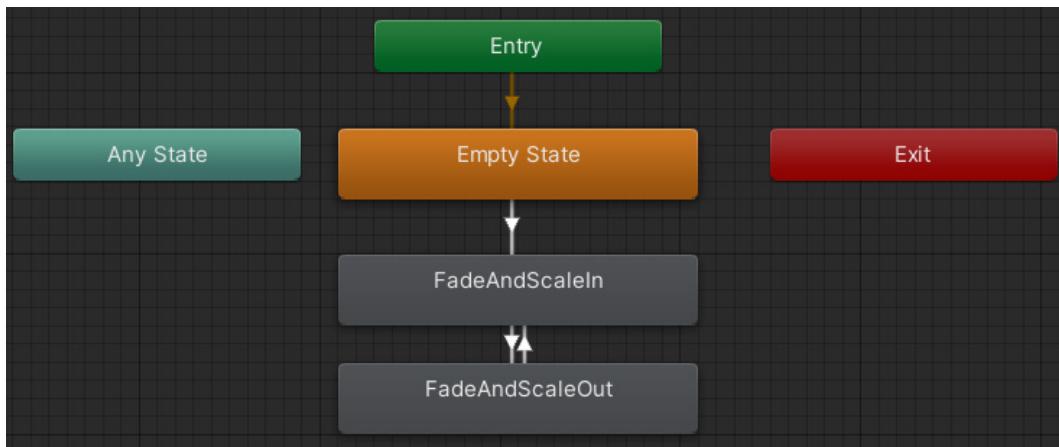


Figure 14.36: The final state transition layout

This transition flow will allow the Panel to transition from no animation to the **FadeAndScale** Animation Clip, then to the reversed **FadeAndScale** Animation Clip, and back and forth between the two.

21. If you play the game, **Pause Panel** will instantly go from the **Empty State** to the **FadeAndScaleIn** state and then to the **FadeAndScaleOut** state, and back and forth between the two indefinitely. This is because transitions are automatically set to occur after an animation is complete. To stop this, you have to tell the transitions to only occur after a parameter has been set. Select the transition between the **Empty State** and the **FadeAndScaleIn** state. Select the plus sign in the **Conditions** list to add a new condition. Ensure that the condition is set to the **FadeIn Trigger**. Since we don't want timing to be a factor in our transition, deselect **Has Exit Time**:

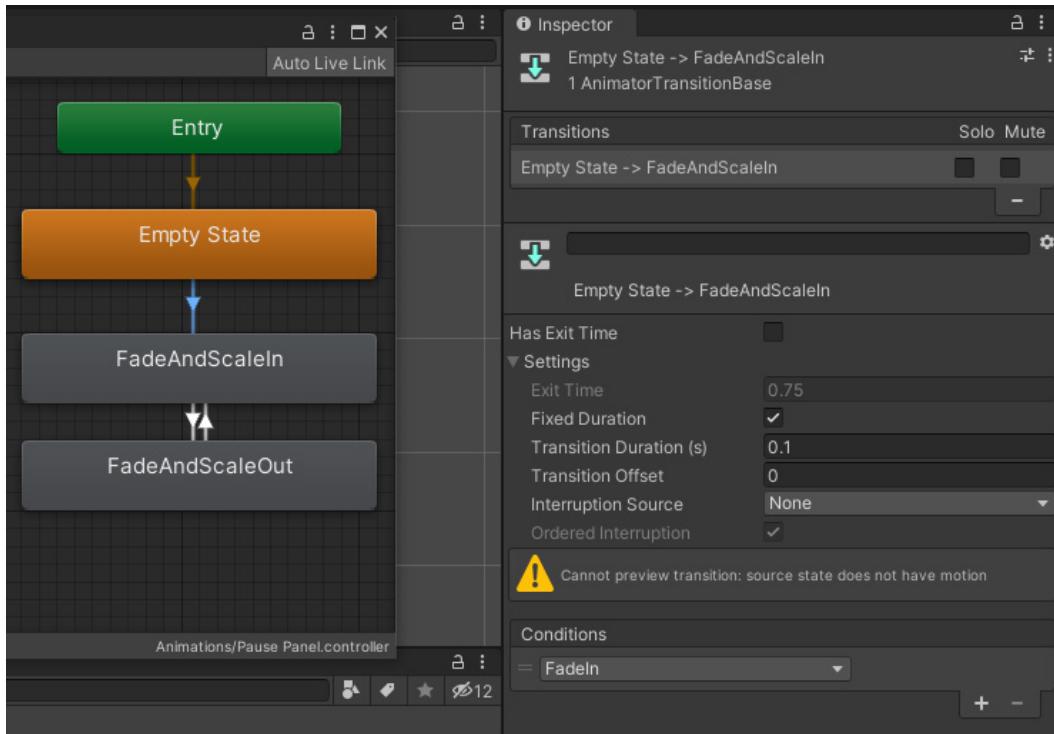


Figure 14.37: The transition properties

22. Complete the same steps for the transitions between the **FadeAndScaleIn** and **FadeAndScaleOut** States. Set the **Condition** for the transition from the **FadeAndScaleIn** State to the **FadeAndScaleOut** State to the **FadeOut** Trigger. Set the **Condition** for the transition from the **FadeAndScaleOut** state to **FadeAndScaleIn** State to the **FadeIn** Trigger. Additionally, deselect **Fixed Duration** and set all values to 0 to ensure an instant transition:

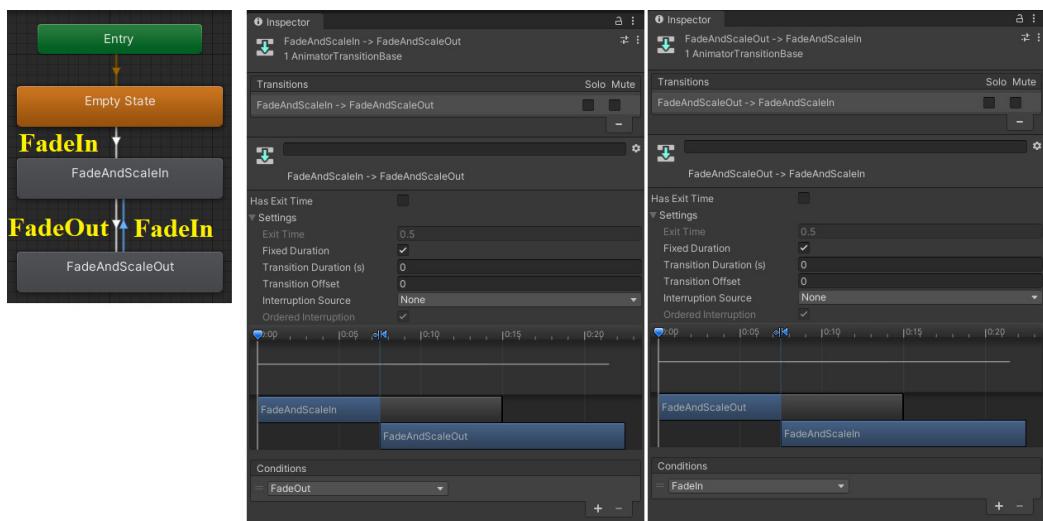


Figure 14.38: How to set up the transitions of the Pause Panel

23. If you play the game now, you'll note that the `Pause Panel` is visible when the game starts. This is because our animation states supersede the code we wrote in `ShowHidePanels.cs` that made the `Pause Panel` invisible at the start of the scene. We'll deal with our broken code later, but for now, make the `Pause Panel` invisible at start by setting its **Canvas Group** component to have the following values:

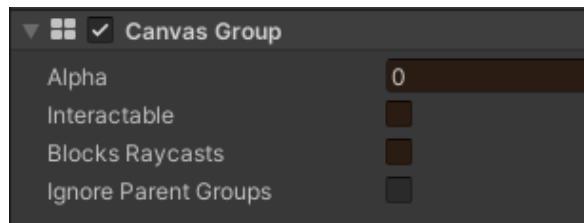


Figure 14.39: The properties of the Canvas Group

Now when you play the game, the `Pause Panel` will not appear at the start.

24. We've completed setting up the animations for the `Pause Panel`, but before you proceed to the `Inventory Panel`, check whether the animations are working correctly. To do so, arrange your windows so that your Game view and Animator Window are both visible:

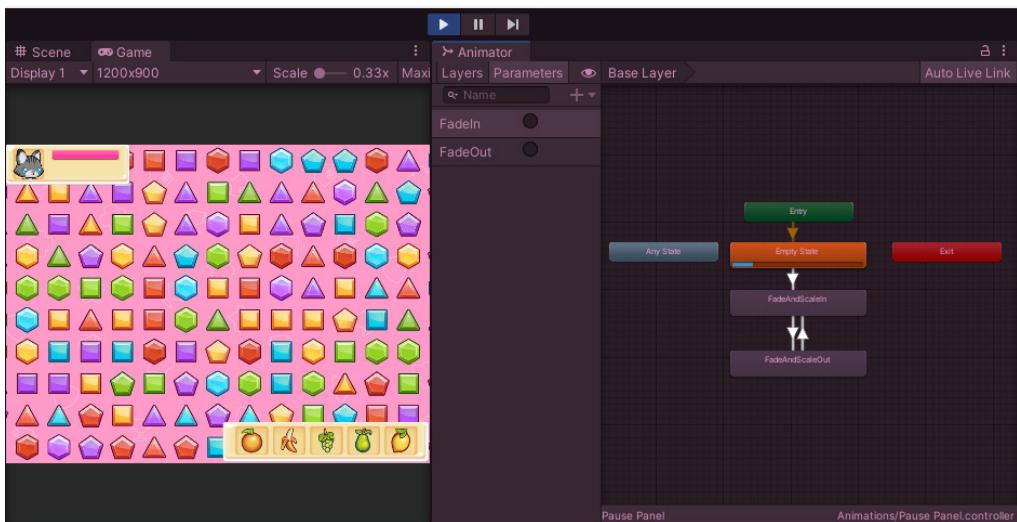


Figure 14.40: Playing the Empty State

You'll see the progress bar on the current state of `Pause Panel` running. To force the transitions, click on the circles next to the appropriate Trigger parameters.

25. If your animations are working the way they should, you can now set up the animations on the `Inventory Panel`. You may be thinking, "Ugh, now I have to do all of this again for the `Inventory Panel`!?" However, worry not, you don't have to do it again, because you can reuse the Animator for the `Pause Panel` on the `Inventory Panel`. All of the properties we changed on the `Pause Panel` in the `FadeAndScale` Animation Clip also exist on the `Inventory Panel`. So, to give the same set of animations and controls to the `Inventory Panel`, we can simply attach the Animator we created to `Inventory Panel`.

Since we will be using the Animator we created on two different objects, it is a good idea to rename it. Change the name from `Pause Panel` to `PopUpPanels`. Now, drag it onto the `Inventory Panel`.

26. Just as we had to change the properties of the `Canvas Group` on `Pause Panel` to stop it from appearing when the scene starts, we also have to change the properties on the **Canvas Group** component of the `Inventory Panel`. Set the properties as they appear in *Figure 14.39*.

Now that we are done setting up our animations for our two Panels, we can begin writing code that will trigger the animations when the *P* and *I* keys are hit to bring up the Panels.

Setting the Animator's Parameters with code

We have a script named `ShowHidePanels.cs` attached to the `Main Camera` that would bring the `Pause Panel` and `Inventory Panel` up when the *P* and *I* keys, respectively, are pressed. Sadly, it no longer functions, since the animations now supersede the properties of the `Canvas Groups` we set within it. We can reuse the logic but will have to do a bit of work to get our Panels popping up again.

The changes that we will make to `ShowHidePanels.cs` will cause the Panels in preceding chapter scenes to stop appearing. If you plan on accessing the previous chapter scenes, save a secondary copy of this script as it is now so that you can access it later.

To trigger the animations on the `Pause Panel` and `Inventory Panel` with code, complete the following steps:

1. Open the `ShowHidePanels.cs` script. Comment out all the code in the `Start()` and `Update()` methods that use `TogglePanel()`. After you comment out those lines, you should see the following:

```
void Start()
{
    // Initialize Panels on Start
    TogglePanel(inventoryPanel, inventoryUp);
    TogglePanel(pausePanel, pauseUp);
}

void Update()
{
    // Handle inventory Panel toggle
    if (Input.GetKeyDown(KeyCode.I) && !pauseUp)
    {
        inventoryUp = !inventoryUp;
        TogglePanel(inventoryPanel, inventoryUp);
    }

    // Handle pause Panel toggle
    if (Input.GetButtonDown("Pause"))
    {
        pauseUp = !pauseUp;
        TogglePanel(pausePanel, pauseUp);
        Time.timeScale = pauseUp ? 0 : 1; // Pause/unpause game
        by setting time scale
    }
}
```

2. Instead of referencing the `Pause Panel` and `Inventory Panel` with their Canvas Group components, we'll now reference them with their Animator components. Create the following two variable declarations at the top of the class:

```
public Animator inventoryPanelAnim;
public Animator pausePanelAnim;
```

3. Now let's create a new method called `FadePanel()` that accepts `Animator` and `bool` parameters:

```
public void FadePanel(Animator anim, bool show)
{
    if (show)
    {
        anim.SetTrigger("FadeIn");
    }
    else
    {
        anim.SetTrigger("FadeOut");
    }
}
```

4. Update the `Update()` method to now call the new `FadePanel()` method where the `TogglePanel()` method was once called. The bolded text in the following code signifies the added code:

```
void Update()
{
    // inventory Panel
    if (Input.GetKeyDown(KeyCode.I) && !pauseUp)
    {
        inventoryUp = !inventoryUp;
        //TogglePanel(inventoryPanel, inventoryUp);
        FadePanel(inventoryPanelAnimator, inventoryUp);
    }

    // pause Panel
    if (Input.GetButtonDown("Pause"))
    {
        pauseUp = !pauseUp;
        //TogglePanel(pausePanel, pauseUp);
        FadePanel(pausePanelAnimator, pauseUp);
        Time.timeScale = Convert.ToInt32(pauseUp);
    }
}
```

That's all we have to do to the code.

5. Drag the `Inventory Panel` and the `Pause Panel` into their appropriate slots on the `Show Hide Panels` component:

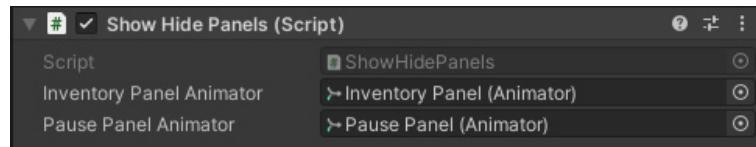


Figure 14.41: The properties of the Show Hide Panels component

6. Play the game and watch it kind of work. `Inventory Panel` should appear and disappear as necessary, but the `Pause Panel` won't fade out the way it is supposed to. This is because the following line of code stops all animations from happening that depend on time scale:

```
Time.timeScale = Convert.ToInt32(pauseUp);
```

That line of code was used to effectively pause the game. However, that means it's pausing our `Pause Panel` pop-up animation. Don't worry, you can still use this simple pause code and run animations when the game is paused. All you have to do is tell the Animator on the `Pause Panel` that it can still function when the time scale is set to 0. You do this by changing the **Update Mode** on the Animator component from **Normal** to **Unscaled Time**:

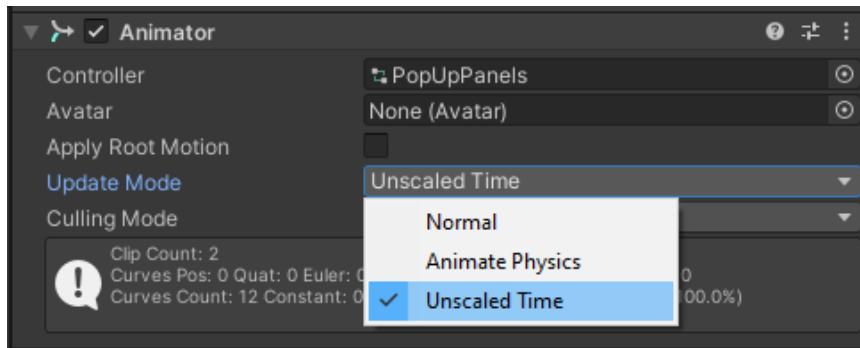


Figure 14.42: Setting Update Mode to Unscaled Time

7. Set the **Update Mode** to **Unscaled Time** on the `Inventory Panel`'s Animator component as well.

Play the game, and you should have smoothly animating Pause and Inventory Panels. We can now move on to a more complex animation.

Animating a complex loot box

For this example, we'll work with a new scene. The animation we will create is a bit complicated—a chest will fly into the scene and then wait for the player to open it. Once the player opens it, the chest will animate open with a particle system that pops in front of it. Then, three collectibles will fly out in sequence. Each collectible will have its own *shiny* animation that begins to play.

The following figure is a storyboard of sorts that shows a few keyframes of the animation playing out:

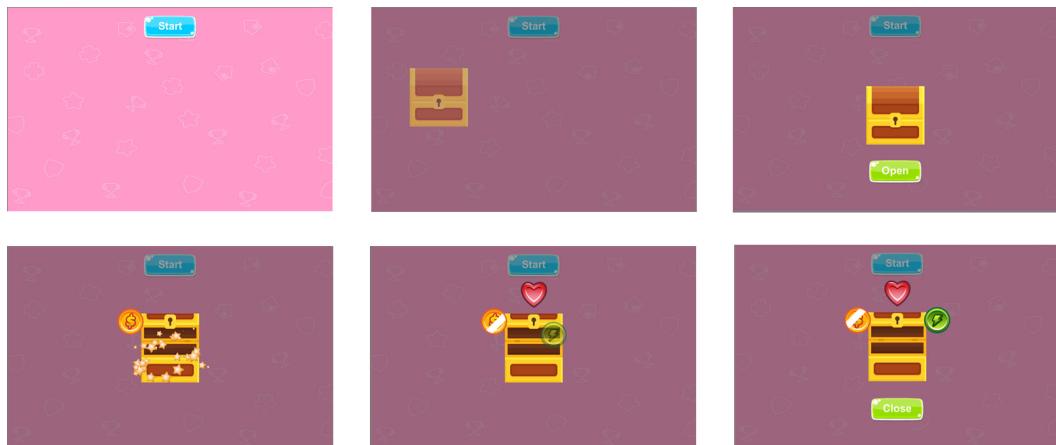


Figure 14.43: The final version of the animation from this example

In this chapter, we will cover all items except for the Particle System, which we will cover in the next chapter.

Note

The chest sprite sheet was obtained from <https://bayat.itch.io/platform-game-assets> and the item sprite sheets were obtained from <https://opengameart.org/content/shining-coin-shining-health-shining-power-up-sprite-sheets>.

This example has a lot going on with it. It is not particularly complicated to build out, but providing the steps to build it entirely from scratch would require too many steps. That would go beyond the scope of this book; but at this point in the book, you can hopefully look at a scene that's already been built out and understand how it was achieved. Therefore, we will start this example with a package file that has all the items placed in the scene, some of the animations already created, and all the new sprite sheets included.

Before you begin, import the Chapter 14 - Examples - LootBox - Start.unitypackage asset package.

If you'd like to view the completed example, view the package labeled Chapter 14 - Examples - LootBox - End.unitypackage.

Note

Unity Layers do not save in Unity asset packages. The example will describe creating a Layer named UI_Particles and having the cameras ignore or include the Layer, but this is not displayed in the provided package.

To make the example easier to absorb, I have broken the steps into three distinct sections. To complete this example, we will perform the following functions:

1. Set up the various animation clips for the individual items.
2. Tie all the animations together and make sure that they are timed appropriately using a state machine, aka Animator.
3. Create the Particle System that displays when the chest opens and make sure that it displays properly within the UI (which will be completed in the following chapter).

Setting up the animations

Let's start this section by creating the animations for each of the objects within the scene. To create all the animations for this scene, complete the following steps:

1. If you have not done so already, import Chapter 14 - Examples - LootBox - Start.unitypackage. You should see a scene that has two Canvases—one with a button and a background image and another with a chest, items, and a button, as shown in the following screenshot:

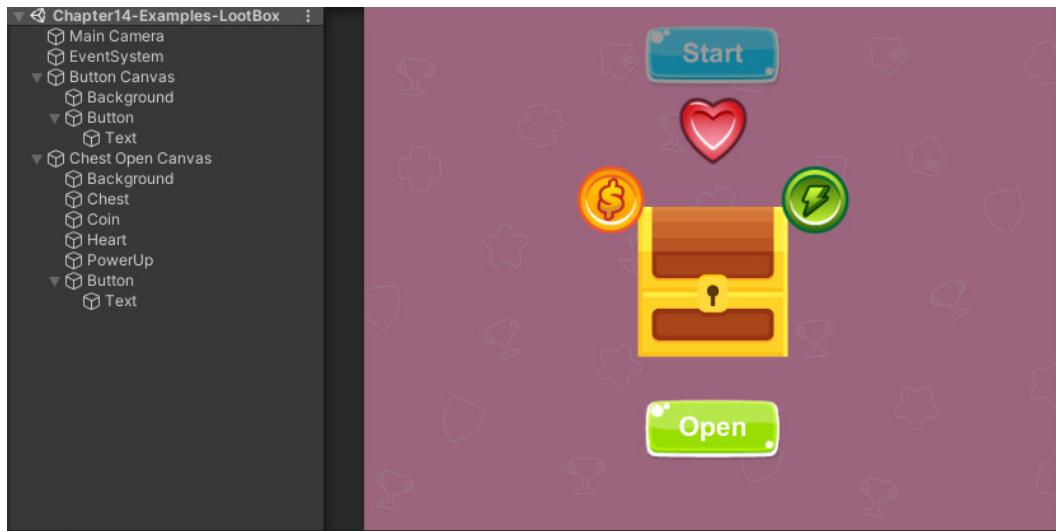


Figure 14.44: The scene layout of the package

After importing the package, you will also note that there are a few animation clips and controllers provided in the **Assets** folder.

2. The **Chest** object needs two animations, one of it *flying in from the side of the screen* and one of its *sprite sheet opening*. Let's make the flying in animation first. Select the **Chest** object in the Hierarchy and then select **Create** in the Animation Window to create a new Animation Clip. Name the new Animation Clip **ChestFlyingIn. anim** and save it in the **Assets/Animations/LootBox/Clips** folder.
3. A new Animation Controller named **Chest.controller** was automatically created in the **Assets/Animations/LootBox/Clips** folder. Move it to the **Assets/Animations/LootBox/Controllers** folder.
4. Within the Animation window, select **Add Property** and expand **Rect Transform**. Click on the plus sign next to **Anchored Position**. This will allow us to animate the position of the **Chest** within the **Canvas**:

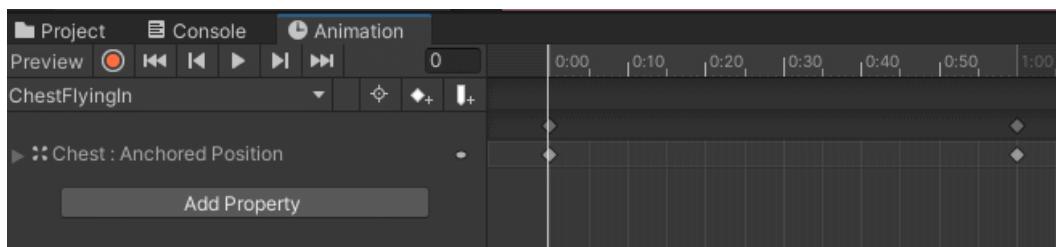


Figure 14.45: The ChestFlyingIn animation

5. Currently, the animation is one second long, which is a bit longer than we want. Move the second keyframe to the 0 : 30 mark so that it will be half a second long.
6. We will be moving objects in the scene at various keyframes and will want them to be saved in the animation. So, to have the animation record any of the changes you make in the scene, select the record button in the Animation window.
7. The position of the chest in the scene right now is where we want it to be at the end of the fly-in animation, so we will not affect the position at the second keyframe. However, we want it to start off screen, so with the animation playhead on the first frame, use the **Move** tool to move the chest outside of the Canvas area. Since record is selected, this will update the Chest's position at the first frame. This is indicated by the red tint on the Rect Transform.

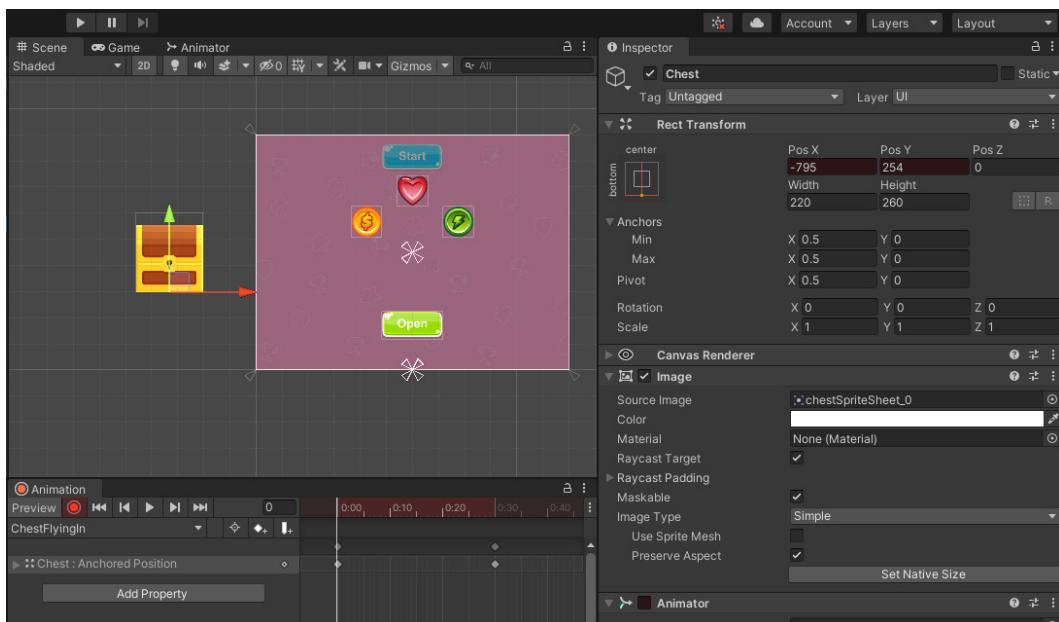


Figure 14.46: Moving the chest and recording the properties

If you scrub the playhead, you will see the chest move from left to right.

8. Now, let's make the chest fly in at an arc instead of a straight line. We'll do this with Animation Curves. Select the **Curves** tab, as follows:

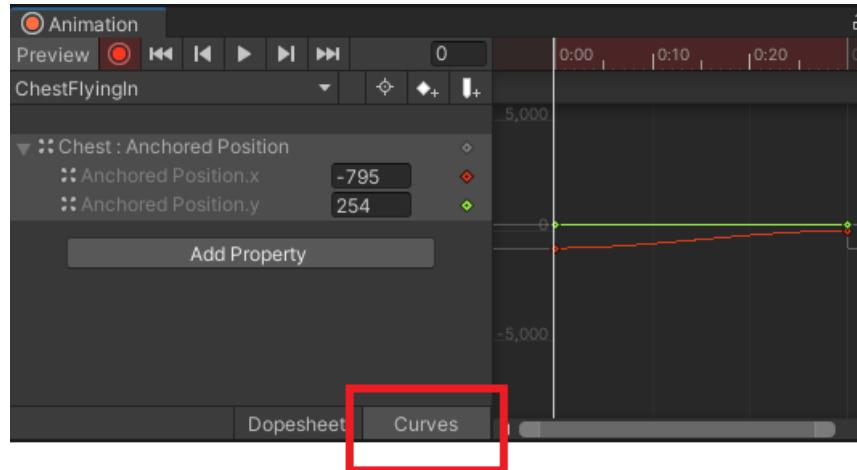


Figure 14.47: Selecting the Curves menu

The green line represents the *y* property of the anchored position. Select the **Anchored Postion.y** property to focus on it.

9. Select the first and second key frame anchors to affect their handles. Move their handles until the green curve looks more like an arc:

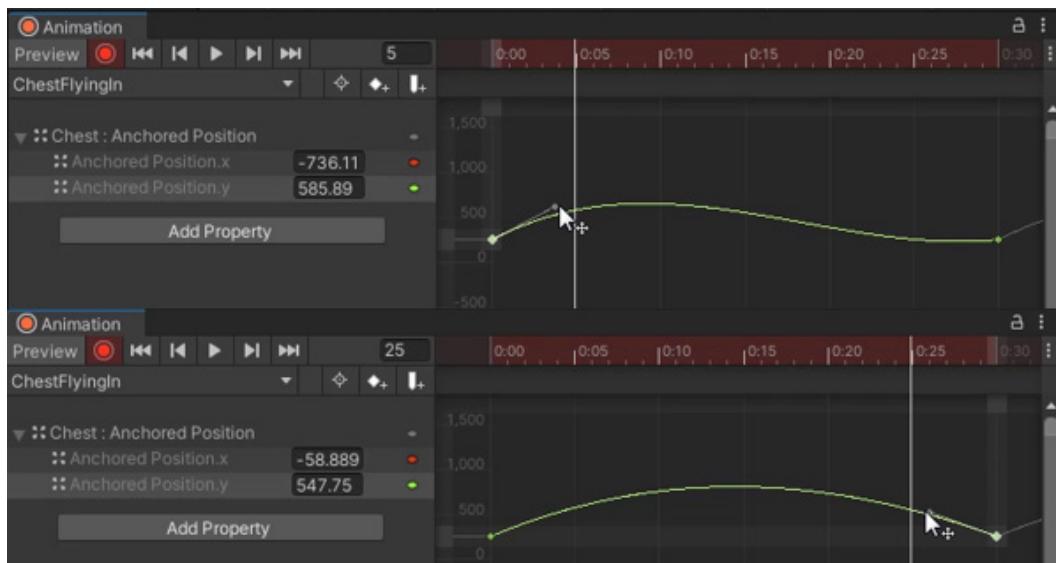


Figure 14.48: Adjusting the anchors of the curve

Now, when you play the animation, you will see the chest move in an arcing path.

- Let's have the chest fade in as it flies by adding a **color** property to the animation. Return to the **Dopesheet** view of the timeline. Select **Add Property**, then expand **Image**. Click on the plus sign next to **Color**. On the first frame, change the **Color.a** property to 0 to make the chest invisible on the first frame:

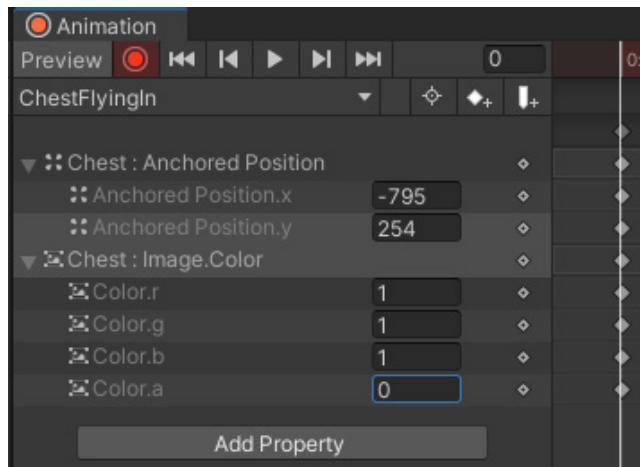


Figure 14.49: The properties of the first frame of the ChestFlyingIn animation

- Whenever a new animation is created, it is automatically set to loop. We don't want this animation to play on a loop, so select the **ChestFlyingIn** Animation Clip from the **Project** view to see its **Inspector** properties. Deselect the **Loop Time** checkbox.
- We don't want the **ChestFlyingIn** Animation Clip to automatically play when the scene starts. Since this was the first Animation Clip created for the **Chest**, it will be set to the Animator default state. With the **Chest** selected, open the Animator Window.
- Create a new default state by right-clicking within the Animator Window and selecting **Create State | Empty**. Rename the new state **Empty State** and set it as the default state by right-clicking on it and selecting **Set as Layer Default State**. We will do more with the **Chest**'s Animator later, but for now, this is all we are going to do.



Figure 14.50: The Animator of the Chest

- Now, let's set up the chest opening animation. With Chest still selected, in the Animation window, select **Create New Clip...** from the Animation Clip dropdown list:

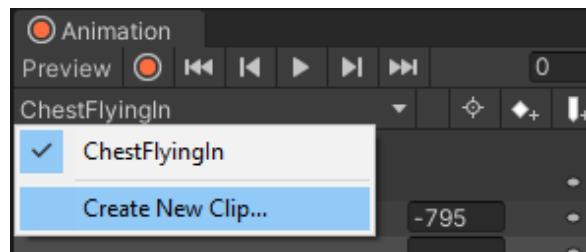


Figure 14.51: Creating a new clip

Name the new Animation Clip `ChestOpening. anim` and save it in the `Assets/ Animations/LootBox/Clips` folder.

- This Animation Clip will contain all the sprites from the sprite sheet. From the Project view, drag and drop all the sub-sprites into the Animation timeline. A new property for **Image.Sprite** will automatically be added, and all the sub-sprites will be added to the timeline in a sequence:

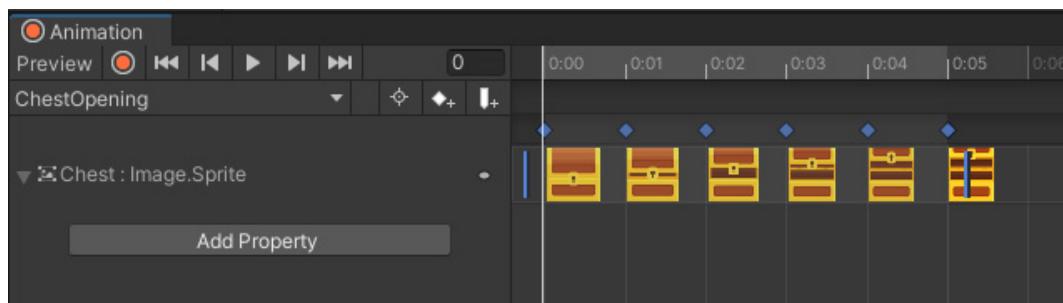


Figure 14.52: The Chest sprite sheet animation

16. Right now, the animation is way too fast. It is running at 60 frames per second, and there are only 6 frames. We need to change the frame rate. First, we must enable the Sample Rate option in the Animation Window. Select the three dots on the timeline and then select **Show Sample Rate**.

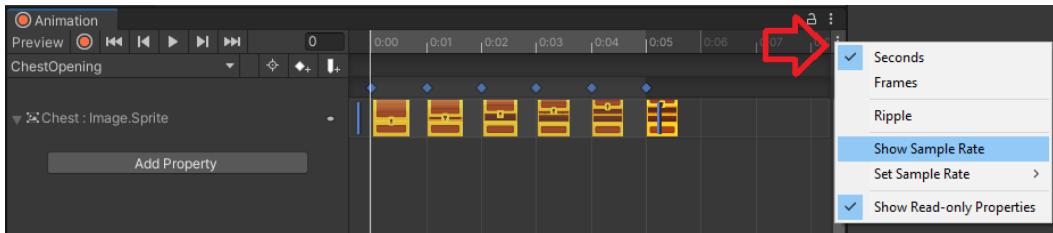


Figure 14.53: Enabling the Sample Rate menu item

17. Now that the **Samples** option is visible in the window, change the animation's **Samples** value to 12; this will change the animation's frame rate to 12 fps:

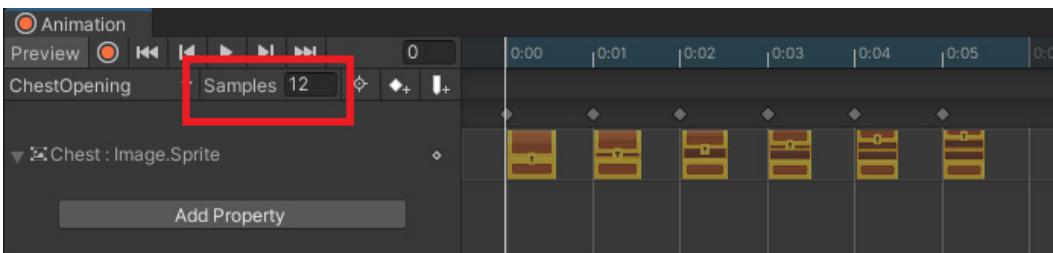


Figure 14.54: Changing the Sample Rate

18. Since Chest has an animation that will affect its alpha value, let's ensure that this animation has full alpha whenever it plays. Select **Add Property**, then expand **Image**. Click on the plus sign next to **Color**. Ensure that the **Color.a** property is 1 on both the first and last frames. Technically, we only need the alpha set to 1 on the first frame, but I like to add a start and end frame here so that I am very sure about what the animation is doing with that value.
19. We don't want this animation to play on a loop, so select the **ChestOpening** Animation Clip from the **Project** view to take a look at its **Inspector** properties. Deselect the **Loop Time** checkbox.
20. All of the other animations needed for this example have already been set up. They are very similar to this one in setup or the ones created in the preceding example. However, they are not hooked up to the correct GameObjects.

Let's give the other objects in `Chest Open Canvas` their animations. Drag the `Coin Animator` to the Inspector of the `Coin GameObject`, the `Heart Animator` to the Inspector of the `Heart GameObject`, and the `PowerUp Animator` to the `PowerUp GameObject`'s Inspector. Each of these Animators can be found in the `Assets/Animations/Loot Box/Controllers` folder. You can now preview the animations of all the items popping out of the chests and shining by selecting the GameObjects and pressing play in the Animation window (Play in the Game view will not yet show the animations playing).

21. Let's initialize the `Chest` and all of the objects as invisible. Select the `Chest`, `Coin`, `Heart`, and `PowerUp` GameObjects from the Hierarchy. In their Image component, change the alpha values of their **Color** to 0.
22. The `Chest Open Canvas` and the `Button` on that Canvas both have animations as well. These animations will affect a Canvas Group component on the objects. Right now, they don't have Canvas Group components, though. So, add a Canvas Group component to `Chest Open Canvas` and its child `Button`.
23. Initialize the two new Canvas Group components to have **Alpha** values of 0 and their **Interactable** and **Blocks Raycasts** properties set to `false`.

You should now only see the Start Button in the scene.

24. Now, add the `CanvasGroupFadeInOut` Animator to `Chest Open Canvas` and its child `Button`.

The animations are now completely set up for each of the objects. We still need to finish working on the Animator Controller for the `Chest` and add some more logic for the various Animators, but we are done with the Animation Clips for now.

Building a State Machine and timing the animations

The next thing to do is set up our state machine and write the code that will make the various animations play.

To hook up the various animations and have them play at the correct time, complete the following steps:

1. We'll start by creating the state machine that will work as the logic for our animation sequences. Create a new Animator Controller named `ChestOpeningStateMachine.controller` in `Assets/Animations/Loot Box/Controllers`.
2. Open the `ChestOpeningStateMachine` Animator Controller and create 12 new Empty States. Arrange, name, and transition the States, as shown in the following screenshot:

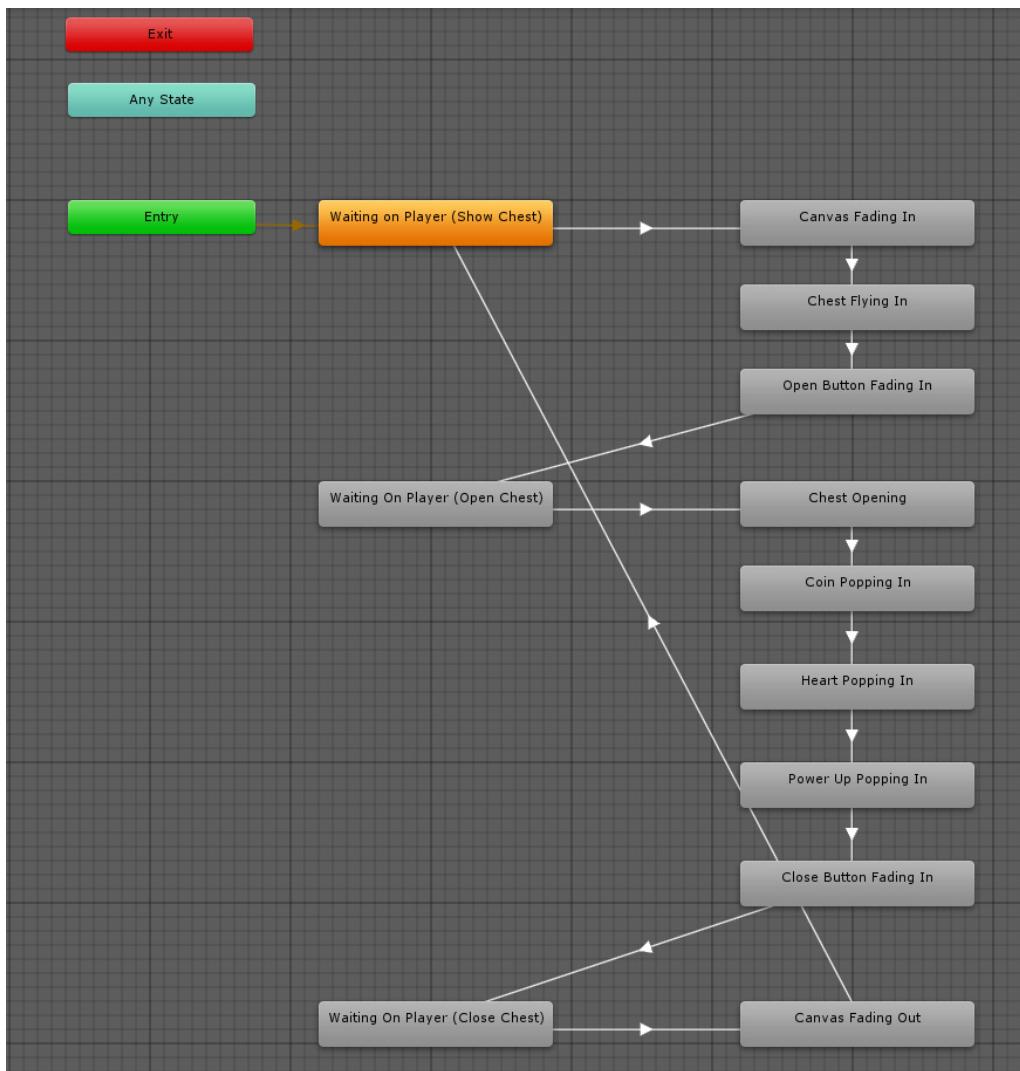


Figure 14.55: The State Machine for the Animation

The state machine shown in the preceding screenshot demonstrates the sequence of events for the animations and interactions of the chest opening. The states labeled **Waiting On Player** will *play* when the game is waiting for the player to press a button to proceed with the animation. The other animations will automatically play based on timed events.

3. The state machine created in the previous steps will not actually contain any animations. It is simply a flow chart describing what is currently happening in the game. We will also use it to send information to the various objects in the scene to let them know what they should or should not be doing. Since we just want this to control the logic of this sequence, and not actually animate anything in the scene, we can add it on an Animator component to any object in the scene. Therefore, let's add it to the Main Camera. Drag the ChestOpeningStateMachine Animator from the project view to the Inspector of the Main Camera.
4. Now, we will need to set the conditions of transition for the various states within the state machine. Create four animation Trigger Parameters, named ShowChest, OpenChest, CloseChest, and AnimationComplete.
5. Now, set Trigger Conditions for each transition, as shown in the following screenshot; with each transition, make sure that you deselect **Has Exit Time** and **Fixed Duration**:

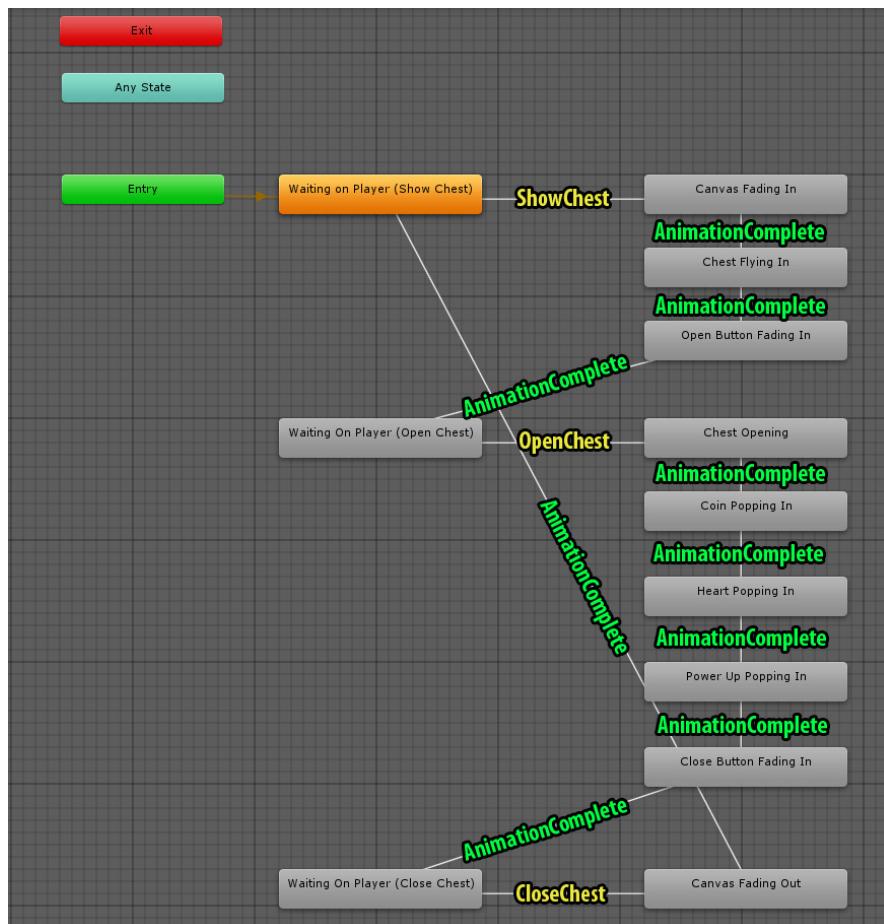


Figure 14.56: The triggers for the various transitions

Note

I highly recommend that you bookmark this image or print out a screenshot of your ChestOpeningStateMachine Animator while working through this example. Having this as a flow chart that you can easily reference while working on this example will make the steps being completed a lot easier to follow.

6. Before we write code, let's set up the Animator for the Chest. Currently, the Animator of the Chest should look something as follows:

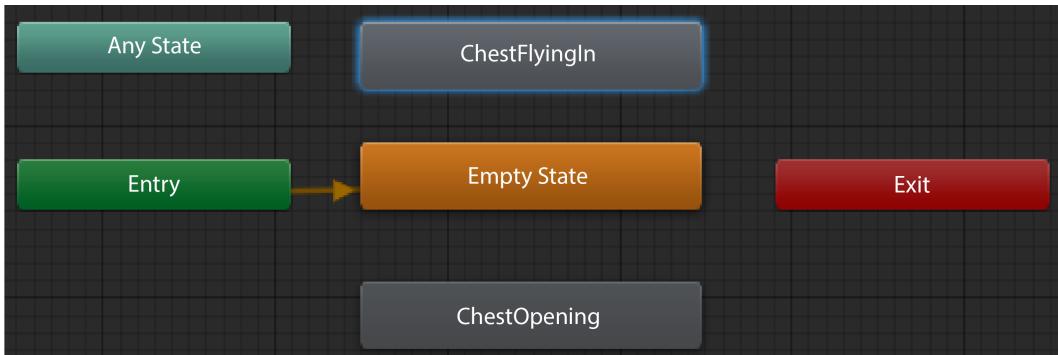


Figure 14.57: The Chest Animator

The Animator only contains the animation states but does not yet indicate how they are all connected. We will need to create transitions and set up the Animation Parameters. Rearrange the states and add transitions to the Animator so it appears as follows:

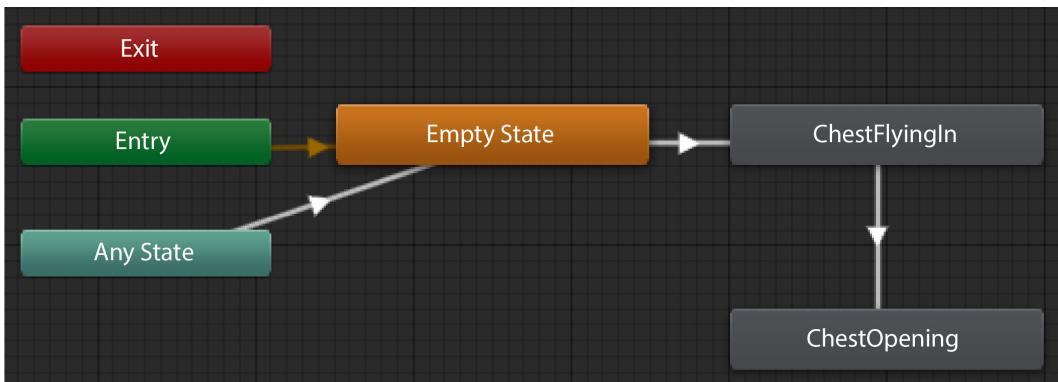


Figure 14.58: The Chest Animator updated

7. Create three animation Trigger Parameters, named ShowChest, OpenChest, and Reset.
8. Now, set Trigger Conditions for each transition, as shown in the following screenshot; with each transition, make sure that you deselect **Has Exit Time** and **Fixed Duration**:

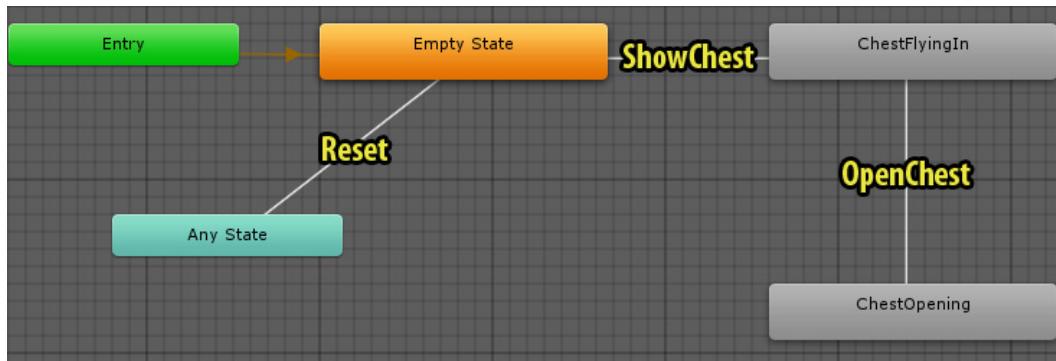


Figure 14.59: The Triggers of the Chest Animator

9. Now that our Animators are all appropriately set, we can begin coding. We will use the `ChestOpeningStateMachine` Animator to make each appropriate animation play when the player clicks on the specified button during the specified state. The `ChestOpeningStateMachine` Animator will then automatically set the appropriate triggers on the Animators on the various objects that appear in the full animated sequence. We need a way to keep track of which items in the scene have Animators that will be controlled by the `ChestOpeningStateMachine` Animator, what their various parameters are, and what conditions need to be met to have those parameters set. We'll keep track of all of this information in a single script. Create a new script called `ChestAnimControls` and save it in `Assets/Scripts/LootBox`.
10. To give us a nice, clean way of keeping track of all of the necessary information, we'll use classes and an enumerated list (I'll explain what these are and why we are using them momentarily). Delete the `Start()` and `Update()` functions from the `ChestAnimControls` class, and write the following code:

```

// The different types of parameters
public enum TypesOfParameters
{
    floatParam,
    intParam,
    boolParam,
    triggerParam
}

// Properties of animation parameters
  
```

```
[System.Serializable]
public class ParameterProperties
{
    public string parameterString; // What string sets it?
    public string whichState; // Name of the state it's called
from, null=not called by the state machine
    public TypesOfParameters parameterType; // What type of
Animator Parameter is it?
    public float floatValue; // Float value required, if float
    public int intValue; // Int value required, if int
    public bool boolValue; // Bool value required, if bool
}

// Make a list of all animatable objects and their parameters
[System.Serializable]
public class AnimatorProperties
{
    public string name; // So the name will appear in the
inspector rather than "Element 0, Element 1, etc"
    public Animator theAnimator; // The animator
    public List<ParameterProperties> animatorParameters; // Its
parameter properties
}
```

You'll note that the preceding code has the following three parts: the `TypesOfParameters` enumerated, the `ParameterProperties` class, and the `AnimatorProperties` class. First, let's look at the `TypesOfParameters` enumerated. This is a list of the types of Animator Parameters that can be used within an Animator. An enumerated list is a custom type that contains a set of constants that are represented with names. A benefit of using an enumerated list is that the list appears as a dropdown menu within the Inspector. Now, let's look at the `ParameterProperties` class. Each object that is animated within the scene has a set of properties related to its Animator Parameters that we need to keep track of. A class can be an effective tool for grouping sets of data together. Therefore, I used a class to group the name of the parameter, which state the parameter will be set in, what type of parameter it is, and its value if it is a `float`, `integer`, or `Boolean` parameter. Note that the type of parameter is defined using the enumerated list, `TypesOfParameters`. This was done because there is a finite and specific set of parameters available for each animator parameter. Now, let's look at the `AnimatorProperties` subclass. For each object in the scene, we will need to keep track of its name, its animator, and all of its parameters along with the conditions in which they are set. Note that the list of parameters and their properties is defined by the `ParameterProperties` class. A big benefit of working with Unity is the ability to assign and view public variables in the Inspector. However, when you create a class within a class, the public variables are not visible within the Inspector unless you place `[System.Serializable]` above the class. This gives the subclass the `Serializable` attribute and allows its public variables to be visible in the Inspector.

Note

I would like to point out that this code allows for the Animators to have Float, Int, and Bool parameters, even though the only parameters we use in any of our Animators are Triggers. I wrote it in this way to make it work more universally so that you can reuse this code for other animations in the future.

11. If you find the code we wrote in the preceding step overwhelming, don't worry—seeing it all listed out in the Inspector will clear it up a bit. All we have done so far is set up a few different groups of data. Now, we will need to create a variable that will use the information. We need a list of all animated items, so add the following code to your script:

```
public List<AnimatorProperties> animatedItems; //all the
    animated items controlled by this state machine
```

12. Attach the `ChestAnimControls` script to the Main Camera by dragging it into its Inspector.
13. In the Inspector of the Main Camera, click on the arrow next to **Animated Items** to expand the list:

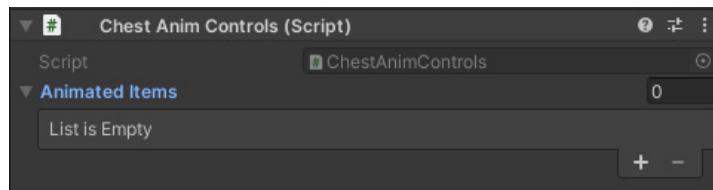


Figure 14.60: The Chest Anim Controls component

14. We have a total of six items that need to have their Animators controlled by this script and the State Machine we created. So, change the List's size to 6. Expand **Animator Items | Element 0** and its **Animator Parameters**:

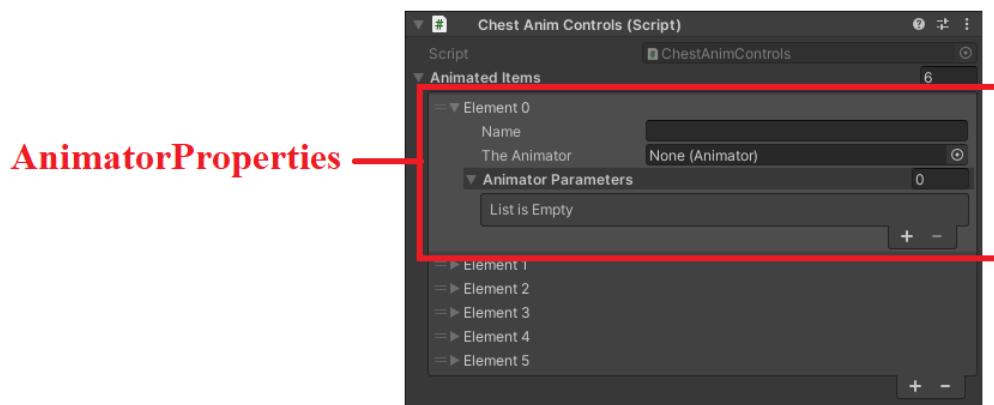


Figure 14.61: The AnimatorProperties of the Chest Anim Controls

Remember that the `animatedItems` variable was a list of `AnimatorProperties`. So, **Element 0** (and all the other **Elements** for that matter) contains all the items that were grouped in the `AnimatorProperties` class.

- The first item we will need to list data for is the `Chest Open Canvas` GameObject. Type `Chest Open Canvas` in the **Name** slot and drag the `Chest Open Canvas` from the Hierarchy into the **The Animator** slot. Once you type `Chest Open Canvas` in the **Name** slot, you'll see that the **Element 0** label is replaced with **Chest Open Canvas**. Whenever you have a list of objects in Unity's Inspector, if the first item in the object is a string, the string will replace the **Element x** label:

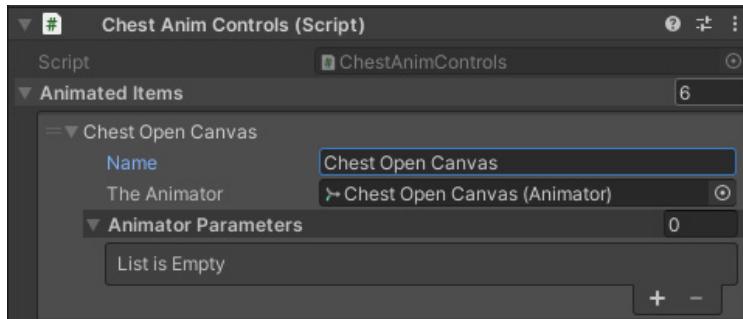


Figure 14.62: The Chest Open Canvas properties

- Now, we will need to list out all the parameters that are used within `Chest Open Canvas`'s Animator and the conditions under which it is set. It has two parameters that we need to list data for, so change the size of **Animator Parameters** to 2. Expand the two resulting **Elements**, as follows:

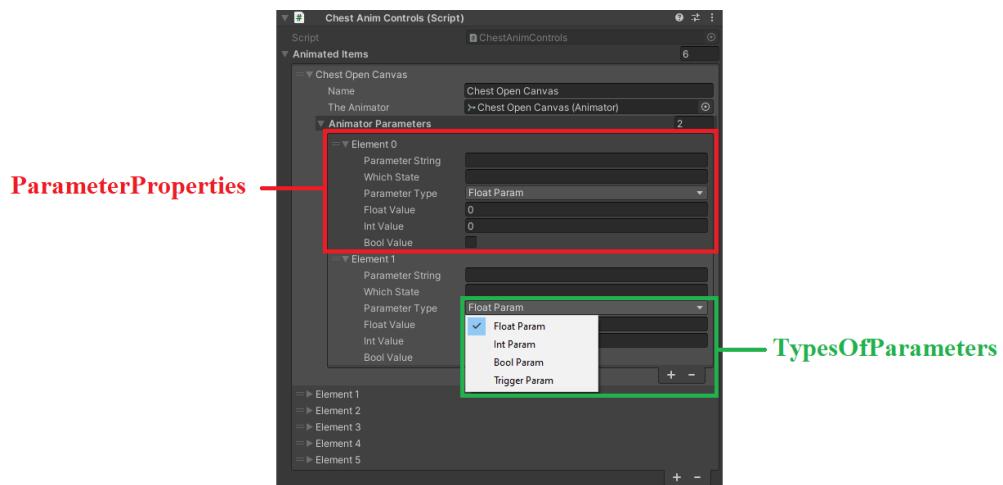


Figure 14.63: The ParameterProperties and TypesOfParameters

Remember that the animatorParameters variable was a list of ParameterProperties. So, the two **Elements** contain all the items that were grouped in the ParameterProperties class. Additionally, within the ParameterProperties class, parameterType was a TypesOfParameters variable. TypesOfParameters was an enumerated list, so any variable of that type will appear as a dropdown menu with the options that appeared within the defined list.

17. We now need to list out each Parameter of the Chest Open Canvas, **Which State** in the ChestOpeningStateMachine will cause the Parameter to be set, and specify its **Parameter Type**:

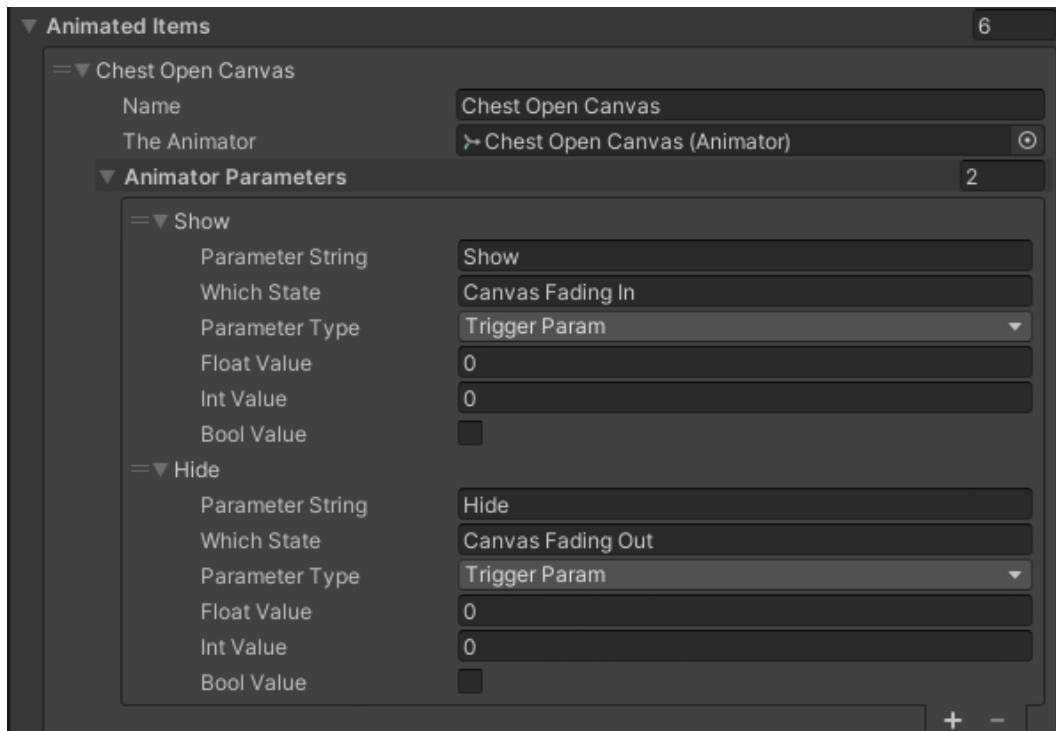


Figure 14.64: The Chest Open Canvas properties

Since each is a Trigger Animator Parameter, we do not have to worry about the values for **Float Value**, **Int Value**, or **Bool Value**.

18. Now, we can fill in the Animator information for the other five animated objects in the same way we filled out the information for the Chest_Open_Canvas. Fill in the information for the animator of Chest as follows:

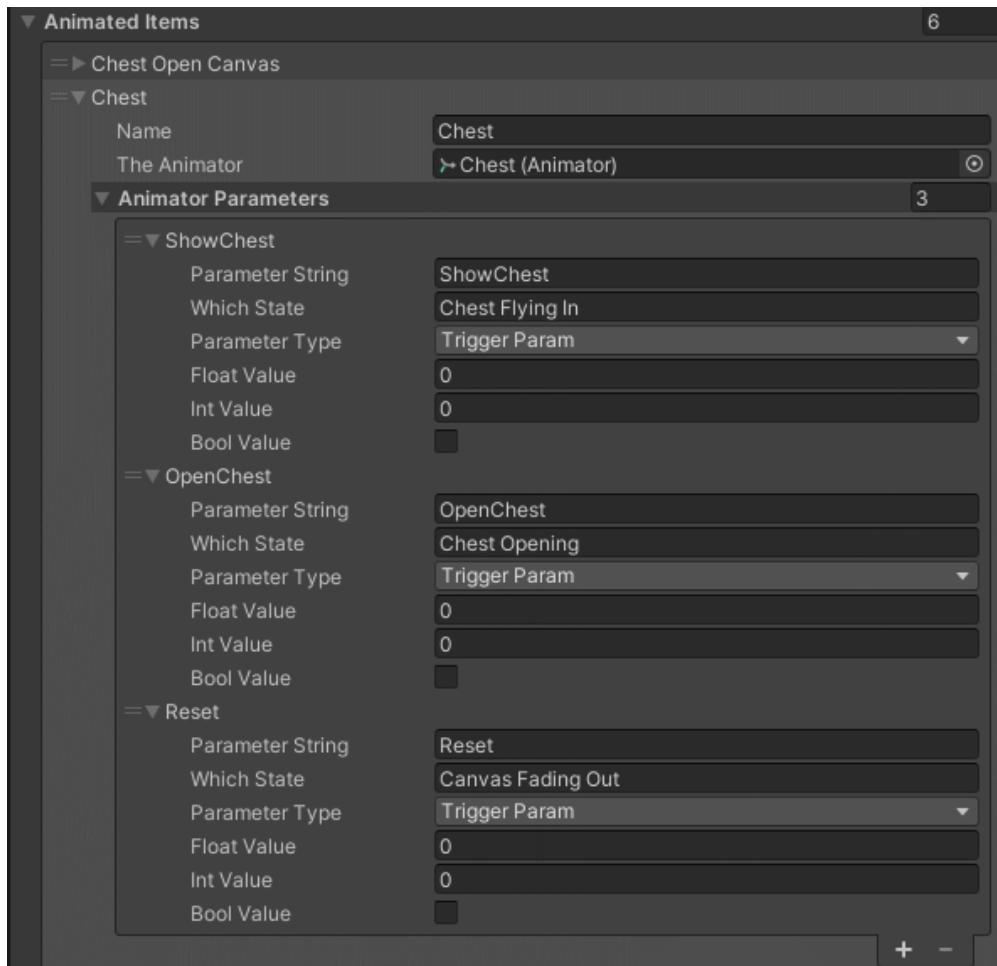


Figure 14.65: The Chest properties

19. Fill in the information for the Animator of Coin as follows:

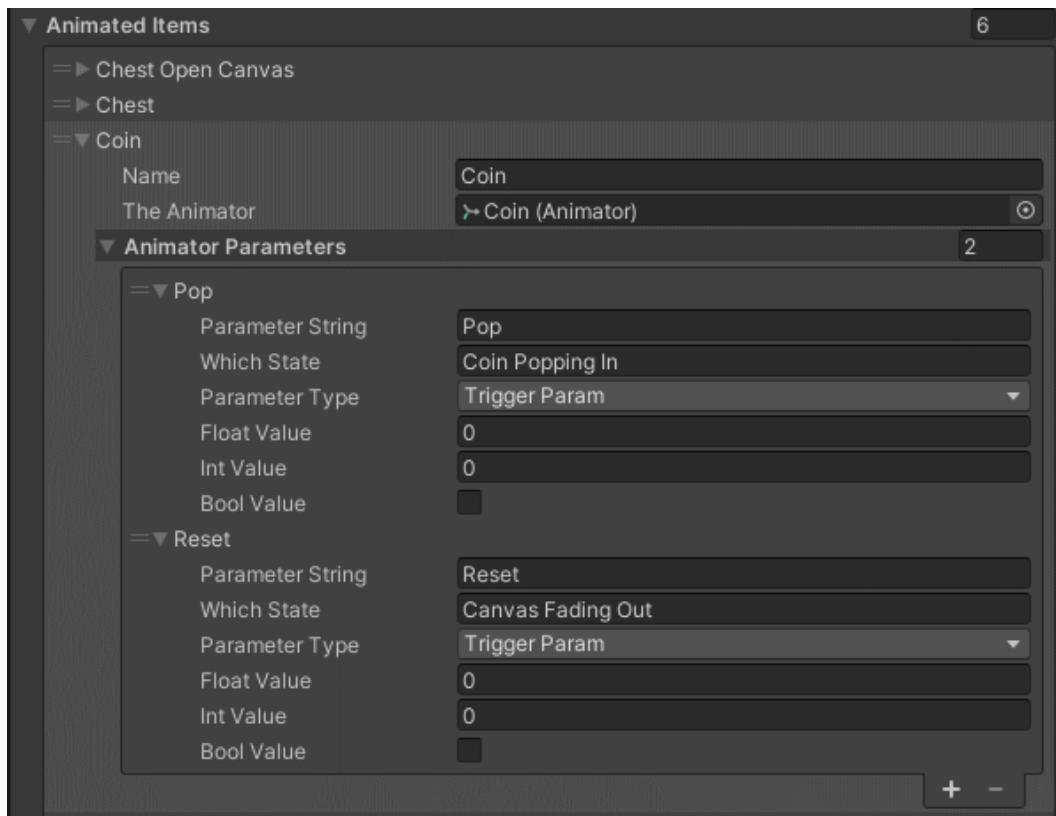


Figure 14.66: The Coin properties

20. Fill in the information for the Animator of Heart as follows:

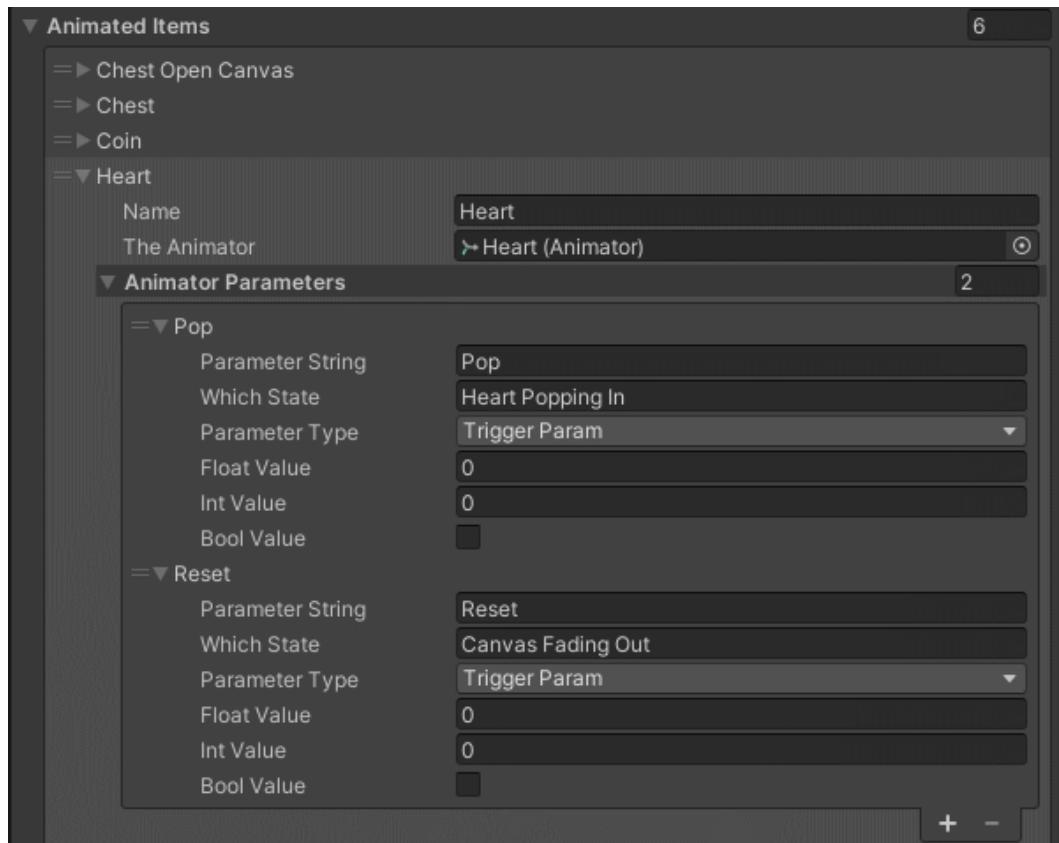


Figure 14.67: The Heart properties

21. Fill in the information for the Animator of PowerUp as follows:

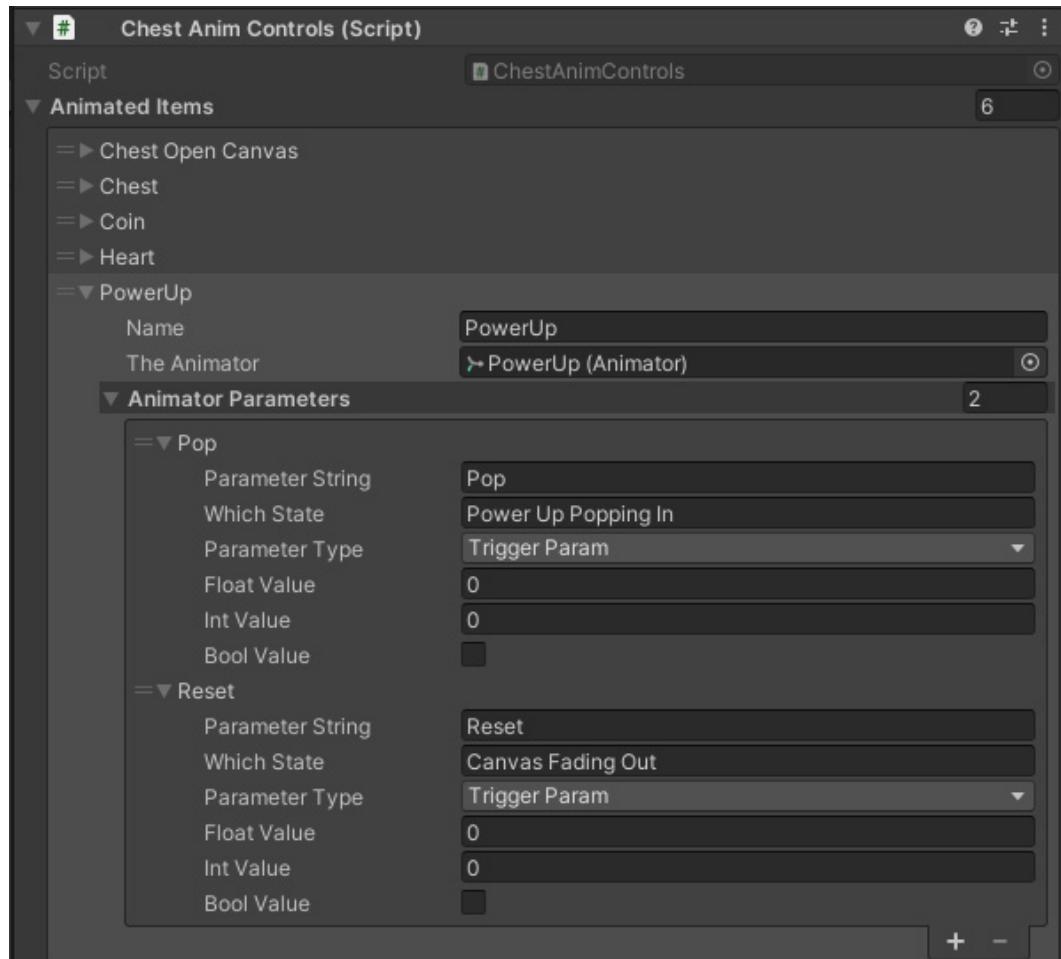


Figure 14.68: The PowerUp properties

22. Fill in the information for the Animator of Button as follows:

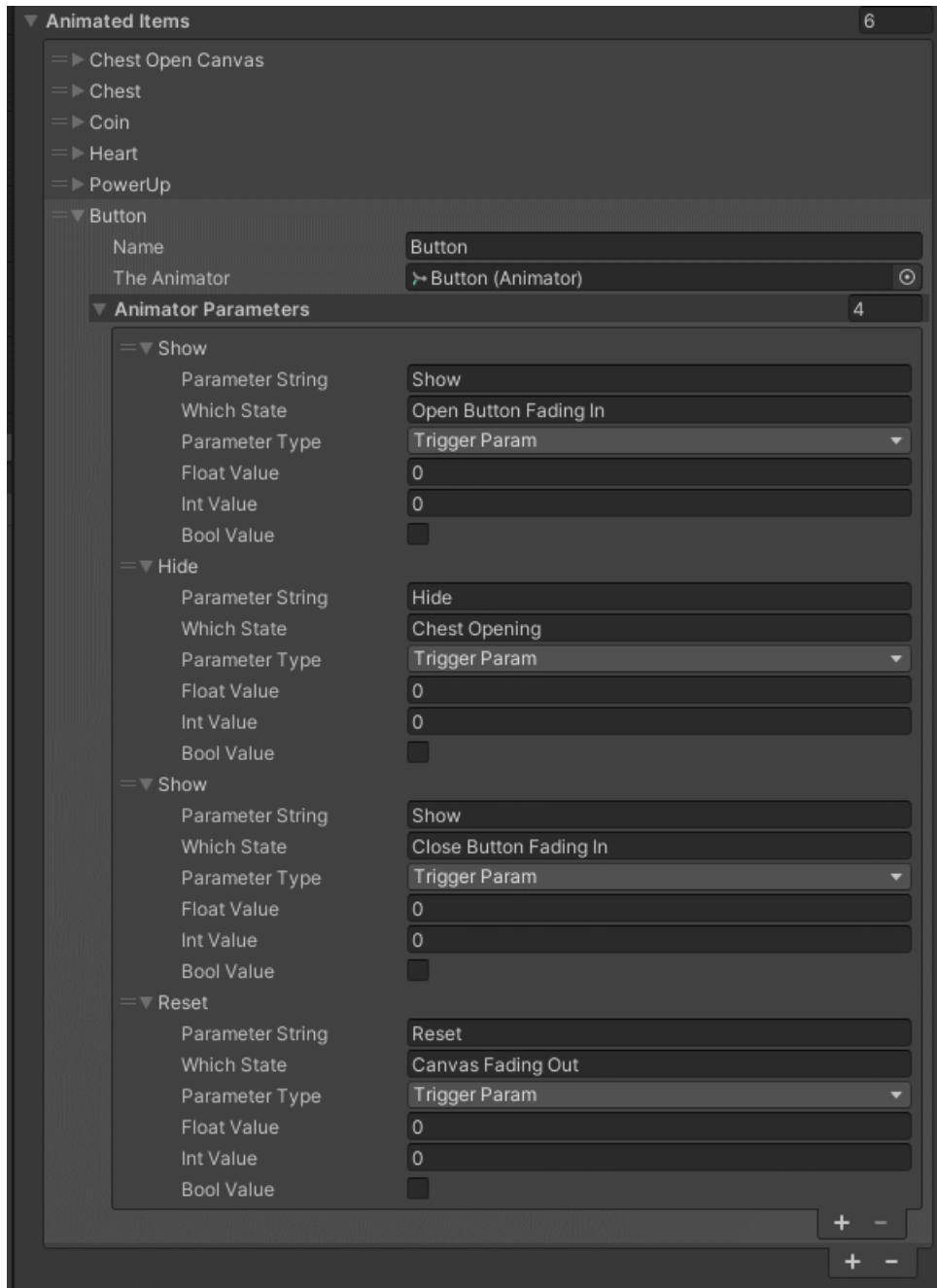


Figure 14.69: The Button properties

23. Now that we have all the appropriate data values for the State Machine initialized and defined, let's actually have the State Machine perform its appropriate logic. First, we will need to create a variable for the Animator. Add the following variable initialization to your script:

```
Animator theStateMachine; //the state machine animator component
```

24. Now, initialize the State Machine's Animator in an Awake () function:

```
void Awake()
{
    theStateMachine = GetComponent<Animator>(); // Get the state
    machine
}
```

25. To have the State Machine automatically set the various parameters of the individual Animators at the appropriate state, we will need to loop through all of the animated items we have listed and each of their listed parameters. If the animated item has a parameter, which is to be set at the current state of the State Machine, we will set it based on the conditions listed. Create the following function to perform that functionality:

```
// Functionality: Check if any of the animations need their
parameters set
// Called from enter state
public void CheckForParameterSet()
{
    // Loop through all of the objects
    foreach (AnimatorProperties animatorProp in animatedItems)
    {
        // Loop through its set of parameters
        foreach (ParameterProperties parameter in animatorProp.
animatorParameters)
        {
            // Find the ones called on the current state
            if (theStateMachine.GetCurrentAnimatorStateInfo(0).
IsName(parameter.whichState))
            {
                // Determine parameter type
                // Float types
                if (parameter.parameterType ==
TypesOfParameters.floatParam)
                {
                    animatorProp.theAnimator.SetFloat(parameter.
parameterString, parameter.floatValue);
                }
                // Int types
                else if (parameter.parameterType ==
TypesOfParameters.intParam)
```

```
        {
            animatorProp.theAnimator.
SetInteger(parameter.parameterString, parameter.intValue);
        }
        // Bool type
        else if (parameter.parameterType ==
TypesOfParameters.boolParam)
        {
            animatorProp.theAnimator.SetBool(parameter.
parameterString, parameter.boolValue);
        }
        // Trigger type
        else
        {
            animatorProp.theAnimator.
SetTrigger(parameter.parameterString);
        }
    }
}
```

The CheckForParameterSet function will determine whether a specified Animator needs a parameter set at the current state of the State Machine. However, this function is not currently called anywhere. We want this function to be called whenever a state in the State machine starts. We can accomplish this with a State Machine Behaviour. Open the ChestOpeningStateMachine Animator and select the **Canvas Fading In** state. In the state's Inspector, click on the **Add Behaviour** button. Select **New Script**, enter ChestStateMachineBehaviour, and click on the **Create and Add** button. A new script named ChestStateMachineBehaviour will be added to the Assets folder. Move it to the Assets/Scripts/LootBox folder and open it.

26. A lot of stuff is included within this State Machine Behaviour. We only need the OnStateEnter function. Adjust the code within the ChestStateMachineBehaviour class to include the following to call the CheckForParameterSet function on the ChestAnimControls script when it starts:

```
ChestAnimControls theControllerScript;

public void Awake()
{
    // Get the script that holds the state machine logic
    theControllerScript = FindObjectOfType<ChestAnimControls>();
}
```

```
// OnStateEnter is called when a transition starts and the state
// machine starts to evaluate this state
public override void OnStateEnter(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex)
{
    theControllerScript.CheckForParameterSet();
}
```

27. We will need this script on every state that does not say **Waiting On Player**. Add **ChestStateMachineBehaviour** to each of the states in the right-hand column (the ones that do not say **Waiting On Player**) by clicking on the **Add Behaviour** button and selecting **ChestStateMachineBehaviour** in their Inspector. The State Machine will now appropriately call each of the individual items' animations when the appropriate states are entered, but we don't have anything that actually controls the flow of the state machine.
28. Right now, it is going to just stay in the **Waiting on Player (Show Chest)** state. We will need to write up some logic to control the flow within the State Machine. Remember that each of the states that say **Waiting on Player** is going to wait for the player to perform some interaction with the game before proceeding to the next state. So, let's start our State Machine logic by making a script that can be used by the Buttons when the player clicks on them. Let's start with the **Button** that is on the **Button Canvas** that says **Start**. Add the following function to your **ChestAnimControls.cs** script:

```
// Called by player interactions (Waiting On Player)
public void PlayerInputTrigger(string triggerString)
{
    theStateMachine.SetTrigger(triggerString);
}
```

This function will trigger the State Machine's trigger parameter specified by the string sent as an argument.

29. We will need to call that function from the **Start** button, so add the following **On Click ()** Event to the **Button** on the **Button Canvas**:

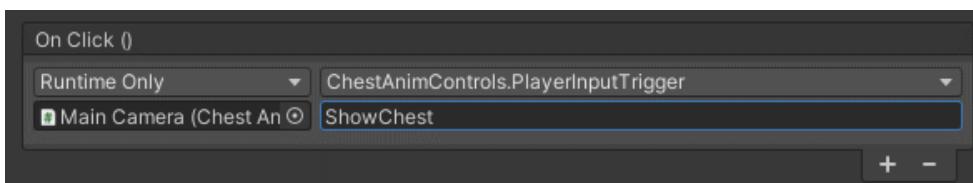


Figure 14.70: The On Click() event of the Button

30. Now, we will need to create some logic that will have the **Button on Chest Open Canvas** transition from the other two **Waiting On Player** states. Create a new script called **OpenCloseButton.cs** in the **Assets/Scripts/LootBox** folder.

31. Add the **UnityEngine.UI** namespace to the **OpenCloseButton** script with the following:

```
using UnityEngine.UI;
```

32. Now, add the following code to the **OpenCloseButton** script:

```
Text buttonText;
Animator chestAnimController;

void Awake()
{
    buttonText = transform.GetComponentInChildren<Text>();
    chestAnimController = Camera.main.GetComponent<Animator>();
}

public void OpenOrClose()
{
    Debug.Log("click");
    if (buttonText.text == "Open")
    {
        chestAnimController.SetTrigger("OpenChest");
        SetText("Close");
    }
    else
    {
        chestAnimController.SetTrigger("CloseChest");
        SetText("Open");
    }
}

public void SetText(string setTextTo)
{
    buttonText.text = setTextTo;
}
```

The **OpenOrClose()** function will be called by the button's **On Click ()** Event. The **Button on Chest Open Canvas** will be used to open and close the chest. It will set the appropriate trigger based on the current text written on the button and will change its text to "Open" or "Close" with the **SetText ()** function.

33. Add the OpenCloseButton script as a component to the Button on Chest Open Canvas.
34. Now, add the following **On Click ()** Event to the Button on Chest Open Canvas:

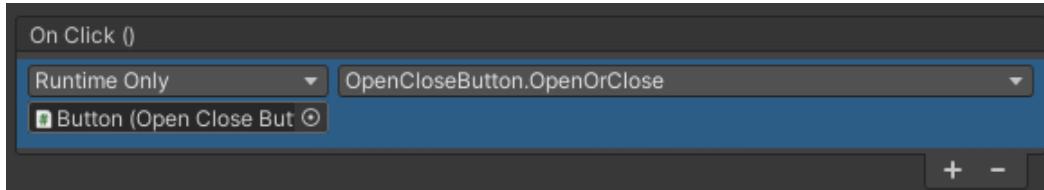


Figure 14.71: The On Click() property of the Button

35. If you play the game now and click on the **Start** button, all that happens is that the Canvas fades in. This is because we still haven't done anything to set the `AnimationComplete` trigger in the `ChestOpeningStateMachine` Animator. We will need another script to set this trigger. Create a new script called `AnimationControls` in the `Assets/Scripts/LootBox` folder.
36. Adjust the code within the `AnimationControls` class, as follows:

```
Animator chestAnimController;

void Awake()
{
    chestAnimController = Camera.main.GetComponent<Animator>();
}

// Call as an animation event on the last frame of animations
public void ProceedStateMachine()
{
    chestAnimController.SetTrigger("AnimationComplete");
}
```

37. The preceding code simply sets the `AnimationComplete` trigger with the `ProceedStateMachine()` function.

This trigger is used to allow each of the individual animations to fully play before the next state is started. To make sure that the animation trigger isn't set until the entire animation has played, we'll use Animation Events to call the `ProceedStateMachine()` function at the appropriate time within the necessary animations.

When you use an Animation Event, the function you want to call must be on a script attached to the same object as the animation. We want the function to be called at the end of animations on Chest Open Canvas, Chest, Coin, Heart, PowerUp, and the Button on Chest Open Canvas. Therefore, add the AnimationControls script as a component to each of them.

38. Now, we will need to add the `ProceedStateMachine()` function as an Animation Event to the various animations. Select Chest Open Canvas and view its `CanvasGroupFadeIn` animation. On its last animation frame, right-click on the top dark gray area of the timeline and select **Add Animation Event**. In the Inspector, select `ProceedStateMachine()` from the dropdown menu. It will be the very last function in the list. You should now have a white flag above the last keyframe that says **ProceedStateMachine** when you hover over it:

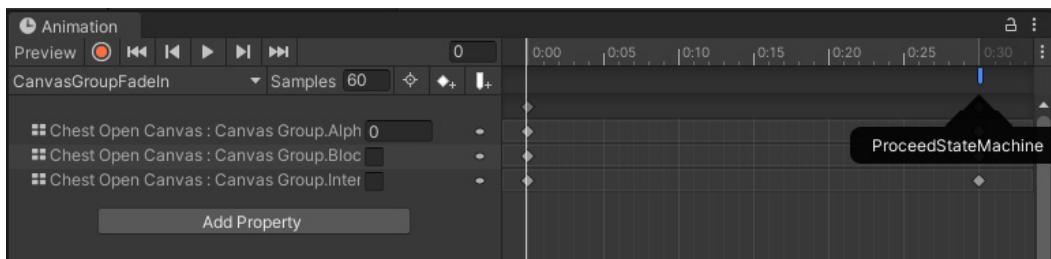


Figure 14.72: Adding the `ProceedStateMachine` Event to the animation

39. If you play the game now and click on the **Start** button, the Canvas will fade in and the chest will fly in. The State Machine will stay on the `ChestFlyingIn` animation. To have the animation sequence finish out, add the `ProceedStateMachine()` function as an Animation Event to each of the following animations: `ChestFlyingIn`, `ChestOpening`, `CoinPopping`, `HeartPopping`, and `PowerUpPop`.
40. Playing the game now almost works the way it should. There's a bit of an issue with the timing of the items popping out of the chest when you replay the animation sequence. Currently, there is a bit of a problem with the `AnimationComplete` trigger. We need that trigger to definitely be unset whenever any of the **Waiting On Player** states start. Otherwise, it will be set to `true` when the State Machine restarts, causing some timing issues. To fix this, we need one more State Machine Behaviour. Select the **Waiting on Player (Show Chest)** state and create and add a new State Machine Behaviour called `ResetTriggers` in its Inspector. Remember that whenever you create a new State Machine Behaviour, it is added to the Asset folder, so move it to the `Asset/Scripts/LootBox` folder.

41. We want the `AnimationComplete` Trigger Parameter to reset whenever a **Waiting On Player** state starts, so change the code in the `ResetTriggers` class, as follows:

```
// OnStateEnter is called when a transition starts and the state  
// machine starts to evaluate this state  
override public void OnStateEnter(Animator animator,  
AnimatorStateInfo stateInfo, int layerIndex)  
{  
    animator.ResetTrigger("AnimationComplete");  
}
```

42. Now, add the `ResetTriggers` State Machine Behaviour to all three of the **Waiting on Player** states.

Playing the game now should have the animation sequence firing correctly, with everything but the particle system, which will be covered in the next chapter.

Summary

Animating UI elements is not significantly different from animating any other 2D object in Unity. Therefore, this chapter offered a brief overview of animation. This chapter also offered an example of the workflow for creating complex animations utilizing a State Machine and Animation Events. There's a lot we can do with animations, and the examples in this chapter showed you many of the techniques you can use while animating UI. Hopefully, these examples will provide you with enough variation of technique that you can determine how to make your own animations in the future.

In the next chapter, we will discuss how to render particle effects in front of the UI.

15

Particles in the UI

Particle effects are a fun and attractive way to add “juice” to your game. The Particle system within the Unity Engine provides you with the tools needed to make all sorts of interesting effects like sparkles, smoke, fire, and so on. This chapter will discuss how you can use particle effects within your UI.

In this chapter, we will discuss the following topics:

- Methods for displaying particle effects in your UI
- Adding flying stars to our loot box animation

This book is about UI, not particle effects. Due to the complex nature of particle effects, I will not be covering all the settings involved and the various intricacies of using particle effects. I will walk you through the steps to create a single particle effect but will not dive further into the process of creating particle effects.

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2015>

Particles in the UI

Using particles in UI is a hot topic. It seems like nearly every mobile game with loot boxes uses particles, but there is no standardized way to implement them. The problem with trying to use particles in the UI is that particles render behind UI on Canvases that have their **Render Mode** set to **Screen Space - Overlay**.



Figure 15.1: Particles rendering behind a Canvas

The preceding screenshot shows two UI Canvases from our working examples and a particle system (the white dots). The pink background is on the **Background Canvas**, which renders with **Screen Space - Camera**. The circular meter and the Panel with the cat are on the **HUD Canvas**, which renders with **Screen Space - Overlay**. The particles are rendering in front of the Background Canvas and behind the **HUD Canvas**. However, I want the particles to be displayed also in front of the **HUD Canvas**.

There are a few solutions to this problem. My two preferred solutions are either of the following:

- Change the **Render Mode** on **HUD Canvas** to **Screen Space - Camera** and adjust the sorting order of it and the particles to make the particles appear in front
- Use a second Camera and a Render Texture to display the particles on a Raw Image in the Canvas

There are benefits and downfalls to both methods. The first method is by far the easiest. It allows you to view particles in front of your UI with only one or two modifications to your scene. However, using **Screen Space - Camera** for your UI's **Render Mode** may not be practical for your project. If you edit the properties of the Camera in your game, the properties of the UI will be changed. Additionally, changing the **Render Mode** of your Canvas after you have already set everything up can cause your UI to stop displaying the way you initially intended.

The second method isn't terribly complicated to implement but does require more work than the first. Its main benefit is that you can render particles in front of UI on Canvases that render with **Screen Space - Overlay**. Its main downfall, other than needing more work to set up, is that you may have to make some complicated decisions about what your two cameras are going to render, and may slightly affect performance. An example covering this method is discussed in the *Examples* section of this chapter.

There are other solutions to this problem, each more complicated (or costly) than the next, and what you choose to do depends on your project. Some projects forego particles entirely and pre-render their particles as sprite sheets using software, such as After Effects. Some projects use assets available in the Asset Store, and others handle everything entirely with scripts and shaders. Although I can't foresee any reason why the second solution I proposed will not work for your project, it's certainly possible that your project has a fringe case I am not considering. Hopefully, if that is the case, you will be able to modify my solution with minimal effort to work on your project.

My best advice to you would be to decide early whether you will use particles in your UI. If you know you are going to use them, plan ahead with your UI layouts and camera setups. Also, if you want to use the first method, do it.

It's really too bad that there is no standard method for this implemented by Unity. I assume that one day there will be a pre-built UI Particle object that will make the process both simple and performant.

Examples

For the examples in this chapter, we will add to the animations created in *Chapter 14* to add a particle effect that occurs when the loot box opens.

Creating a Particle System that displays in the UI

Let's create a particle system that will pop when the chest opens up. As stated earlier in this chapter, my two preferred ways of displaying particles in front of UI are to either use **Screen Space - Camera** as the Canvas **Render Mode** or to use a Render Texture. Since the second option is more complicated, it merits an example. You'll notice that our Canvases all have their **Render Modes** set to **Screen Space - Overlay**, so using a **Render Texture** is the best method for the way the project is currently set up.

We will create a particle system that is rendered to a texture via a second camera and then have that texture displayed on a **Raw Image** within the UI.

To create a particle system, complete the following steps. We will work on displaying it in the UI in the next section:

1. The first thing we will need to do is create a material that will be used for the particle. Create a new folder in the Assets folder named **Textures and Materials**.
2. Right-click within the new folder and select **Create | Material**. Name the new material **StarsMaterial**.
3. Set the **StarsMaterial Shader** to **Unlit/Transparent**.
4. Drag the **starIcon** sprite into its texture slot.
5. To display the particles in front of the UI objects, we will need a second Camera. Duplicate the **Main Camera** using **Ctrl + D** and rename the duplicate **UI Particles Camera**.
6. You can only have one Audio Listener in the scene, so delete the **Audio Listener** component on the new camera.
7. Remove the **Animator** and **Chest Anim Controls** components.
8. You also won't want any code later to think this might possibly be the **Main Camera**, so change the tag from **MainCamera** to **Untagged**.

9. This camera will be used only to display the particle pop we're going to make, so we might as well make the particle system a child of this camera by right-clicking on the UI **Particles Camera** and selecting **Effects | Particle System**.



Figure 15.2: Creating a Particle System as a child of UI Particle Camera

Note

Since this book isn't about Particle Systems but about UI, we will not spend time going over every property of Particle Systems. Luckily, most are somewhat self-explanatory, and fiddling with the various properties lets you see what they can do. Therefore, rather than going through each property of the Particle System, I will simply provide screenshots of the necessary properties.

10. Scroll down to the bottom of the **Particle System** component and expand the **Renderer** property by clicking on it.
11. Assign the **StarsMaterial** material to the **Material** property. This will now make stars appear in your particle effect.

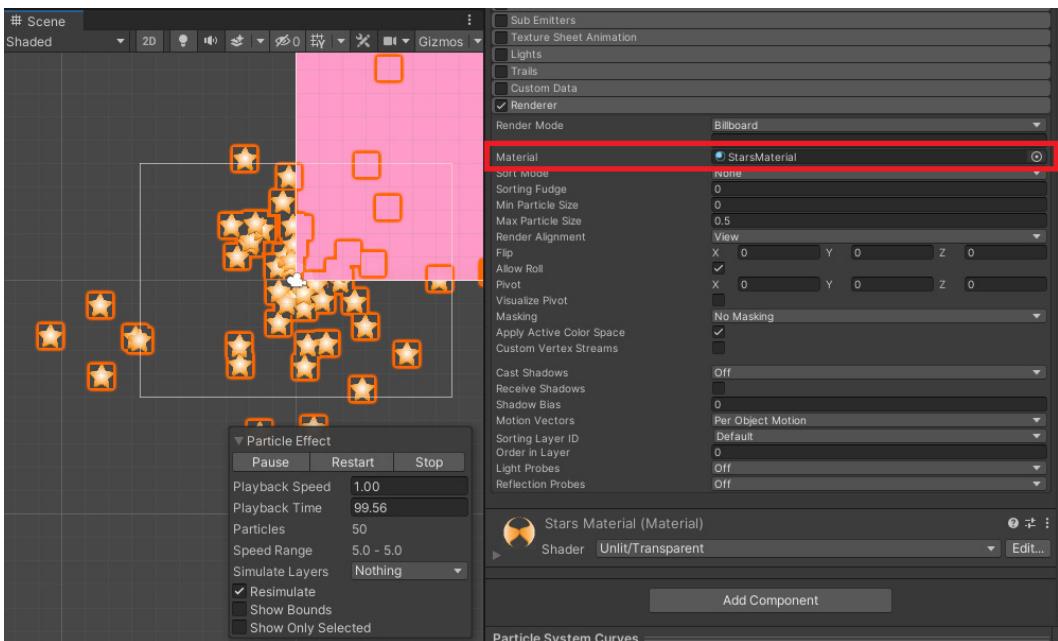


Figure 15.3: Adding StarsMaterial material to the Renderer Material property

12. Change the Transform **Rotation X** to **-90**. The particles should now shoot upward.

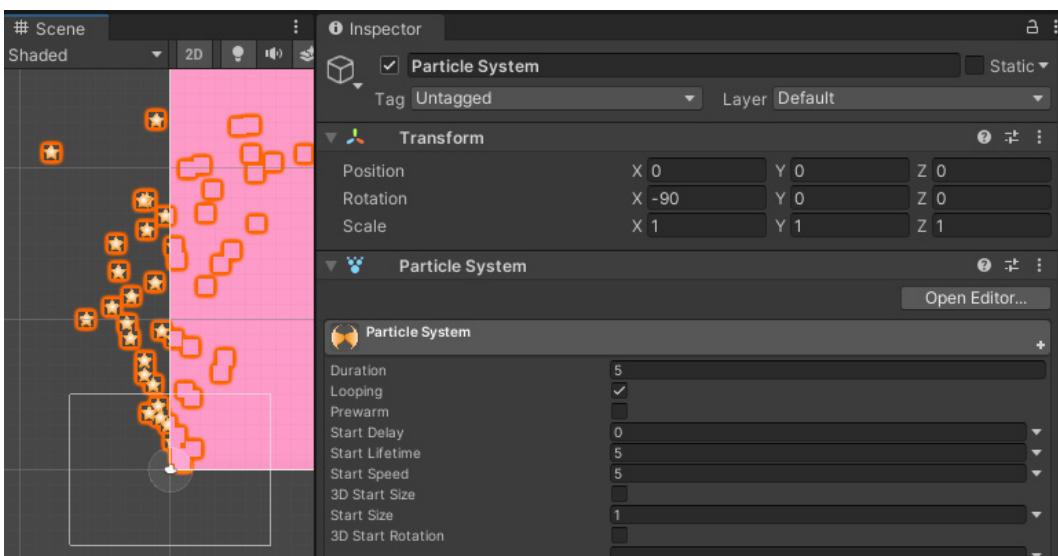


Figure 15.4: Changing the Transform Rotation

13. Change the **Position Z** value to 10.
14. Change the **Duration**, **Start Lifetime**, and **Start Speed** properties of the Particle System as follows:

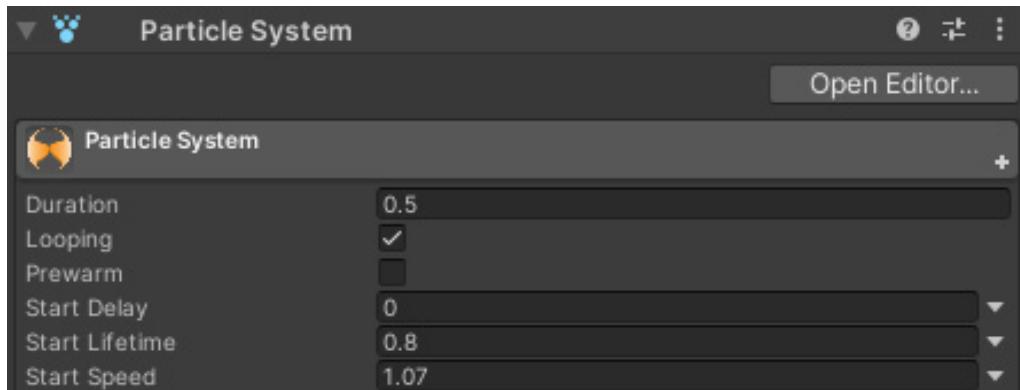


Figure 15.5: Updating some of the Particle System settings

15. Select the dropdown on **Start Size** and select **Random Between Two Constants**. Set the values to 0 and 0.5.
16. Select the dropdown on **Start Rotation** and select **Random Between Two Constants**. Set the values to -45 and 45.

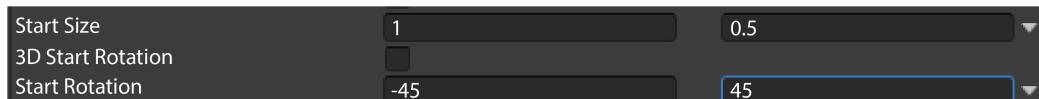


Figure 15.6: Updating Start Size and Start Rotation

17. Change **Flip Rotation** to 1 and **Gravity Modifier** to 0.5.



Figure 15.7: Updating the settings of the particle system

18. Expand the **Emission** property and add a **Burst** with the plus sign.

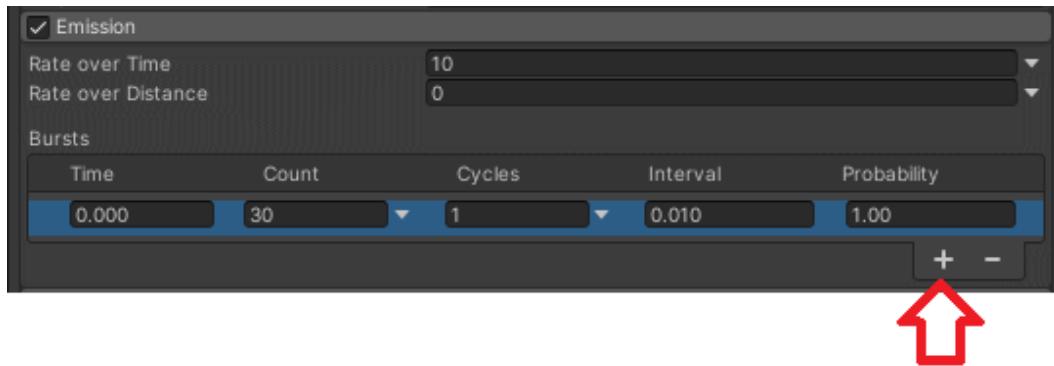


Figure 15.8: Adding a burst

19. Expand the **Shape** property and select the **Hemisphere** from the **Shape** dropdown.

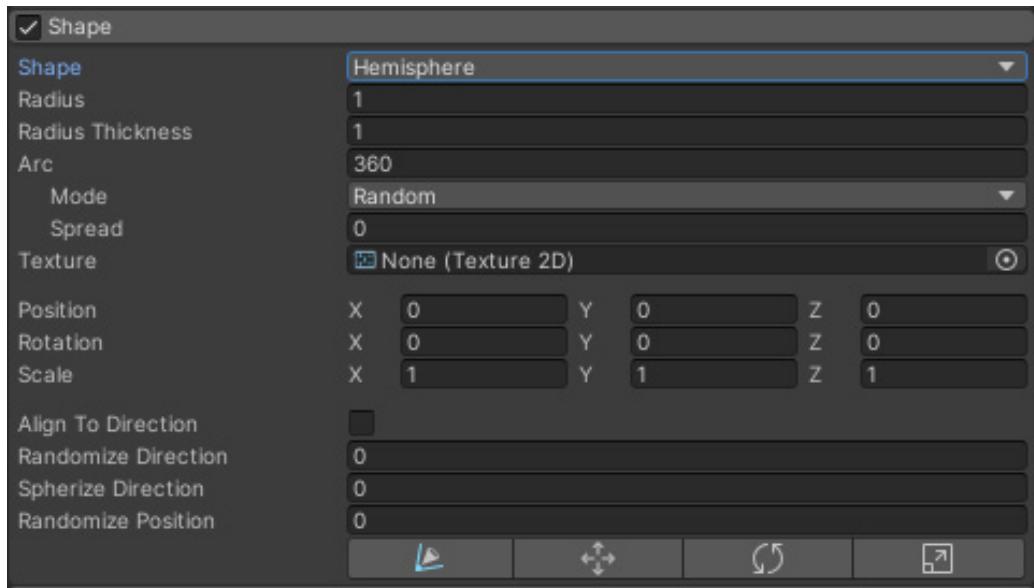


Figure 15.9: Changing the shape

20. Select the **Size over Lifetime** property.



Figure 15.10: The Size over Lifetime property

21. Select the **Rotation by Speed** property.

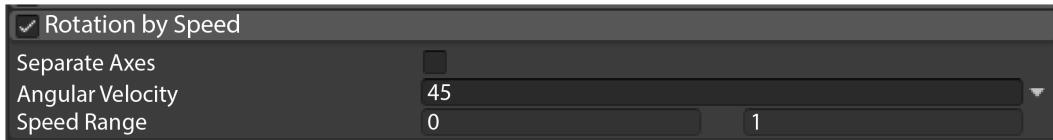


Figure 15.11: The Rotation by Speed property

And now, we're done with the properties of the Particle System. Eventually, we will select **Looping** and **Play on Awake**, but for now, we will leave them deselected so that we can see the particle system constantly playing when the game is playing.

22. We want to make sure that **UI Particles Camera** displays only **Particle System** and **Main Camera** displays everything but **Particle System**. We will accomplish this with **Layers**.

Select the **Layers** dropdown menu and select **Add Layer....**

23. Add a new **User Layer** named **UI Particles**.
24. Assign **UI Particles** to the **Layer** property of **Particle System**.
25. Now, we need to specify to each of the cameras what they will be displaying using their **Culling Mask** property.

Set the **Culling Mask** property of **UI Particles Camera** to only display **UI Particles** and the **Culling Mask** property of **Main Camera** to exclude **UI Particles**.

26. Now, let's have **UI Particles Camera** render to a texture. Within the **Textures and Materials** folder, right-click and select **Create | Render Texture**, and name it **StarPopRenderTexture**.
27. Change its **Size** value to **512 x 512**.
28. Assign the **StarPopRenderTexture** texture to the **Target Texture** property of **UI Particles Camera's Camera** component.

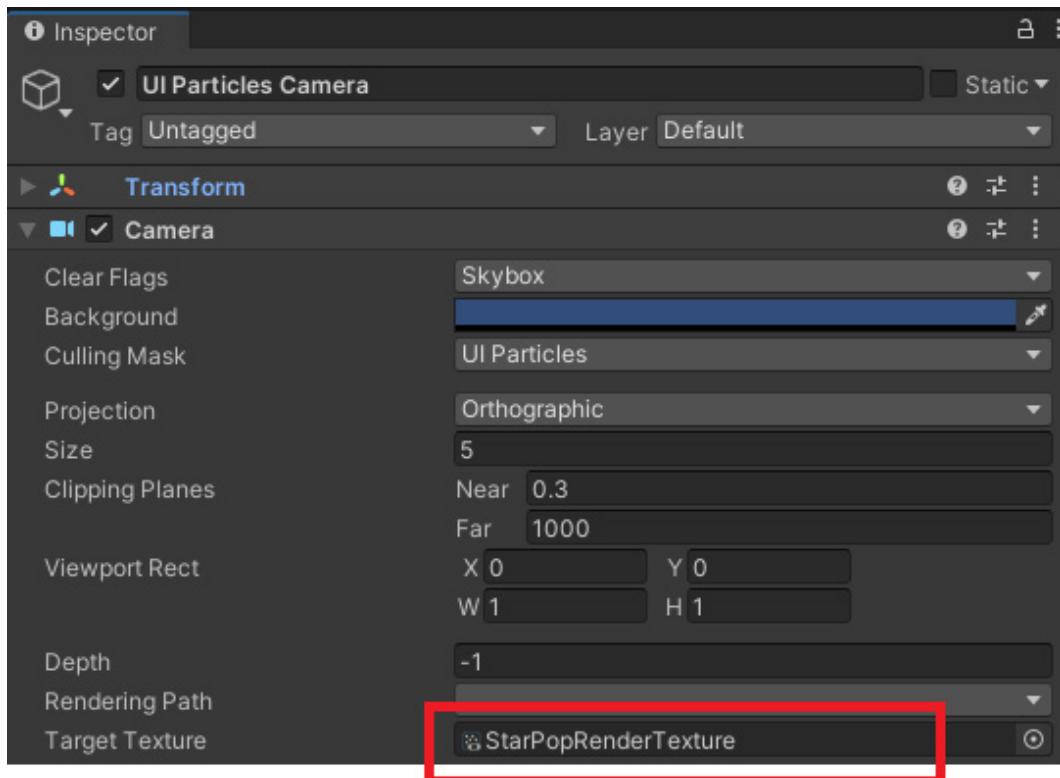


Figure 15.12: Assigning the Target Texture

29. The only thing left to do is have the render texture display in the UI.

Create a new UI Canvas with **Create | UI | Canvas**. Rename the Canvas to **Particle Canvas**.

30. With **Particle Canvas** selected, select **Create | UI | Raw Image**. Rename it **Particle Renderer**.
31. Change the **Width** and **Height** values of **Particle Renderer** to 512 and 512, respectively, to match the properties of **StarPopRenderTexture**.
32. Assign **StarPopRenderTexture** to the **Texture** property of the **Raw Image** component of **Particle Renderer**.

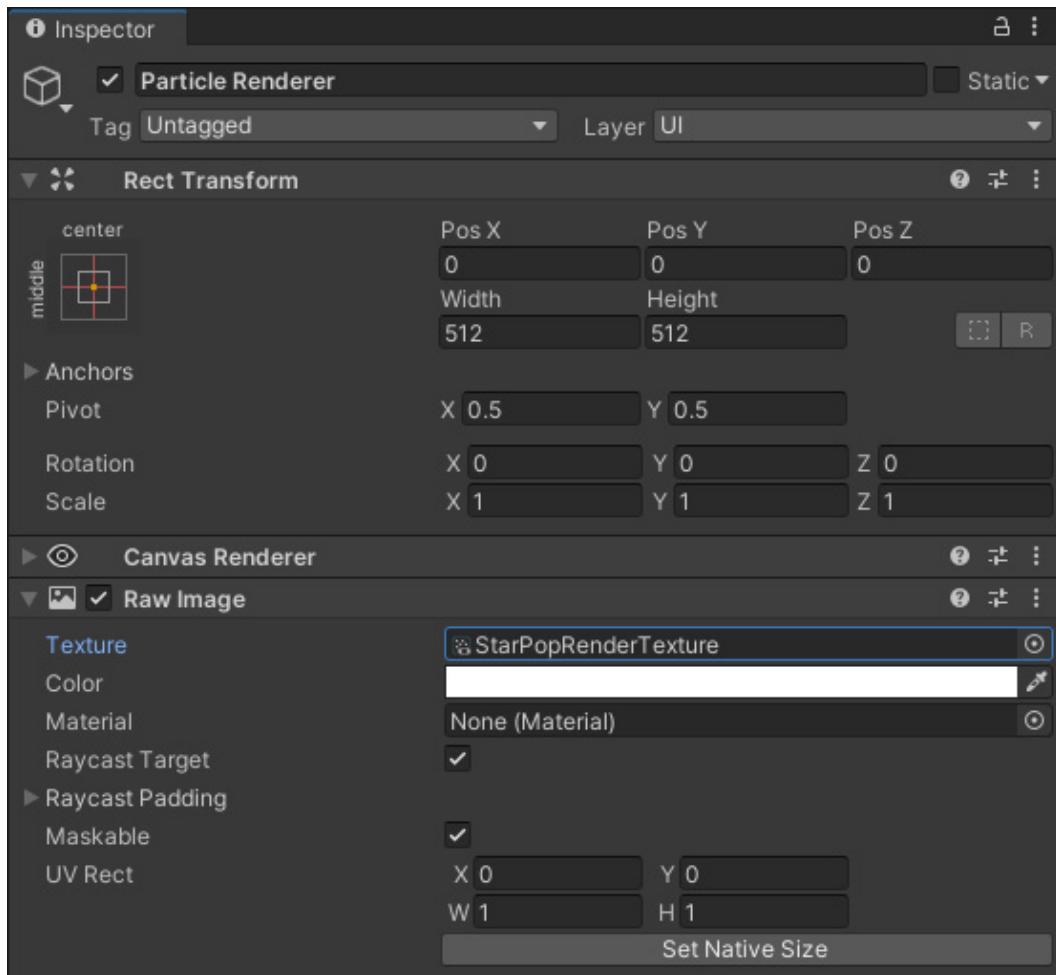


Figure 15.13: The Render Texture assigned to the Texture property

33. We don't want this image to block our mouse clicks, so deselect **Raycast Target** from the **Raw Image** component.
34. Now, we just need to make sure that **Particle Canvas** displays in front of the other two Canvases. Set the **Canvas Sort Order** value to 2 on **Particle Canvas**'s **Canvas** component.
35. Play the game now, and you should now see the particles displaying the scene.
36. We set the Particle System to be constantly playing, so deselect **Looping** and **Play on Awake** to reset the values to what they should be.

That's it for setting up a particle system that displays in a UI.

Timing the Particle System to play in our loot box animation

Now that the particle system is set to display in front of the UI, we can set up the logic to have the animation trigger in the correct order. To do that, go through the following steps:

1. We want the particle system to play when the chest opens, so we have to write some code to control its behavior. Create a new script called `PlayParticles` in the `Assets/Scripts` folder.
2. Edit the `PlayParticles` class to have the following code:

```
public ParticleSystem stars;

void PlayTheParticles() {
    if (!stars.isPlaying) {
        stars.Play();
    }
}
```

3. All this code does is check whether the particle system is currently playing with the `PlayTheParticles()` function. If it is not playing, it plays when the function runs.
4. We'll have this function triggered via an Animation Event on the Chest. So, add the script to the Chest as a component.
5. Assign `Particle System` from the **Hierarchy** to the **Stars** slot.
6. Add the `PlayTheParticles` function as an Animation Event on the very first frame of the `ChestOpening` animation.



Figure 15.14: The particle Animation Event

Playing the game now should result in all animations playing at the appropriate times and the particle system displaying when the chest opens. And that concludes the loot box animation tutorial!

Summary

At first glance, it would appear that you cannot use Unity particles within Unity UI that is set to Screen Space - Overlay. However, there are a few simple tricks that let you implement particle effects in the UI and have the particles render in front of them.

In the next chapter, we will discuss how to use the World Space Canvas Render Mode to have UI elements appear directly in your Unity scene rather than on the screen.

16

Utilizing World Space UI

In *Chapter 6*, we discussed the three different Render Modes you can assign to a Canvas. We've used Screen Space-Overlay and Screen Space-Camera but haven't used World Space yet. As described in *Chapter 2*, UI rendered in World Space is placed directly in the scene. We've already discussed the properties of World Space Canvas rendering, so this chapter will just look at when to use it and examples of implementation.

In this chapter, we will discuss the following topics:

- When to use World Space UI
- General techniques to consider when working with World Space UI
- Using World Space Canvases in a 2D game to create status indicators positioned relative to characters
- Using World Space Canvases to create health bars that hover over enemies' heads in a 3D game

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2016>

When to use World Space UI

There are many reasons you may want to use a World Space Canvas. The most common reasons for using this render mode are the following:

- To have better control of individual UI objects' positions in relation to objects in the scene
- To rotate or curve UI elements

For example, the game *Mojikara: Japanese Trainer* uses World Space Canvases to have rotated Panels and keep UI objects, such as Text, attached to 3D objects. As you can see from the following screenshot, the Panel on the left is rotated just slightly in 3D space, because it is on a World Space Canvas:



Figure 16.1: Mojikara: Japanese Trainer (image provided by Lisa Walkosz-Migliacio, Intropy Games)

Another example of rotated UI can be found in the game *Cludbase Prime*, as shown in the following screenshot. It also used World Space rendering to create indicators that hover over objects and characters.



Figure 16.2: Cludbase Prime (image provided by Tyrus Peace, Floating Island Games)

All the UI in *Cludbase Prime* was done on World Space Canvases. This allowed the developer to create cool curving UI, as follows:



Figure 16.3: Cludbase Prime (image provided by Tyrus Peace, Floating Island Games)

Here, you can see how the UI looks in the Editor versus how it looks to the player. This gives a nice peek at how the UI was built:



Figure 16.4: Cludbase Prime (image provided by Tyrus Peace, Floating Island Games)

I recommend checking out the following site to see more ways in which *Cloudbase Prime* has implemented World Space UI, as they are truly beautiful: <https://imgur.com/a/hxNgL>.

Another common usage of World Space UI is simulating computer screens and monitors within a scene. For example, I built the following UI for a friend's VR game named *Cloud Rise*. The monitor was simulated by placing a World Space Canvas right on top of the in-game screen. I was then able to easily anchor and animate the UI; in the same way, I rendered the UI in Screen Space.



Figure 16.5: Cloud Rise (image provided by Meredith Wilson, Bedhouse Games)

In general, the interactive UIs of VR games are on World Space Canvases since the player cannot interact with *the screen*. Common usages of VR UI are flat floating Panels or wrapping Panels.

Hovering indicators are by far the most common use of World Space UI; they are specifically used for health bars over the heads of in-game characters, as shown in the following screenshot of *Iris Burning*:

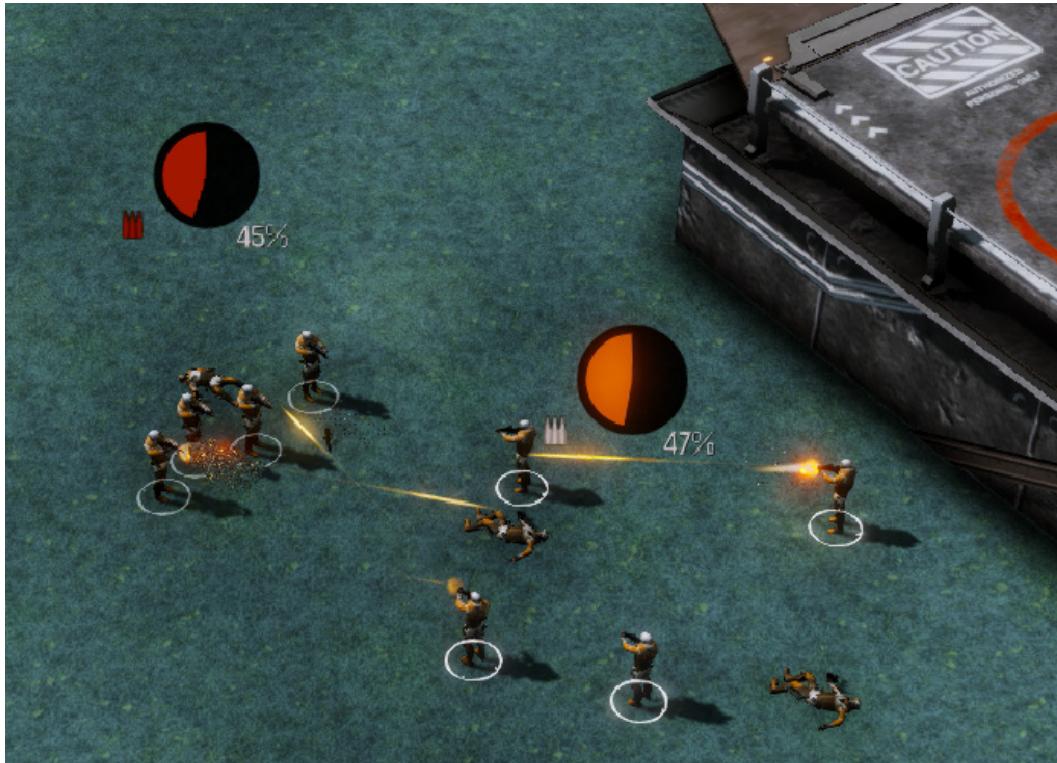


Figure 16.6: Iris Burning (image provided by William Preston, DCM Studios)

Most people think of 3D games when they think of World Space UI because they think of UI that appears *far away*, but it is commonly used in 2D games as well! Management and RTS games use UI quite frequently to create buttons and progress bars, and other UI elements maintain their position with the object they interact with. The World Space UI can be on one Canvas that encompasses all items on the screen, with individual UI items matching the 2D World Space coordinates of the items they represent, or they can be on individual Canvases of their respective items. We will cover how to create a 2D game using World Space UI in an example at the end of the chapter.

Now, let's explore how to use World Space UI.

Appropriately scaling text in the Canvas

Whenever a Canvas is created, it is initialized with **Screen Space - Overlay** as its **Render Mode**. Therefore, when you change the **Render Mode** property to **World Space**, the Canvas will be huge in your scene.

When you scale down the Canvas to the appropriate size in the scene, the text will likely be super blurry or not visible at all. Let's say we created the following Canvas in **Screen Space - Overlay** but decided to put it in **World Space**:

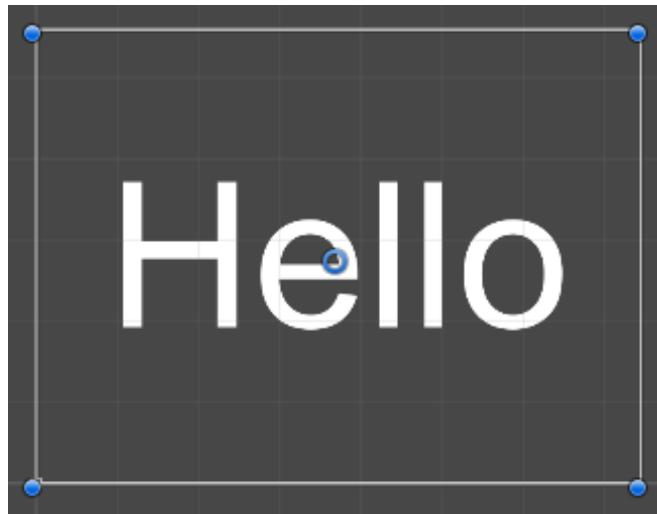


Figure 16.7: A Canvas in Screen Space - Overlay

Converting it to **World Space** doesn't initially cause any problems, but once we scale it down to something like a **Width** of 4 and **Height** of 3 (since it was initially created with a 4:3 aspect ratio screen), the text will disappear!

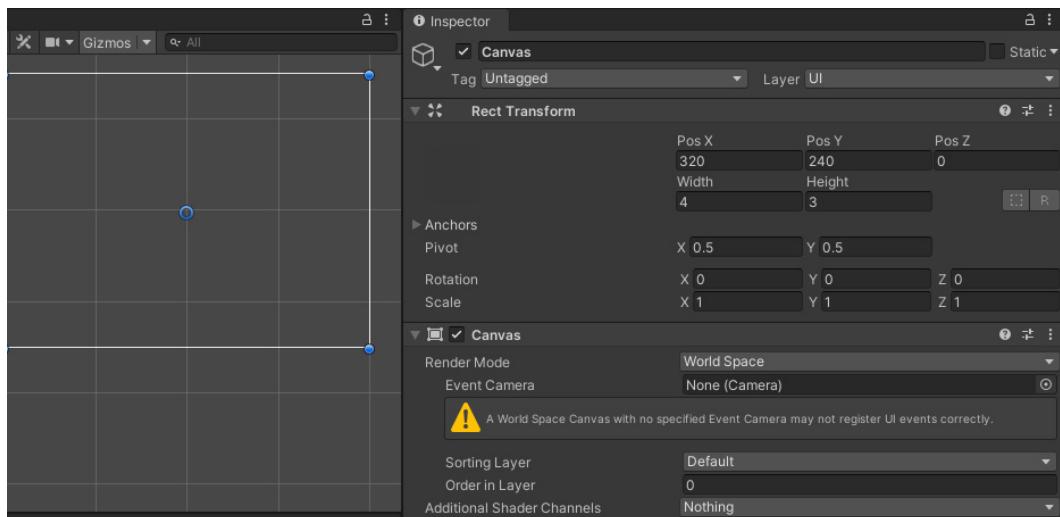


Figure 16.8: A Canvas in Screen Space - Overlay

If I set the **Text** to allow **Horizontal Overflow** and **Vertical Overflow** you'll see that it is huge when compared to the Canvas! In the following screenshot, the tiny rectangle in the middle is the Canvas:



Figure 16.9: World Space Canvas with huge overflowing text example

To fix this, and to get it looking the way we want, we need to adjust the **Dynamic Pixels Per Unit** property on the **Canvas Scaler** component (initially discussed in *Chapter 6*). This property is initially set to 1.

Usually, to determine the new **Dynamic Pixel Size** value, I take the starting **Width** of the Canvas before I scale it down, which is 905, divide it by the new **Width** 4, and enter that division in my **Dynamic Pixels Per Unit** property. (Typing the actual division $905 / 4$ in the box will perform the calculation.)

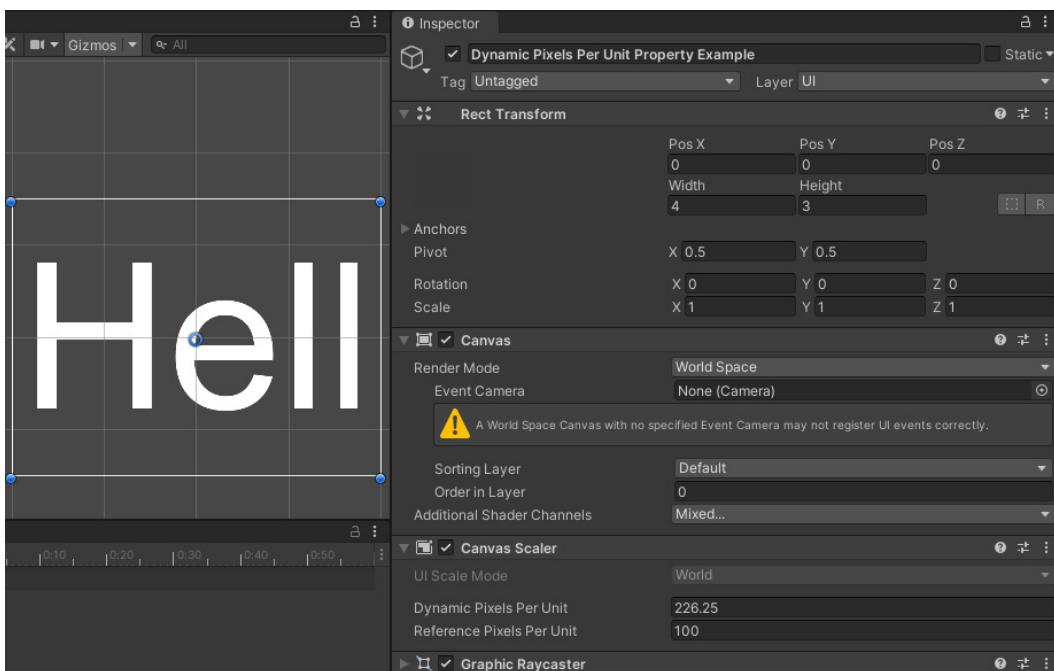


Figure 16.10: Dynamic Pixels Per Unit property adjusted

However, that calculation didn't get the exact look I was looking for. So, I increased the size until it looked right:

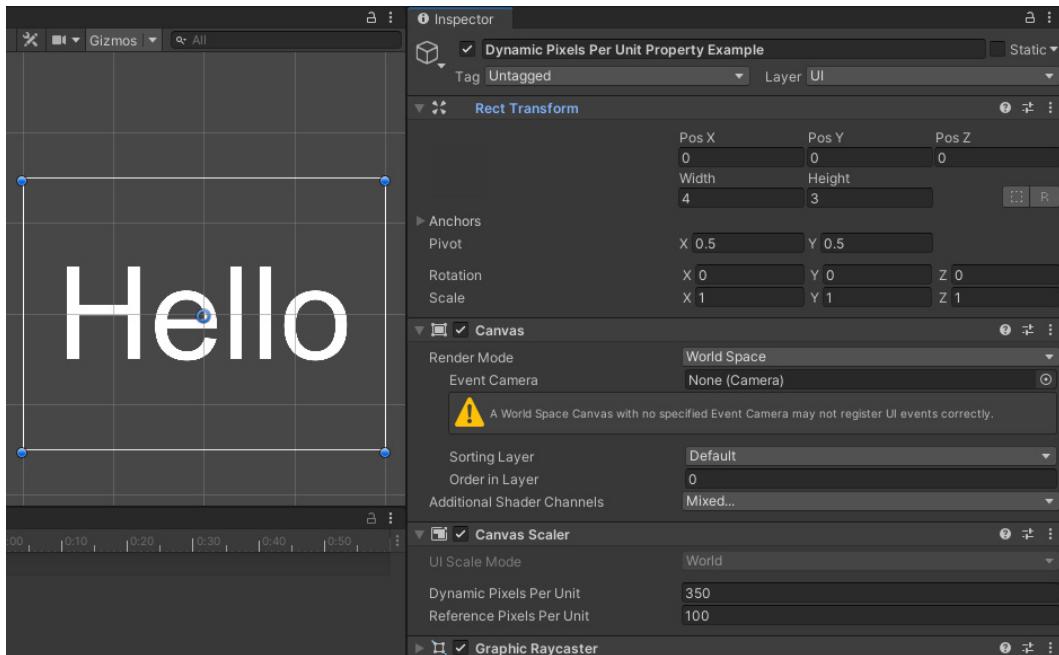


Figure 16.11: Dynamic Pixels Per Unit property example in the Chapter16 scene

Every time you change the **Width** and **Height** of the Canvas, you will have to adjust the **Dynamic Pixels Per Unit** property. Decreasing the size of the Canvas will mean increasing the **Dynamic Pixels Per Unit** property, and increasing the size of the Canvas will mean decreasing the size of the **Dynamic Pixels Per Unit** property.

Here are two Canvases, both one-fourth the size of the one from the previous figure. In the top Canvas, I changed the **Width** and **Height** to 1 and .75. In the bottom Canvas, I changed the **Scale X** and **Scale Y** to 0.25:

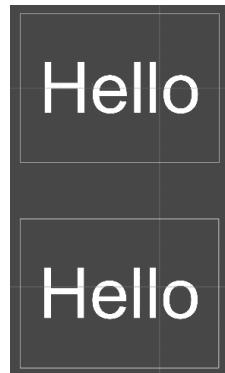


Figure 16.12: Width and Height change examples

In the first example, since I changed the **Width** and **Height** of the Canvas to one-fourth of the size, I typed `350*4` in the **Dynamic Pixels Per Unit** property, and it automatically calculated `1400` for me (I love that Unity performs calculations in the boxes).

However, in the second Canvas, I did not have to change the **Dynamic Pixels Per Unit** size, because scaling with the **Scale** property in this way does not require me to change it.

The takeaway from this is that if your text isn't displaying or looks incredibly blurry, adjust the **Dynamic Pixels Per Unit** property until it looks the way it should, or scale your Canvas by adjusting its **Scale** and not its **Width** and **Height**.

Text scaling will be the most important consideration with using World Space UI, but let's review some other important topics.

Other considerations when working in World Space

For the most part, working with UI in World Space isn't much different than working with UI in Screen or Camera Space. There are a few things you have to keep in mind, though.

When working with 3D scenes, you may want your UI to always face the player, regardless of how the player turns the camera—this is known as a *billboard effect*. You can achieve this with a simple `LookAt()` function on the transform of the object in the `Update()` function:

```
transform.LookAt(2*transform.position-theCamera.transform.position);
```

You can use a variation of the preceding code, depending on how you want the rotation to behave.

Another consideration with 3D World Space UI is the distance it is away from the camera. You may want to have UI only render when it is a specific distance from the camera as it may be difficult to see when it is too far away.

Depending on your project, using World Space Canvases may cause difficulties with Raycasting, making interacting with UI a problem. Tyrus Peace of Floating Island Games recommends creating your own physics layer if you end up having to create your own Raycasting system, as he did with *Cloudbase Prime*, shown earlier in the chapter.

Examples

Working with World Space Canvases isn't significantly different than working with Canvases in Screen and Camera Space. World Space Canvases offer many benefits. If you have an object that exists within your scene that has UI specifically tied to its location, it is helpful to use a World Space Canvas so that the UI follows it wherever it is. This removes the necessity of trying to convert the object's World Space coordinates to Screen coordinates to ensure that the UI always lines up with the object. It also guarantees that the UI object will always display correctly with respect to the object's location, even when the screen's resolution changes. In this chapter, I will cover two common uses of World Space Canvases: one in 2D space and another in 3D space. Let's begin with 2D space.

2D World Space status indicators

For this example, we will start a new scene. For you to not have to build out the scene, we will start with an Asset package that includes all the required items.

We'll create UI that allows a character to have a status indicator pop up above his head. After the scene has played for 3 seconds, a status-indicating button will appear over the character's head. Once the player clicks on the status indicator, a dialog will appear. After 5 seconds, the dialog will disappear. The status indicator will re-appear 10 seconds later.

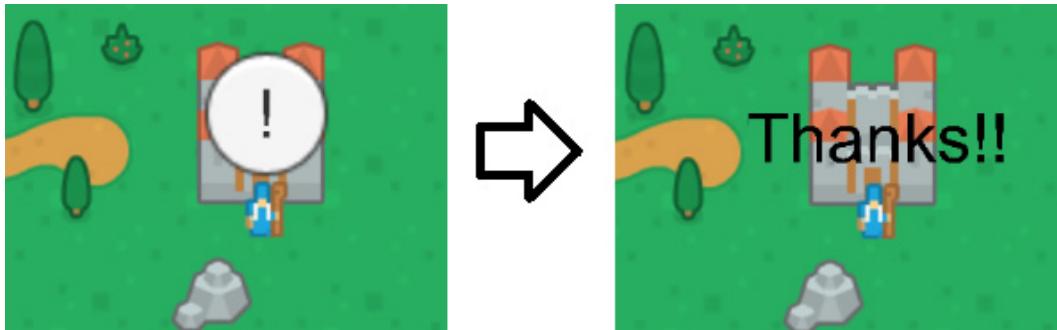


Figure 16.13: 2D World Space UI example

The art used in this example was accessed from <https://opengameart.org/content/medieval-rts-120>.

To create the status-indicating UI demonstrated by the previous example, complete the following steps:

1. Import the `Chapter 16 - Example 1-Start.unitypackage` package. This package contains a scene with a background image and a 2D sprite named Mage. The `Assets/Scripts/MageInteractions.cs` script included with the package controls the timers on the appearance of the status indicator. This script requires two Canvas Group items—the `ExclamationPoint` and the `DialogBox`—and contains a function, `ShowTheDialogBox()`, that can be called via a button's `On Click()` event.
2. We want the status indicator and the dialog to be tied to the position of the Mage within the scene. Therefore, we will create a Canvas that is a child of the Mage in **World Space**.
Right-click on the Mage in the Hierarchy and add a UI Canvas as a child of the Mage.
3. Select the newly created Canvas. Change the **Render Mode** on the **Canvas** component to **World Space**.
4. Assign the Main Camera to the **Event Camera** slot.
5. Since this Canvas is a child of Mage, its coordinate system is relative to the Mage. To have it perfectly positioned over the Mage, change the **Rect Transform** component's **Pos X** and **Pos Y** to 0.

6. The Canvas is significantly bigger than the Mage. Make the size more reasonable by changing the **Rect Transform** component's **Width** and **Height** to 1.
7. Now that we have the Canvas scaled and positioned in the scene around the Mage, we can add UI elements to it. Right-click on Canvas in the Hierarchy and create a UI Button. Name the new Button Alert.
8. Resize Alert to match the Canvas by setting its **Rect Transform** component's stretch and anchor to stretch fully across the Canvas.
9. Change the **Source Image** on the Alert Button's **Image** component to the UI Knob image. It looks better as a circle than as the default button sprite:

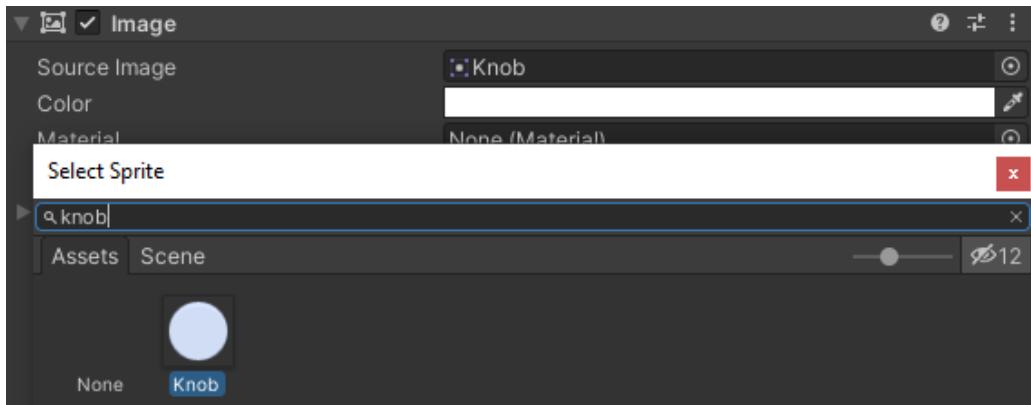


Figure 16.14: Selecting the UI Knob image

10. Change the Text child of Alert to display an exclamation point instead of Button.
11. The Button's text is not currently visible. To fix this, change the **Dynamic Pixels Per Unit** property on the **Canvas Scalar** component of Canvas to 1000.
12. Move the Alert Button so that it is positioned over the Mage's head.

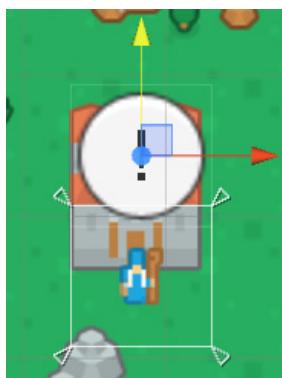


Figure 16.15: Moving the exclamation point over the Mage's head

13. Add a **Canvas Group** component to the Alert Button.
14. Set the **Canvas Group** component's **Alpha** property to 0 and set both the **Interactable** and **Blocks Raycasts** properties to **False**.
15. Give the Alert Button an **On Click()** event that calls the `ShowTheDialogBox()` function on the `MageInteractions` script attached to the Mage:

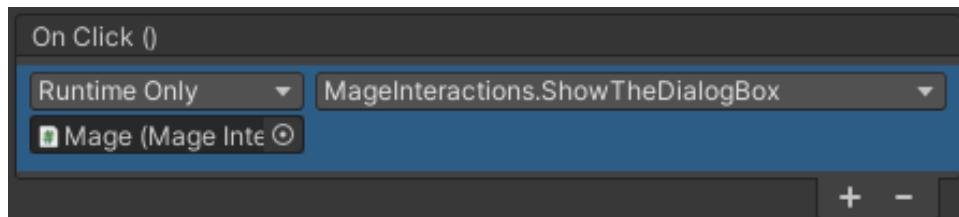


Figure 16.16: The On Click() event of the Alert Button

16. Right-click on Canvas in the Hierarchy and create a UI Text object. Name the new Text object **Dialog**.
17. Resize the **Dialog** object to match the Canvas by setting its **Rect Transform** component's stretch and anchor to stretch fully across the Canvas.
18. Change the **Text** component on the **Dialog** object to say **Thanks!!!**. Also, center-align the text and set the **Horizontal Overflow** property to **Overflow**.
19. Move the **Dialog** object so that it is positioned above the head of the Mage, like so:

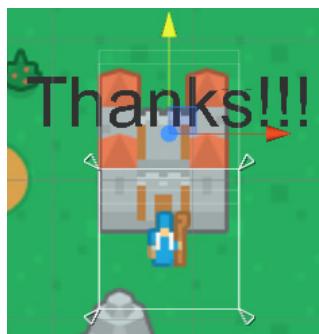


Figure 16.17: The Dialog box

20. Add a **Canvas Group** component to the **Dialog** Text object.
21. Set the **Canvas Group** component's **Alpha** property to 0 and set both the **Interactable** and the **Blocks Raycasts** properties to **False**.
22. Select the Mage and add **Alert** to the **The Exclamation Point** property. Add **Dialog** to the **The Dialog Box** property.

If you play the game, you'll see the exclamation point Button appear after 3 seconds. Clicking on the Button will make the Text appear. Try moving the Mage character around in the scene. You'll see that no matter where he is, the exclamation point Button and Text appear over his head. This is a really helpful technique for creating UI elements that stay with moving characters.

I know that the example is a bit boring the way it is now, but I recommend using some of the techniques discussed in the previous two chapters to add a nice bouncy animation to the exclamation point and have the Text fade in and out.

3D hovering health bars

Making World Space UI in a 3D scene takes a little more work than making World Space UI in a 2D scene if the camera can be rotated and moved throughout the 3D space. If the camera can move and rotate, the UI likely needs to constantly *face* the camera. Otherwise, the player will not be able to see the UI elements.

For this example, we will once again create a new scene. For you to not have to build the scene from scratch, we will start with an Asset package that includes all the required items.

We'll create a simple hovering health bar that constantly faces the camera. It will also receive clicks so that we can watch the health bar reduce:



Figure 16.18: Astronaut with a hovering health bar

The art used in this example was accessed from <https://opengameart.org/content/space-kit>.

To create a health bar that always faces the camera and receives player-click input, complete the following steps:

1. Import the Chapter 16 - Example 2 - Start.unitypackage package. This package contains a scene with a 3D character facing the camera. The camera has a simple `RotatingCamera` script attached to it that rotates the camera around the astronaut with the mouse. The package also contains a `ReduceHealth` script that is attached to the astronaut character. This script has a function, `ReduceHealthBar`, that we will call when the health bar above the character's head is clicked on.
2. We want the health bar to be tied to the position of the astronaut within the scene. Therefore, we will create a Canvas that is a child of the astronaut in World Space.
Right-click on the astronaut in the Hierarchy and add a UI Canvas as a child of the astronaut.
3. Select the newly created Canvas and change the **Render Mode** on the **Canvas** component to **World Space**.
4. Assign the Camera to the **Event Camera** slot.
5. Since this Canvas is a child of astronaut, its coordinate system is relative to astronaut. To have it perfectly positioned over the astronaut, change the **Rect Transform** component's **Pos X** and **Pos Y** to 0.
6. The Canvas is significantly bigger than the astronaut. Make the size more reasonable by changing the **Rect Transform** component's **Width** to 10 and the **Height** to 1.
7. Position the Canvas so that it is above the head of the astronaut:

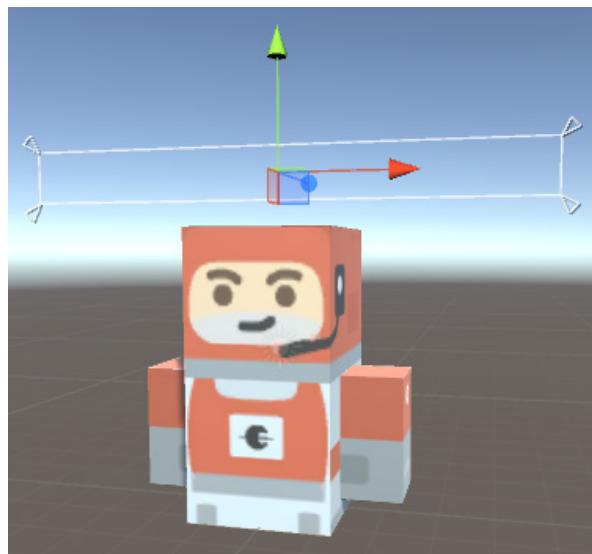


Figure 16.19: Scaling the Canvas over the astronaut

8. Now that we have the Canvas scaled and positioned in the scene around the astronaut, we can add UI elements to it. Right-click on Canvas in the Hierarchy and create a UI Button. Name the new button Health Bar.
9. Resize the Health Bar to match the Canvas by setting its **Rect Transform** component's stretch and anchor to stretch fully across the Canvas.
10. Change the **Source Image** on the **Image** component of the Health Bar to **None** to give it a white rectangle as an image.

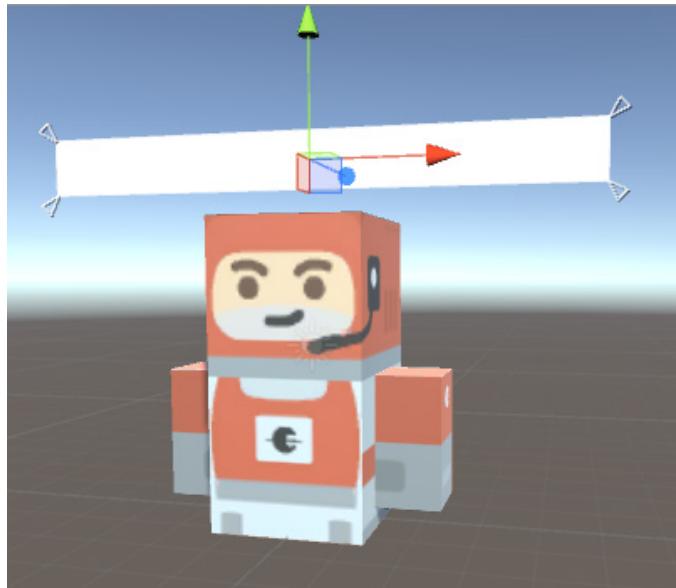


Figure 16.20: Adding a blank image to the Button

11. Change the Text child of Health Bar to say Click to reduce my health.
12. Set both the **Horizontal Overflow** and **Vertical Overflow** properties of the **Text** component on the Text child to **Overflow**. This will allow you to see the size the text is currently rendering at in the scene:



Figure 16.21: The oversized text of the Health Bar

13. Set the Text's **Font Size** to 10 and deselect **Raycast Target** on the **Text** component.

14. Select the Canvas and hover over the **Dynamic Pixels Per Unit** property in the **Canvas Scaler** component until you see two arrows appear around your mouse cursor. Once you see those arrows, click, and drag to the right. This makes the property work like a slider, allowing you to see how increasing the **Dynamic Pixels Per Unit** property continuously changes the way the text renders in the scene. Do this until the text fits within the Canvas:



Figure 16.22: The oversized text of the Health Bar

Note

When trying to get the text to look nice in 3D space, if only changing the **Dynamic Pixels Per Unit Size** results in *choppy* text, change the property until the text looks perfectly crisp in the scene. Then, use a combination of changing the **Rect Transform** component's **Scale** and **Font Size** of the Text object to find the *sweet spot*.

15. Right-click on the **Health Bar** Button and add a UI Image as a child. Name the new Image **Health Fill**.
16. Resize **Health Fill** to match the **Canvas Health Bar** by setting its **Rect Transform** component's stretch and anchor to stretch fully across the **Health Bar**.
17. Now, change the anchor and pivot to **left stretch** so that it will scale *leftward*.

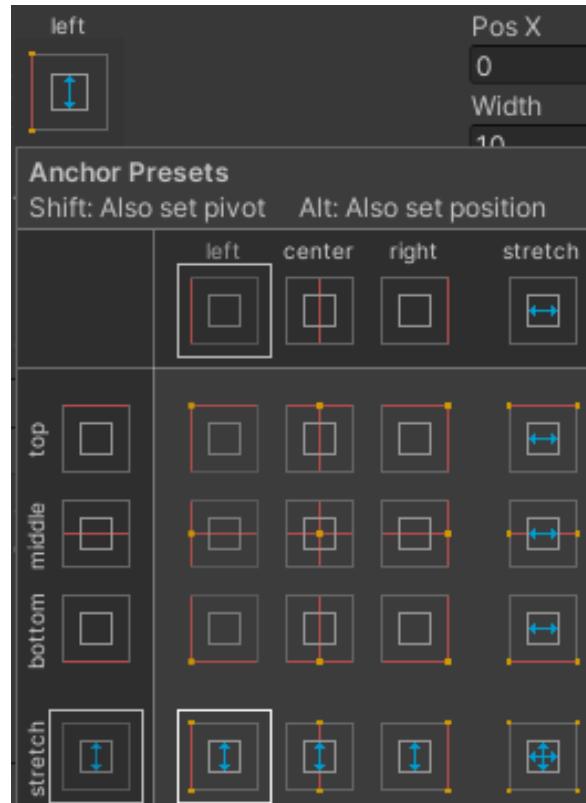


Figure 16.23: Left-stretching anchor

18. Reposition the Health Bar in the Hierarchy so that it is above Text. This will have the fill render behind the Text object.

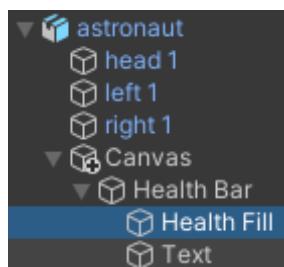


Figure 16.24: The Hierarchy of GameObjects

19. On the **Image** component of the **Health Fill**, change the **Color** property to red and deselect the **Raycast Target** property.
20. Select the astronaut and assign the **Health Fill** object to the **Health Fill** property on the **ReduceHealth** component.
21. Add an **On Click()** event to the **Health Bar** Button that calls the **ReduceHealthBar** function of the **ReduceHealth** script on the astronaut:

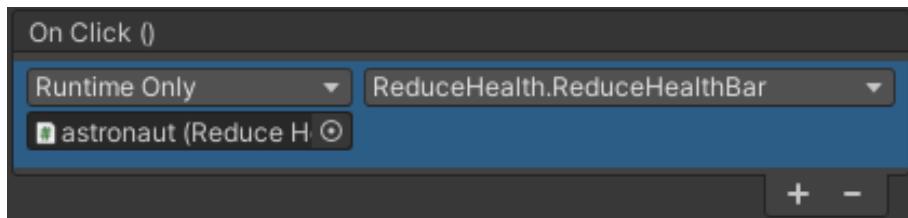


Figure 16.25: The OnClick() event of the Health Bar

Playing the game now should result in the **Health Fill** reducing its fill value when you click on the **Health Bar** Button:



Figure 16.26: The Health Bar button reducing

22. Now, we just need to add a billboard effect to the **Canvas**. Create a new script called **BillboardPlane** in the **Assets/Scripts** folder.
23. Change the script of the **BillboardPlane** class to the following:

```
public Camera theCamera;

void Update()
{
    transform.LookAt(2 * transform.position - theCamera.
    transform.position);
}
```

24. Attach the `BillboardPlane` script to the Canvas.
25. Assign the Camera to the `Camera` slot in the `BillboardPlane` component.

If you play the game now, you'll see that as you move the camera around, the health bar always faces the Camera. Try changing the `Transform` position of the camera in the scene to see the `LookAt()` function work more drastically.

Summary

World Space UI is not significantly different in its implementation than UI that renders in the Camera or Screen Space. Adding UI to your World Space gives you the ability to create cool effects and gives you more control over your UI's position relative to objects in the scene.

In the next chapter, we will discuss how to optimize Unity UI.

Optimizing Unity UI

Optimization is the process that we use to make sure that our game runs smoothly and the framerate is consistent. Through optimization we first locate resources within our game that are reducing our game's performance and then implement solutions that will improve that performance.

There are lots of things that can cause a game to have poor performance or low framerate. This can include things like unoptimized lighting, poorly written scripts, large assets, and improper UI construction. Since this is a UI book, we'll focus only on how improving your UI construction can improve your performance.

In this chapter, I will discuss the following:

- Key terms and basic information related to optimization
- An overview of the tools provided within unity that can help you determine how performant your game is
- Various optimization strategies for UI

Before you can optimize your UI, you need to learn how to tell if it is performant or not. Let's review some basic terms and principles of graphics rendering.

Note

This chapter will only cover performance profiling on a basic level and its focus will be more on things you can do to build better UI. If, by the end of the chapter, you'd like more information on performance profiling, I suggest the following resource: <https://unity.com/how-to/best-practices-for-profiling-game-performance#gpu-bound>

Optimization basics

As I stated earlier, optimization is the process that we use to make sure that our application runs smoothly, and the framerate is consistent. We want to optimize our game to ensure that all players have the same experience regardless of the conditions in which they are playing. So, for example, if we are making a PC game, we want to make sure that every player has the same experience regardless of the power of their machine. We also want to make sure if a player has many things rendering on the screen, the game does not lag compared to when there were less things rendering on the screen.

We do not want the frame rate to reduce, and we do not want the inputs to lag. This is extremely important for things like games. In something like a first-person shooter or platform this could a lag in input could mean a player loses a match or falls off a cliff.

Let's review some basic terminology that you hear often when discussing optimization. First, we'll start with a common metric for determining an application's performance.

Frame Rate

Framerate is one metric in which we measure our application's optimization. It's a good metric, because it's not just something that is happening in the background that the user never notices. Users can see and notice changes in framerate, so measuring its performance measures how our users experience our games.

Framerate can be measured in **frames per second (fps)** or time in milliseconds. The goal is to have a consistent framerate regardless of what is happening in the game.

When measuring framerate in frames per second, each individual frame renders out in a certain amount of time. Let's say you have a goal of 60 fps. You would need to make sure that all aspects of your application can run at 60 fps. So, your goal is to have a consistent number of frames per second. Therefore, when we set a fps benchmark, that means we want our application to run at that fps consistently across all parts of the application. We don't want our game to run at 60 fps at the beginning and then dip when our UI opens.

Another way to measure framerate is time in milliseconds. Every frame takes some amount of time to render. We want that amount of time to be as low as possible for each frame. 10 ms per frame is the equivalent of 60 fps and is a good time in milliseconds.

Now that we've discussed what frame rate is, let's talk about where resources are used on our computer.

GPU and CPU

When trying to optimize your game, you will investigate your game's resources to determine where the most resources are being used. You will want to identify whether the issues are on the GPU or CPU. The **Central Processing Unit (CPU)** is the brain of the computer. The **Graphics Processing Unit (GPU)**, is what renders images. Whether our resources are on the GPU or CPU will determine the optimization solution.

Examples of problems on the CPU are too many instructions, meaning too many scripts are running or the scripts take too long to run. Problems with the GPU tend to mean too many things are being rendered. The CPU does still play a part in rendering, however, as the CPU gives out the instructions and tells the GPU what to render. Often when discussing GPU and CPU, you'll hear the term **draw calls**. A draw call is when the CPU tells the GPU it needs to draw (or display) something on the screen. In general, you want to reduce the number of draw calls your game makes.

We'll discuss GPU and CPU and how various choices we make designing our UI are affected by GPU and CPU more thoroughly in future sections of this chapter.

Now that we've covered some basic concepts of optimization, let's review some tools provided by Unity that help us with optimization.

Tools for determining performance

When determining your game's performance, you may use what is called benchmarking. **Benchmarking** allows you to see how well your app is running at various times under given conditions. When benchmarking, you collect performance metrics and establish a baseline, or benchmark metric. You then compare subsequent results to that benchmark to see if the metric has improved or worsened. This lets you know if changes you have made have affected your game's performance.

There are a few tools provided by Unity that can help you assess the performance of your game by providing you performance metrics. Let's look at those tools now.

Note

Be aware of the environment and what you are benchmarking. It's recommended that you benchmark on your target platform. That way you can determine the minimum standard device you plan to run your game on. The performance of your game will only get better with a better device.

Statistics window

One simple way to view the performance of your game is through the **Statistics** window (or **Stats** windows) of the **Game** view.



Figure 17.1: Analyzing the Stats Window

If you select the **Stats** button in the top right corner of your **Game** view, you'll see various information about your project. Under the **Graphics** section, you'll see frame rate in both frames per second and time in milliseconds. This will continuously change as it is based on each frame. You also see information about the CPU main thread and render thread.

If you're using frame rate as your benchmark, you can watch the framerate values here and see how they change over time. Remember, the goal is consistency.

Note

You can learn more about the stats window, here: <https://learn.unity.com/tutorial/working-with-the-stats-window-2019-3?uv=2019.4#>

The **Stats** window gives some at a glance basic information about your game's performance. However, if you want a more in-depth breakdown of how your game is running, you can use the Unity Profiler.

Unity Profiler

The Profiler shows you detailed information about your game's performance. To view the Profiler, select **Window | Analysis | Profiler**. When you play the game, you can then see which items are responsible for performance issues.



Figure 17.2: Viewing the Unity Profiler

Within the **Profiler**, you can select the UI Modules only to narrow down how each of your individual UI elements are affecting your performance.

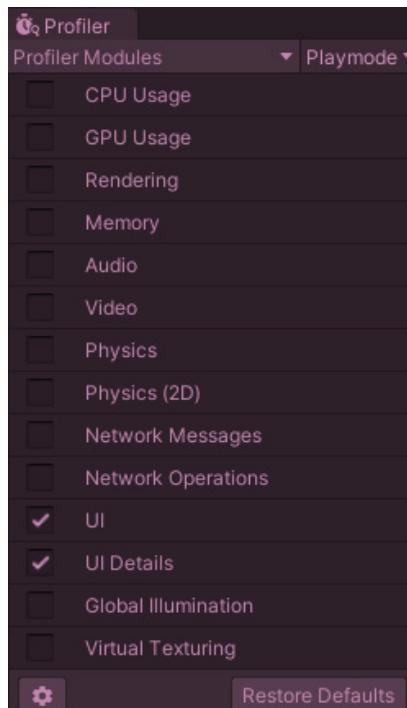


Figure 17.3: Enabling only the UI Profiler Modules

After doing so, you can see information about each individual Canvas.



Figure 17.4: Observing the UI Objects in the Profiler

Note

You can learn more about the Unity Profiler and how to use it here: <https://docs.unity3d.com/Manual/Profiler.html>

If you're interested in how items in your game are being batched, you can use the Unity Frame Debugger.

Unity Frame Debugger

The Frame Debugger can help you troubleshoot batching issues with your UI. You can access the Frame Debugger via **Window | Analysis | Frame Debugger**.

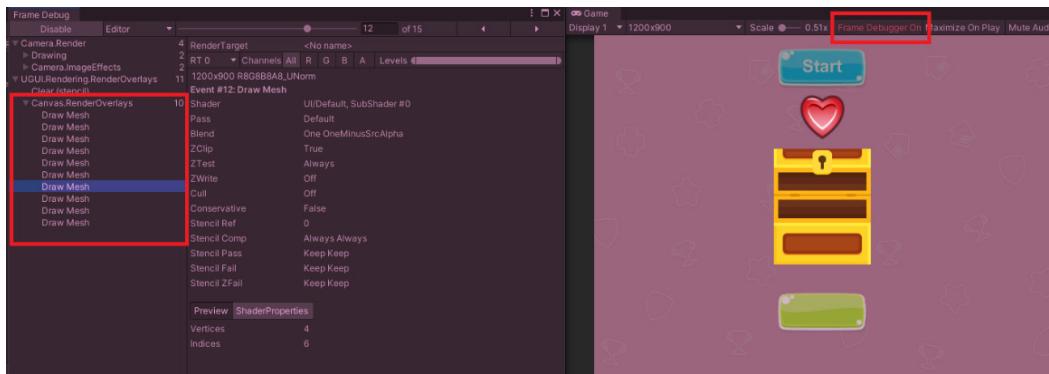


Figure 17.5: Reviewing the Frame Debugger

Enabling the Frame Debugger will allow you to see various rendering events. Clicking on them will step forward in the Game window, giving you a preview of the event. This will show you which items are being combined into batches.

Note

You can learn more about the Frame Debugger here: <https://docs.unity3d.com/Manual/frame-debugger-window.html>

Now that we've discussed the basics of optimization, we can start talking about ways to optimize your UI.

Basic Unity UI Optimization Strategies

Remember, each Canvas has its own Canvas Renderer component. Canvases combine all their elements into batches that are rendered together. A Canvas is considered **dirty**, if its geometry needs to be rebuilt. One of the main goals of optimizing UI is to reduce the number of times a Canvas or its elements are considered dirty, to reduce the number of times that the Canvas needs to be rebatched. With that in mind, let's look at some techniques for optimizing Unity UI.

Using multiple Canvases and Canvas Hierarchies

Whenever an element on a Canvas is modified, the Canvas is considered dirty, and a draw call is sent to the GPU. If there are multiple items on the Canvas, all the items on the Canvas will need to be reanalyzed to determine how best they should be drawn. So, changing one element on the Canvas requires the CPU to rebuild every element on the Canvas, potentially causing a sudden surge in CPU usage. Due to this, you should put your UI on multiple Canvases.

When determining how to group your Canvases, consider how often the items on the Canvas will need to be changed. It is good practice to group all static UI element on separate Canvases from items that are dynamic. This will stop the static items from having to be redrawn whenever the dynamic items change. Additionally, split dynamic elements into Canvases based on when they will update, trying to keep ones that update at the same time together.

Other aspects to consider when attempting to reduce draw calls on Canvases are the elements z-coordinates, textures, and materials. Grouping UI elements based on these properties can also reduce your CPU usage with regards to UI.

Minimizing the use of Layout Groups

Layout Groups are incredibly helpful when you want properly spaced UI; however, they are extremely inefficient. Whenever an element in a Layout Groups changes, every element in that group must call at least one `GetComponent` for each parent object that has a Layout Groups. This makes nested Layout Groups very non-performant.

To avoid this, you could simply not use Layout Groups. Instead, you could use Anchors and write your own layout code for dynamically placing items.

Avoiding them entirely isn't necessarily the best solution for everyone, however, so if you do choose to use them, try to avoid nested Layout Groups or layout groups with very large amounts of children. You can also choose to use layout groups for dynamic elements only and avoid them with static elements. You can also disable them as soon as dynamic UI has been properly positioned.

Hiding objects appropriately

If you want to hide elements on a Canvas, it is best to disable the entire Canvas component if possible. Remember, the Canvas component is what renders the UI, so disabling the Canvas component will cause it to stop rendering. It is recommended you disable the Canvas component rather than simply changing the visibility of the Canvas, because the Canvas component will continue to make draw calls, even if it is not visible. Disabling the component does not cause the Canvas to rebuild. This is less expensive than enabling and disabling the Canvas GameObject, as that will trigger the `OnEnable` and `OnDisable` methods.

If you want to hide everything in your game with UI, for example you have a pause menu that completely covers your screen, you should stop the camera from rendering anything in your game other than the UI. For example, if you have a menu that completely covers the screen, even though you cannot see the items behind the menu, they are still being rendered.

Appropriately time object pooling enabling and disabling

Object pooling is an optimization technique used to reduce the number of times repeatable objects are created (or instantiated). When object pooling, you place a collection of disabled objects in a pool at the start of the game and then, rather than instantiating those objects while the game is playing, you pull them from the pool.

Object pooling is a great way to improve the performance of your game, because it reduces the number of times an object has to be created by creating a collection of objects that can be reused.

Note

For more information on the concept of object pooling, visit the following site: <https://learn.unity.com/tutorial/introduction-to-object-pooling>

Because reparenting UI objects causes a Canvas to be marked as dirty, you want to carefully time your disabling, enabling, and reparenting when using an object pool. Since dynamic changes to elements within a Canvas cause the Canvas to send a draw call to the GPU, inappropriately parenting items before or after disabling them, can double up your draw calls unnecessarily. The goal, is to not cause the objects in the pool to send any draw calls to the GPU. So, objects in the pool should only ever be parented to the pool in a disabled state.

If you are placing an object in an object pool, disable it, then reparent it into the pool. By making sure it is disabled before being placed in the pool, you will remove the need for the pool to need rebuilding. Conversely, if you want to remove an object from the pool, reparent it, then enable it. This too will remove the need for the pool to send a draw call.

Reducing Raycast computations

Every UI Image component has the property **Raycast Target**. By default, this is selected. If you do not wish for your UI element to block raycasts, disable this to reduce raycasting checks.

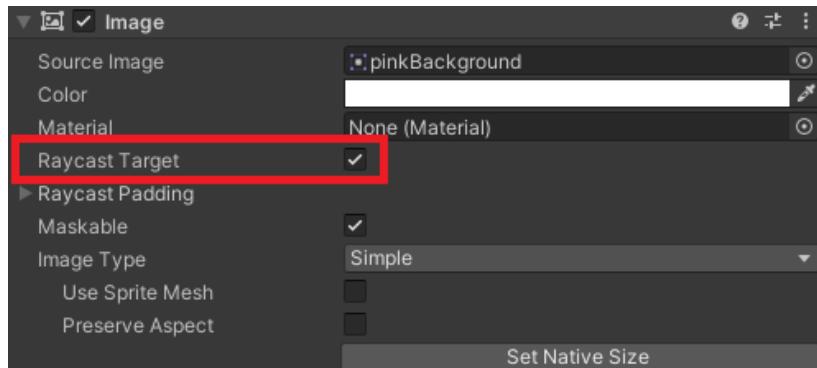


Figure 17.6: The Raycast Target property

Additionally, if your UI is not interactable at all, you can remove the Graphic Raycaster component to remove the unnecessary calculation it causes.

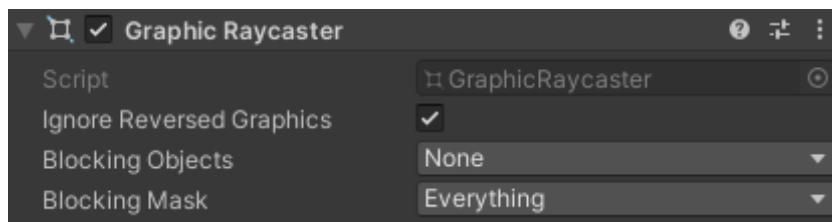


Figure 17.7: The Graphic Raycaster component

Using the basic Unity UI optimization strategies covered in this chapter will have you well on your way to creating UI that does not inversely affect the performance of your game.

Summary

In this chapter, we discussed the basics of optimizing Unity UI. We reviewed some fundamental concepts of optimizing in Unity, looked at the tools that allow us to assess our game's performance, and then discussed strategies for creating performant UI.

Note

This chapter focused only on how to improve the performance of your game through UI. But remember there are multiple aspects of your game that can cause it to have performance issues. For a good resource on improving the performance of all aspects of your game, I suggest reviewing the free ebook provided by unity on optimization: <https://resources.unity.com/games/performance-optimization-e-book-console-pc>

In the next chapter, we'll talk about a new UI system created by unity called the new unity UI toolkit. It can be used to create UI not only during runtime but also within the editor. Thus, it can be used to help you create tools to improve your workflow.

Further reading

For more information on optimizing UI, I recommend the following resources:

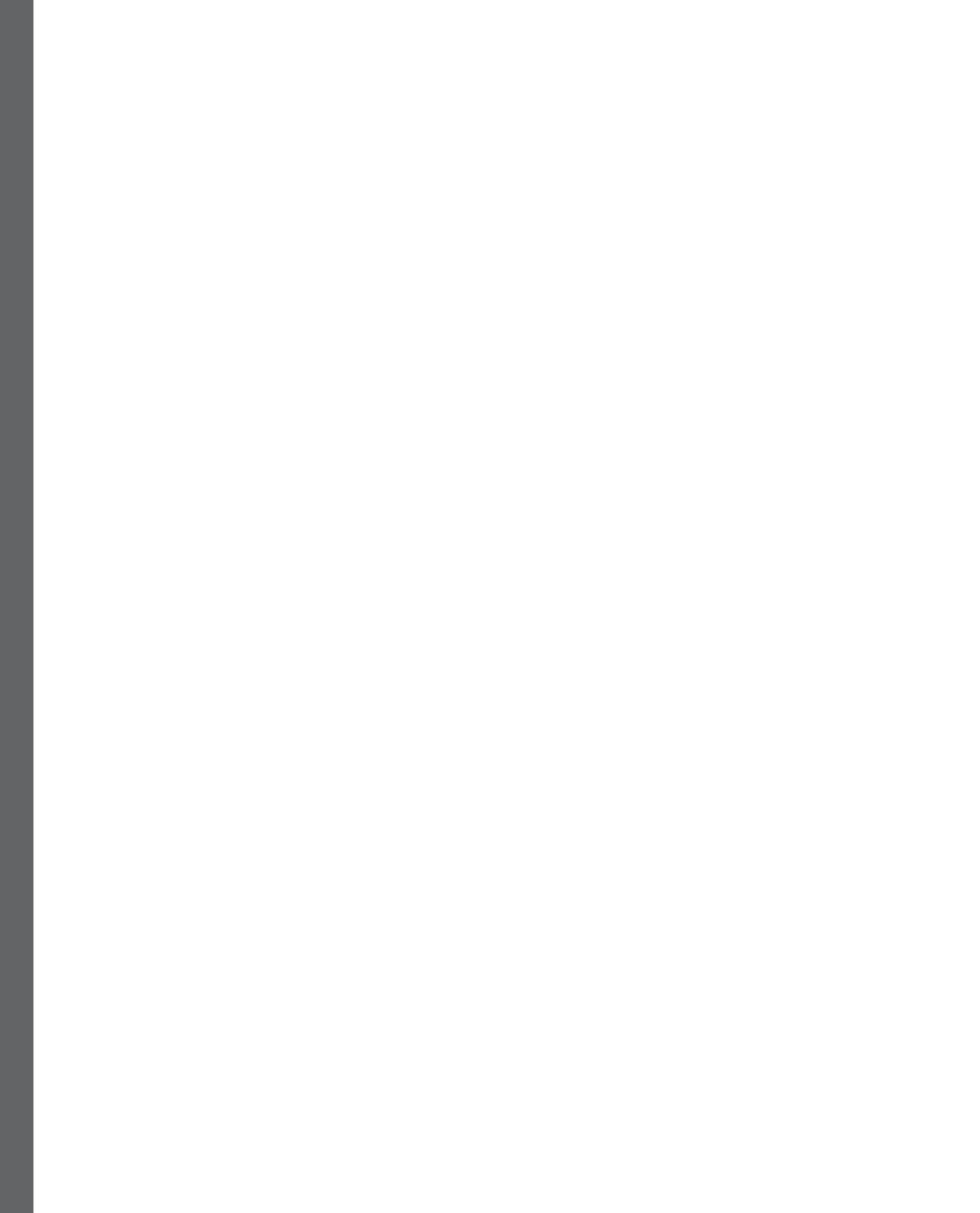
- <https://learn.unity.com/tutorial/optimizing-unity-ui#>
- <https://unity.com/how-to/unity-ui-optimization-tips>
- https://www.youtube.com/watch?v=FPotgj_NHK4

Part 5: Other UI and Input Systems

In this part, you'll step away from exploring the Unity UI system and look at the two other systems provided by Unity to create UI – UI Toolkit and IMGUI. You'll also look at how to handle input with the New Input System.

This part has the following chapters:

- *Chapter 18, Getting Started with UI Toolkit*
- *Chapter 19, Working with IMGUI*
- *Chapter 20, The New Input System*



18

Getting Started with UI Toolkit

Up to this point, the focus of this book has been on **Unity UI (uGUI)**. However, Unity has been developing another system in which you can develop UI for your game: the **UI Toolkit**. It is based on the principles of web design and is meant to allow you to create UI in a more flexible and extensible way than the uGUI. It accomplishes this by divorcing the design and development of UI from scenes and GameObjects and instead creates UI via code and style sheets—just like in web design.

The UI Toolkit is an entirely different system, so beginning development with it may feel jarring at first if you are used to developing UI with uGUI. However, if you have experience with `.html` and `.css` or developing Android or iOS interfaces using XML, this should all feel very familiar.

Since the UI Toolkit is an entirely different system using entirely different principles than the rest of the chapters in this book, to fully explain all that you could do with it would merit a whole other textbook devoted purely to it. Therefore, this chapter will give you the basic information you need to get started with using the system, but not fully discuss every aspect of it. I will give you additional resources throughout this chapter to review if you'd like to go even further with your study of the UI Toolkit.

In this chapter, I will discuss the following:

- An overview of the UI Toolkit and how to install it if it's not already in your version of Unity
- The various parts of the UI Toolkit that work together to create and style an interface
- What are Visual Elements and how does the UI Toolkit Hierarchy work?
- How to use the UI Builder to design and layout UI Toolkit interfaces
- How to access UI Toolkit-built UI in C#
- Creating a virtual pet that hangs out with you in your Unity Editor and encourages you
- Using the UI Builder to create style sheets and animation transitions
- Using web requests to randomly generate your UI's images and text

Note

This chapter assumes you have a cursory knowledge of HTML and CSS. However, having more extensive experience with HTML and CSS will make adopting the UI Toolkit system significantly easier.

Before we jump into the inner workings of the UI Toolkit, let's review its use cases and when you will use it.

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2018>

Overview of UI Toolkit

As you may recall from *Chapter 5*, there are three total UI systems that can be used within Unity. Up to this point, the focus of this book has been on Unity UI (uGUI), which can be used to make in-game (aka runtime) UI. However, if you want to make UI that can be viewed in your Editor, you will have to use a different system. The two systems that can be used to create Editor UI are **IMGUI**, which we will discuss in the next chapter, and the UI Toolkit. However, while IMGUI can only be used to make Editor UI, the UI Toolkit can be used to make both runtime and Editor UI.

Where is the UI?

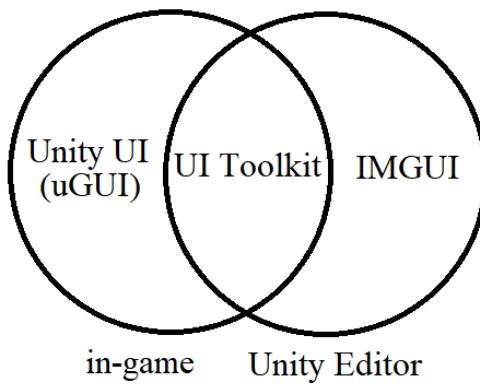


Figure 18.1: Comparing the three UI systems

Unity's goal is to replace the uGUI system entirely with the new UI Toolkit. However, it is still in development and does not have all the functionalities that uGUI does. For example, the UI Toolkit cannot make UI that is positioned in the 3D world, like what we discussed in *Chapter 16*. It can only make UI that overlays on top of the screen. Additionally, it cannot use custom materials and shaders and is not easily referenced from MonoBehaviours.

So, when would you want to use the UI Toolkit? The UI Toolkit is extremely helpful if you want to create stylistic UI without dealing with the bloat that comes along with prefabs and prefab variants. You can change the design of UI through code instead of altering a GameObject's Inspector properties. This makes UI more easily sharable across multiple projects and quickly customizable. Additionally, if you have experience with web development, you may be more comfortable with the workflow of the UI Toolkit.

Odds are, if you are working on an older project, you will need the information discussed in this book that concentrates on uGUI as it is still the most widely used UI system. However, if you are creating a new project and would like to consider using the UI Toolkit for your project, I recommend you review the following documentation to ensure your needs will be met by this system: <https://docs.unity3d.com/Manual/UI-system-compare.html>

Now that we've reviewed when you can use the UI Toolkit, let's get started with it by installing it in our project.

Installing the UI Toolkit package

Depending on which version of Unity you are using, the necessary UI Toolkit packages may not be installed. The version of the UI Toolkit that allows you to develop Editor UI comes packaged with all relatively recent versions of Unity; however, the version that allows you to also make runtime UI only comes with the most recent versions.

You can determine if you already have all necessary versions of the UI Toolkit package installed by going to **Window | UI Toolkit**. If the **UI Builder** submenu is not present or the UI Toolkit menu item is completely missing, you will need to import the package.

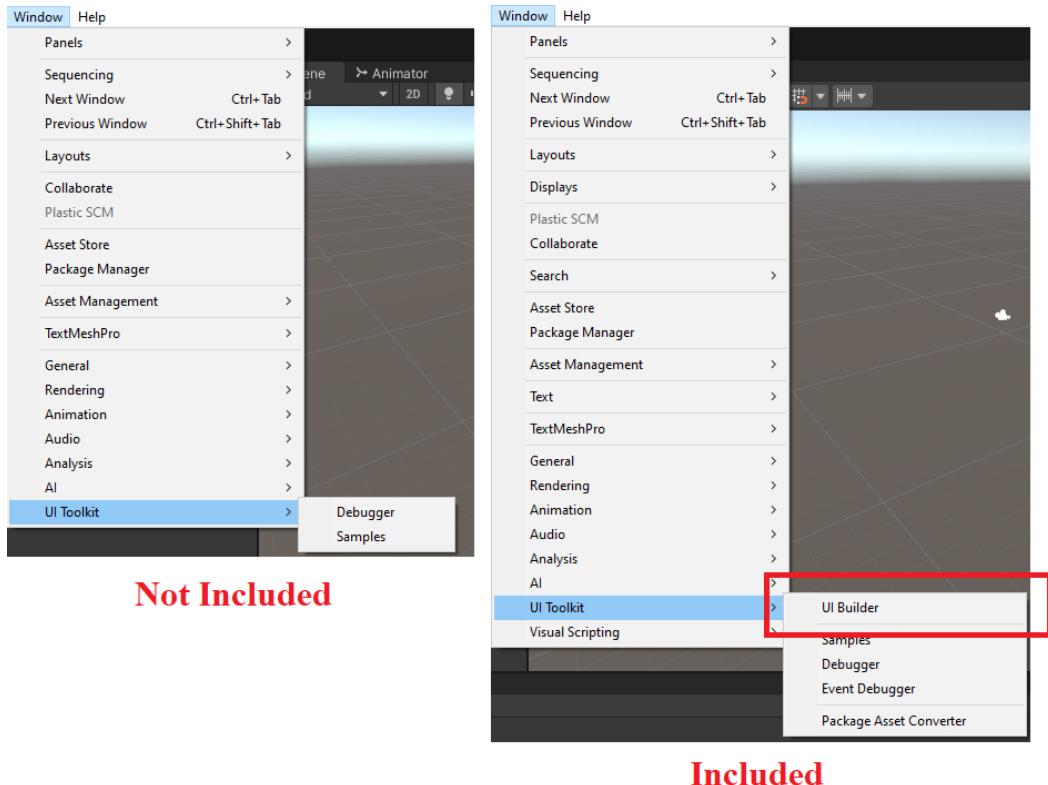


Figure 18.2: Determining if you have the UI Toolkit package installed

If you need to install the UI Toolkit, you can do so by importing the package from the `git` URL. To do so, complete the following steps:

1. Open the package manager with **Window | Package Manager**.
2. Select the **+** button, then choose **Add package from git URL....**

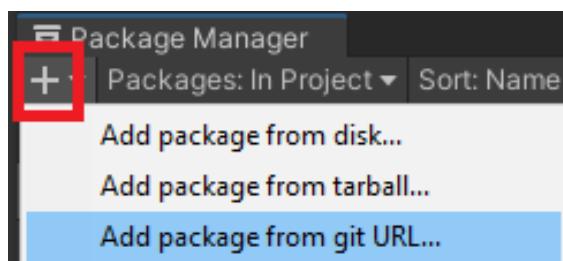


Figure 18.3: Adding a package via the git URL

- Type `com.unity.ui` into the text field that appears. Make sure not to add a space after the address. If you do, you will receive an error message.

After the import finishes, you should see the following toolkit in your package window.

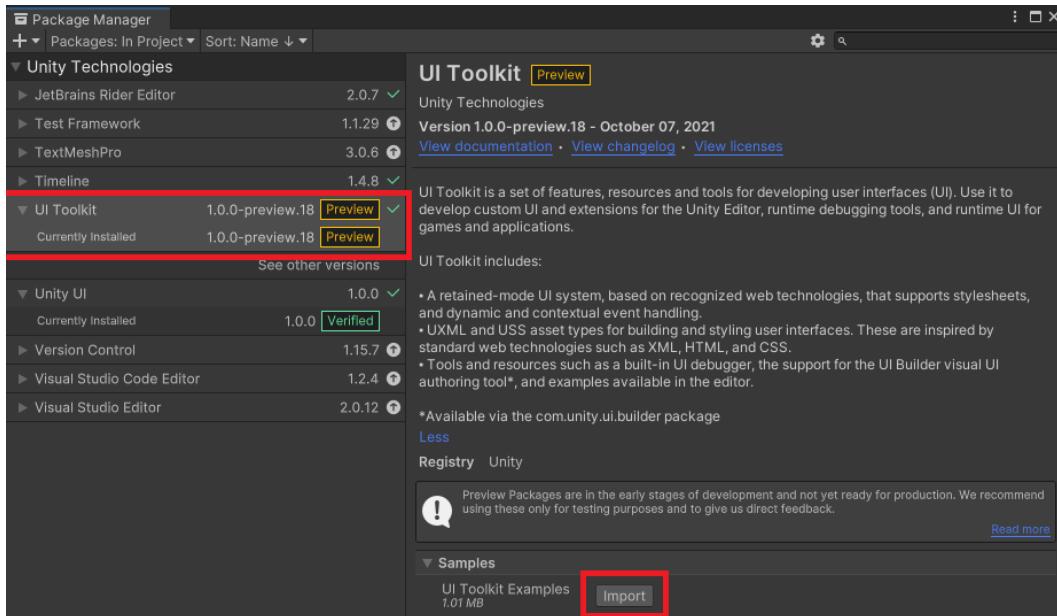


Figure 18.4: The UI Toolkit package

- If you'd like to import the examples into your project, select the **Import** button to download them.
- You may notice that the UI Toolkit description says that the UI Builder is not included in this package. So, you must import that package separately. To do so, select the **+** button, then choose **Add package from git URL...** again.
- This time, enter `com.unity.ui.builder` into the textbox that appears.

After the import finishes, you should see the following package.

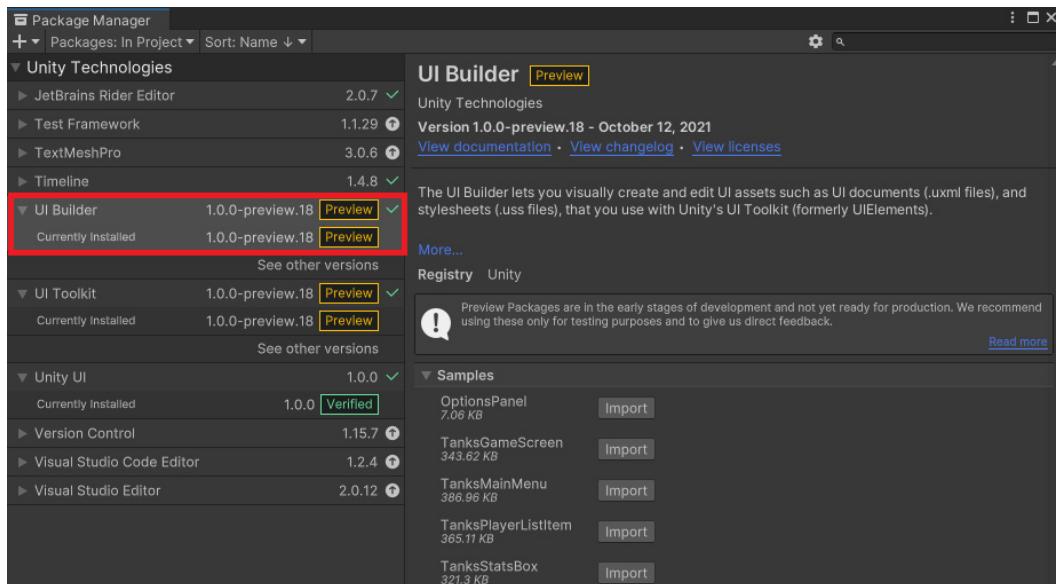


Figure 18.5: The UI Builder package

- As with the other package, there are samples you can import into your project with the **Import** buttons.

Now that we have the appropriate packages installed, let's look at the various parts of the UI Toolkit system.

Parts of the UI Toolkit system

The UI Toolkit uses a set of assets and a GameObject component to create UI. The primary assets used to create UI are as follows:

- UI Document (UXML)
- Style Sheet (USS)
- Panel Settings
- Theme Style Sheet (TSS)

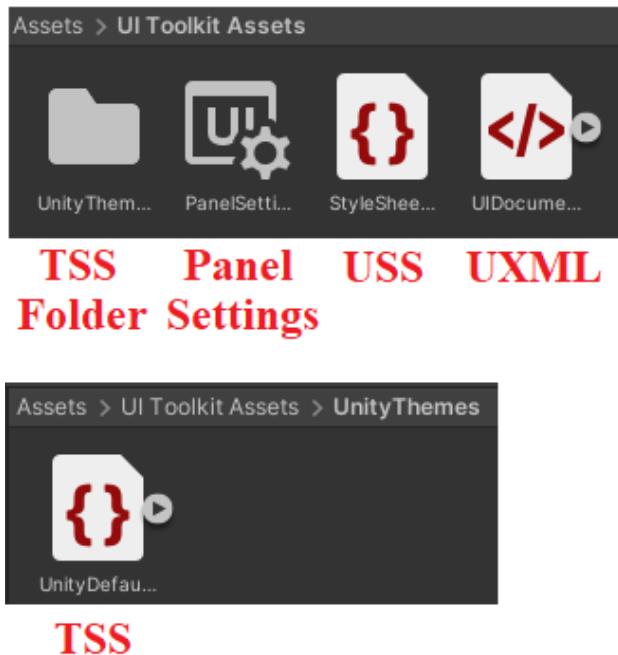


Figure 18.6: Assets used when making UI Toolkit interfaces

A **UI Document** is a UXML file type (.uxml extension). This file uses **Unity Extensible Markup Language (UXML)** to define the layout and structure of the UI. While UXML is a unity-specific markup language, it works similarly to other markup languages like HTML and XML. You can create UXML files by right-clicking in an Asset folder and selecting **Create | UI Toolkit | UI Document**, then naming the file appropriately.

A **Style Sheet** is a USS file type (.uss extension). This file is used to designate style properties that can be referenced in a UXML file. This works very similarly to a CSS file when working with HTML. You can create USS files by right-clicking in an Asset folder and selecting **Create | UI Toolkit | Style Sheet**, then naming the file appropriately.

A **Panel Settings** asset (.asset extension) defines the collection of properties that the UI will have.

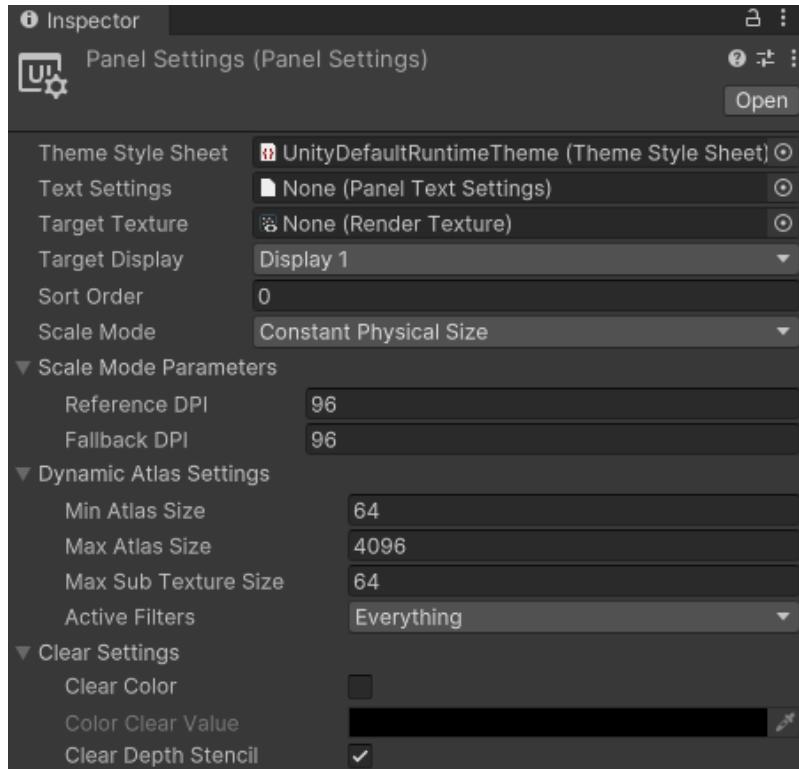


Figure 18.7: A default Panel Settings asset's properties

A **Theme Style Sheet** is a TSS file type (.tss extension). It determines which Panel Settings will be used with which TSS and USS files, by maintaining a collection of them, as shown in the following figure.

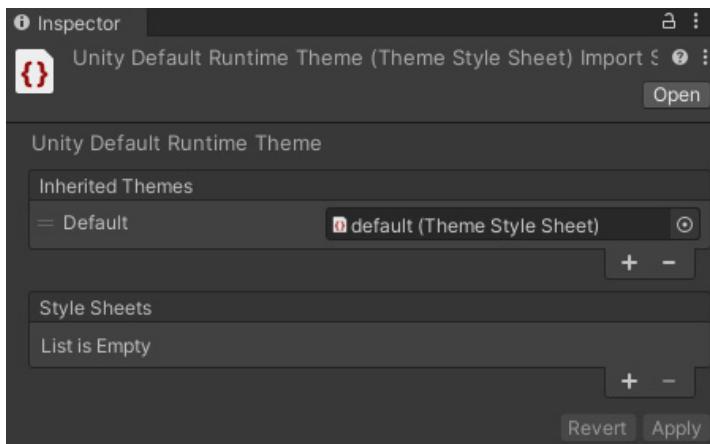


Figure 18.8: The Inspector of a TSS file

When you create a Panel Settings asset, a folder called `Unity Themes` is automatically created in the same folder. Within it, a single TSS file called `UnityDefaultRuntimeTheme.tss` will be created for you. However, you can create more TSS files by right-clicking in an Asset folder and selecting **Create | UI Toolkit | TSS Theme File**.

The final piece of the UI Toolkit system is the **UI Document** component, shown in the following screenshot.

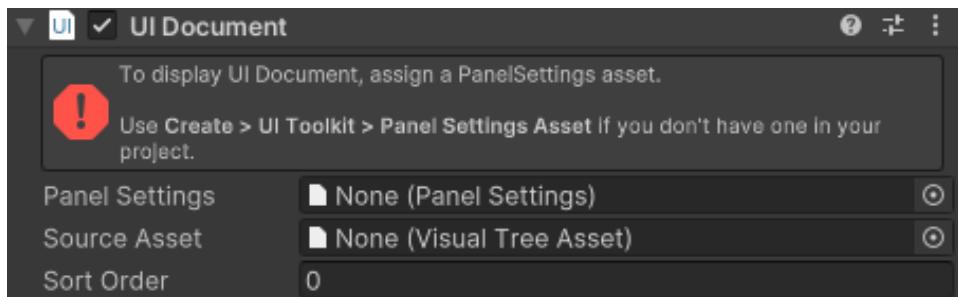


Figure 18.9: The UI Document component

This component is added to a GameObject in your scene and allows your UI created with the UI Toolkit to be rendered. It has three properties: **Panel Settings**, **Source Asset**, and **Sort Order**.

You assign a Panel Setting asset to the **Panel Settings** property to signify which settings the UI rendered by this component will have. The **Source Asset** property is where you assign the UI Document (UXML) file. The **Sort Order** property determines what order the UI defined by the UXML file will render relative to any other UXML files you are rendering in the scene that use the same Panel Settings. You can add this component to a GameObject by using **Add Component** in the **Inspector** and searching for **UI Document**.

Now that we know all the required assets and components, let's review how to actually build these assets using the UI Builder.

Visual Elements and UI Hierarchy

In the same way that a uGUI-built UI is comprised of multiple GameObjects, a UI Toolkit-built UI is comprised of **Visual Elements**. There are multiple UI elements available to you in the UI Toolkit system (buttons, labels, sliders, etc.), but the Visual Element is the base class for all of them, so they all derive many of their properties from it.

Just as GameObjects are organized in the Unity Editor Hierarchy, UI Toolkit organizes its Visual Elements in something called a **UI Hierarchy**. For example, let's say I'd like to create a UI similar to the one displayed in *Figure 18.10*.



Figure 18.10: Simple sample UI

The Scene Hierarchy in the Editor, if it were built with uGUI, is quite similar to the UI Hierarchy, if it were created with UI Builder, as shown in *Figure 18.11*.

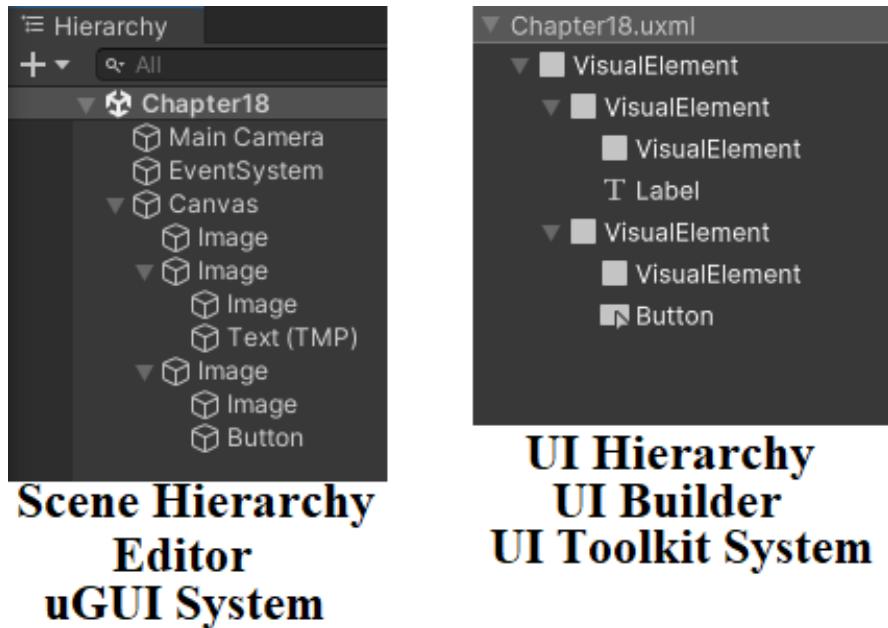


Figure 18.11: Scene Hierarchy for uGUI versus UI Hierarchy for UI Toolkit

Except for some slight changes in naming and nesting, they are almost the same. Where you can consider the UI Document, represented by `Chapter18.uxml` in the UI Hierarchy acting with a similar function to the Canvas in the Scene Hierarchy.

Note

The UI Hierarchy screenshot was obtained from the UI Builder tool, which we will discuss in the next section.

When working with uGUI, the alignment and position of each GameObject is based on the parent object it is nested under. This is true also for Visual Elements created with the UI Toolkit system. So, having a sense of how items should be nested is important for developing UI with the UI Toolkit system.

Note

You can find the example UI that was developed for *Figure 18.10* and *Figure 18.11* within the Unity project provided in the code bundle. They can be found within the scene labeled `Chapter18.asset`. In the Scene Hierarchy, the `Canvas` GameObject holds the uGUI version and the `UIDocument` GameObject holds the UI Toolkit version.

Now that we've reviewed some basic concepts of the UI Toolkit system, let's look at how we can actually build UI with it.

Creating UI with the UI Builder

To create UI with the UI Toolkit, you have to create a UI Document to describe what Visual Elements you will render as well as their layout and other properties. There are two ways to create UI Documents within the UI Toolkit system:

- Write the code that will lay out the Visual Elements
- Lay the elements in UI Builder and have it write the code for you

You can also do a combination of both and bounce between editing your UI via code or the UI Builder.

To access the UI Builder, select **Window | UI Toolkit | UI Builder**.

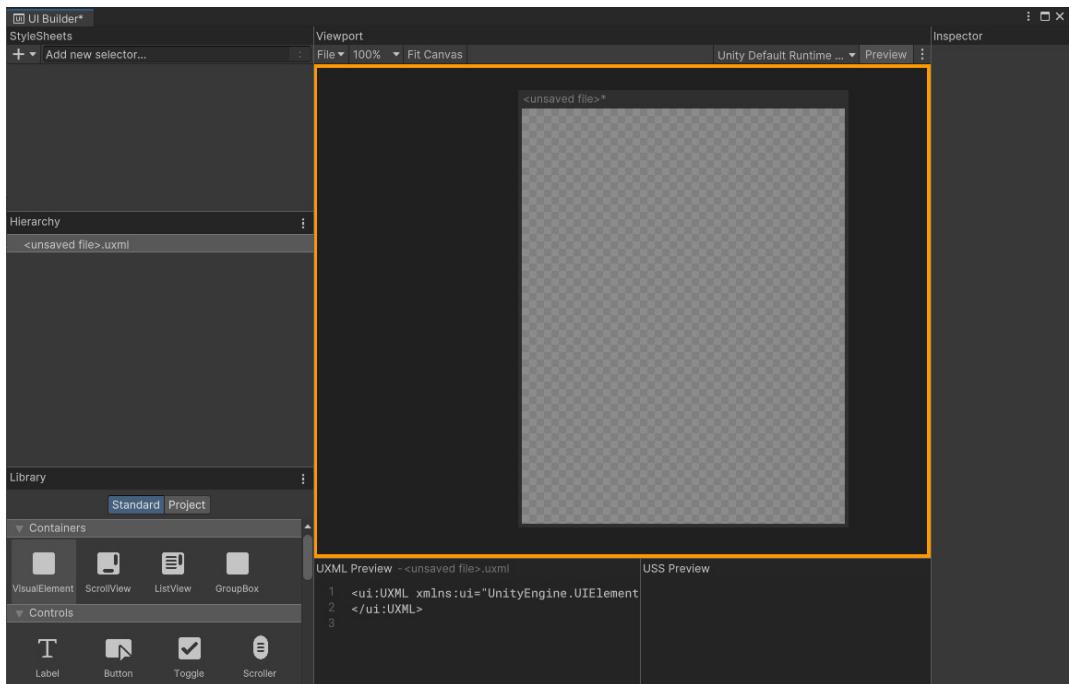


Figure 18.12: The UI Builder window

In this window, you can drag and drop various Visual Elements from the **Library** (bottom left) into the **Viewport** (center). You can resize and move the Visual Elements around in this Viewport, as well. You can change the parenting, thus changing how the Visual Element is aligned via the **Hierarchy** (left center). Additionally, you can also view the code file that is generated from your layout in the **UXML Review** Panel and the code files that represent your styles in the **USS Preview** Panel (both are in the bottom center). An example of walking through how to build a UI with the UI Builder is in the *Examples* section at the end of this chapter.

To be able to use the UI layout you've created with the UI Builder, you must save the UI Document as a .uxml file. Pressing **Ctrl + S** will save the file for you in a location of your choosing. I recommend you save them in a folder called **UI Toolkit** within **Assets**.

Now that we've reviewed how to create a UI Document, let's review how to tie the document to your game's scene.

You can rename any of your Visual Elements by double-clicking on them in the Hierarchy and typing a new name. When using the UI Builder, the names of the Visual Elements will act as variables if you need to reference them by C# code. So, it's important to give the Visual Elements distinct names if you plan to access them via C# code (see the *Making the UI Interactable with Events* section for further information).

Using the UI Document component

As mentioned in the *Parts of the UI Toolkit System* section of this chapter, the UI Document component must be added to a GameObject in your scene so that the UI you've created can be rendered. You can do this by either adding the UI Document component to an existing GameObject or selecting + | UI Toolkit | UI Document. This GameObject will have a **Transform** component and a **UI Document** component.

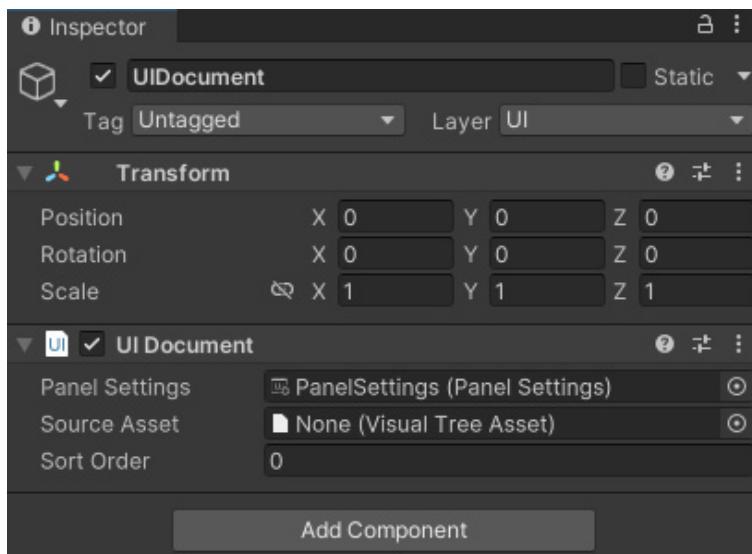


Figure 18.13: The Inspector of a UI Document GameObject

If you do not already have a folder called `UI Toolkit` within your project's `Assets` folder, one will automatically be created for you. Within it, you will find a `PanelSettings` asset, and a folder called `UnityThemes`. Within `UnityThemes`, you will find a TSS file called `UnityDefaultRuntimeTheme.tss`. Your `UIDocument` GameObject's **UI Document** component will automatically have the `PanelSettings` asset assigned to the **Panel Settings** property. However, you will need to hook your UI Document into the **Source Asset** property. Drag and drop whatever `.umx1` file you created into the **Source Asset** property to view the UI you created in your scene.

Now that we know the basics of building and rendering our UI, let's look at how to code the interactions of UI created with the UI Toolkit.

Making The UI interactable with C#

The UI Builder and your UXML document only handle the visual properties of your UI. While you can assign some basic responses (like changing visuals on hover) through the use of style sheets, you will need to create C# scripts to handle any logic or events related to your UI.

The `UIElements` namespaces

To be able to write code that interfaces with UI Documents, you must use the `UnityEngine.UIElements` namespace. If you are using the UI Toolkit to make editor UI, you may also need the `UnityEditor.UIElements` namespace.

Note

For more information about the `UnityEngine.UIElements` and `UnityEditor.UIElements` namespaces, see the following resource: <https://docs.unity3d.com/Packages/com.unity.ui@1.0/api/UnityEditor.UIElements.html>.

Getting a reference to UI Documents variables

Within your C# script, you can create a reference variable to a `UIDocument` type. You can assign this variable in the same way you would generally assign a variable of another class: assigning it in the **Inspector**, using `GetComponent`, passing a reference to it from another script, and so on.

Note

While you can use `FindObjectOfType`, this is not recommended for finding a reference to a `UIDocument`, as you will very likely have multiple UI Documents in your scene.

After you get a reference to the `UIDocument` you want to work with, you can get a reference to the Visual Elements within it (e.g., the buttons, labels, etc.) by getting a reference to the root Visual Element on the UI Document, then Querying that element for the Visual Element type by its name.

For example, in the UI shown earlier in the chapter, I have renamed the Label to `DogLabel` and the Button to `CatButton`, as shown in the following figure:

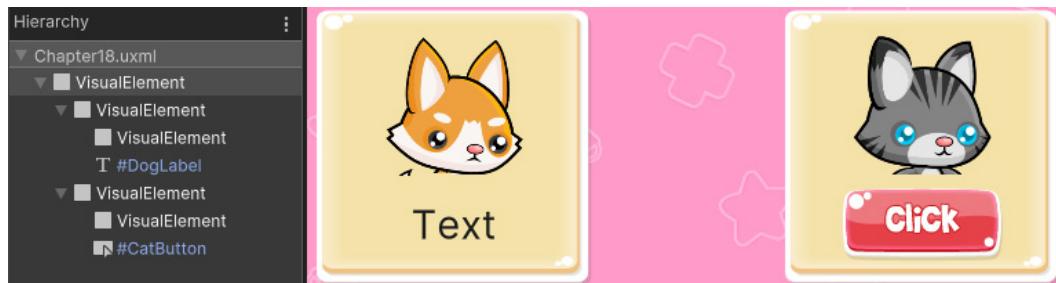


Figure 18.14: New names for Label and Button

To create a reference to these variables in a C# script, I first created reference variables for the UIDocument, the Label, and the Button, with the following code:

```
[SerializeField] private UIDocument uiDocument;
private Label dogLabel;
private Button catButton;
```

The uiDocument variable will be assigned in the Inspector. As with all MonoBehaviour scripts, this script needed to be attached to a GameObject in the scene. I attached this script to the UIDocument GameObject and then dragged the UIDocument GameObject into the **Ui Document** property.

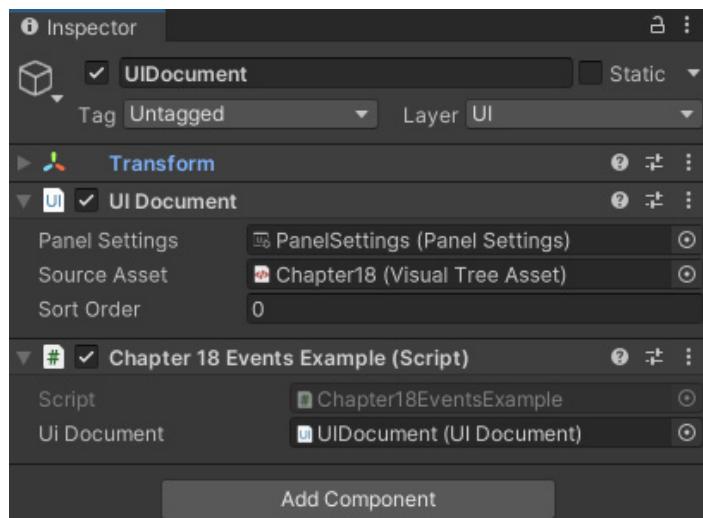


Figure 18.15: Assigning the UIDocument to the script

It's important to note that when you reference a UIDocument in C#, you are referencing the UIDocument component, not a UI Document source file.

To initialize dogLabel and catButton, I got a reference to the rootVisualElement on the uiDocument, then used the Query method to find each Visual Element.

```
void Start() {
    var root = uiDocument.rootVisualElement;
    dogLabel = root.Q<Label>("DogLabel");
    catButton = root.Query<Button>("CatButton");
}
```

Notice how I used `root.Q` to find the Label and `root.Query` to find the Button. You can use either one, and I only did this so you can see that there are two different ways to do it. It is up to your preference which one you use.

Managing Visual Element events

Since the UXML document does not manage events, you can find out if the UI defined in the UXML document is interacted with by either subscribing to an event or registering a callback method when the event is triggered.

For example, if I want to log a message in the Console when the `catButton` is clicked, I need to create a method that is subscribed to the `catButton`'s `clicked` event.

First, I add the following method to my script:

```
private void OnCatButtonClicked() {
    Debug.Log("CatButtonClicked");
}
```

Then, I need to subscribe to the `catButton`'s `clicked` event, with the following in the `Start()` method.

```
catButton.clicked += OnCatButtonClicked;
```

To avoid any errors with the event subscription when the `GameObject` is disabled or destroyed, I unsubscribe from the event in the `OnDisable()` method.

```
private void OnDisable() {
    catButton.clicked -= OnCatButtonClicked;
}
```

This causes the `OnCatButtonClicked` message to display in the Console whenever you click the button.

Accessing Visual Element properties

Similar to the way uGUI `GameObjects` can have their properties changed via C# code, a Visual Element can have its properties adjusted via C# code, as well. For example, changing the `OnCatButtonClicked()` method to the following will make the word `Text` change to `Meow!`

```
public Camera theCamera;

void Update()
{
    transform.LookAt(2 * transform.position - theCamera.transform.position);
}
```

The preceding code causes the UI to appear as follows:

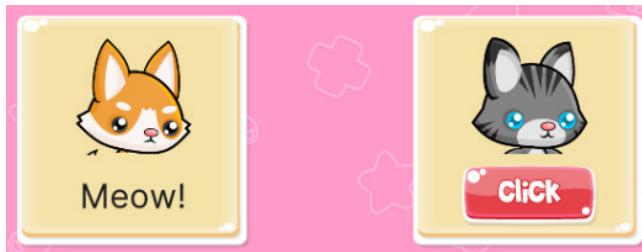


Figure 18.16: The UI after the text changes via a button click

There is a lot more I could say about working with the UI Toolkit, but as I said at the beginning of this chapter, I could only give a general overview—otherwise, I'd have to write a whole separate book just on the UI Toolkit! However, I believe I've given enough of an overview of the important concepts so that you can dive into working with the examples in the next section.

Examples

Now that we have the basic building blocks we need to get started with actually creating some UI with UI Toolkit. I didn't dive deep into style sheets or the properties of the various Visual Elements, but I'll show some examples that will expand upon that here. If after completing these examples, you want to learn more, see the *Resources* section for a bunch of examples and documentation.

When using the UI Toolkit to develop UI, there are two main parts to the work:

- Laying out the UI with the UI Builder (or by editing the UXML document)
- Writing C# code to add functionality to the UI

So, each of these examples will be broken into two parts: one for building the UI and the other for writing the functionality.

Because the UI Toolkit can be used for both Editor and runtime UI, I'll show you one example of both. Let's start with the Editor example!

Using the UI Toolkit to make an Editor virtual pet

While the primary focus of this book is on runtime UI, I would be remiss to show you how to use the UI Toolkit to make some Editor UI, since that's one of its main benefits. Editor UI is usually used for tools to facilitate development, improve productivity, streamline your workflow, and so on. Originally, I planned to show you how to make some generic Editor tools that you could expand to improve your workflow. However, I decided instead to make an Editor tool that can improve your mood. Self-care is essential to productivity, after all! While this example might not have any practical purposes in the traditional sense, it's fun and it shows you how to use lots of Editor UI features, so it's a win-win.

This example covers how to make a virtual pet that hangs out with you in a window in your Editor and compliments you when you click on it. This also demonstrates how to create animated sprite sheets within the Unity Editor.

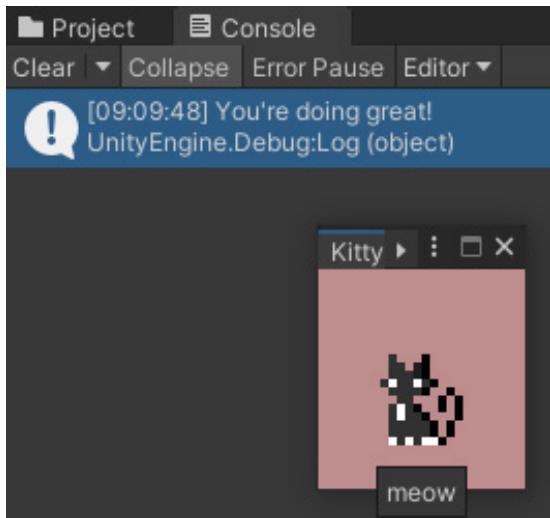


Figure 18.17: The Editor virtual pet created in this example

The cat sprite used in this example is derived from the public domain art asset provided here: <https://opengameart.org/content/a-cat>.

Note

For this example, since we are making an Editor Tool that is not related to any of the work we've done so far, you might want to create a new Unity project to complete this work in. If you do, you will need to reimport any 2D UI packages.

Let's start with laying out the UI with the UI Builder!

Using the UI Builder to make our Editor Window's UI

To create the virtual pet shown in *Figure 18.17*, complete the following steps:

1. When writing code for your Editor, you should put all your editor code within an `Editor` folder. This signifies to Unity that any code within it is only to be used for the Editor and not at runtime. Create a folder in `Assets` called `Editor` and a folder within it called `Resource`.
2. Drag the `idleCat.png` sprite sheet found in the book's source files into your newly created `Assets/Editor/Resources` folder.
3. Set its **Import Settings** so that its **Texture Type** is **Sprite (2D and UI)** and **Sprite Mode** is **Multiple**.

4. Open the **Sprite Editor** and you'll notice that the images are extremely blurry. We need to edit the import settings further to fix this.



Figure 18.18: The blurry sprite sheet

5. Return to the sprite's **Import Settings**, and under the **Default** settings, set the **Max Size** to 64 and **Compression** to **None**. Also, change the **Filter Mode** to **Point (no filter)**.
6. Select **Apply** and the sprites should now be crisp pixel art cats.

7. Now, return to the **Sprite Editor** and select **Slice**. Change the Type or **Grid By Cell Size** and set the **Pixel Size** to 16×16 .

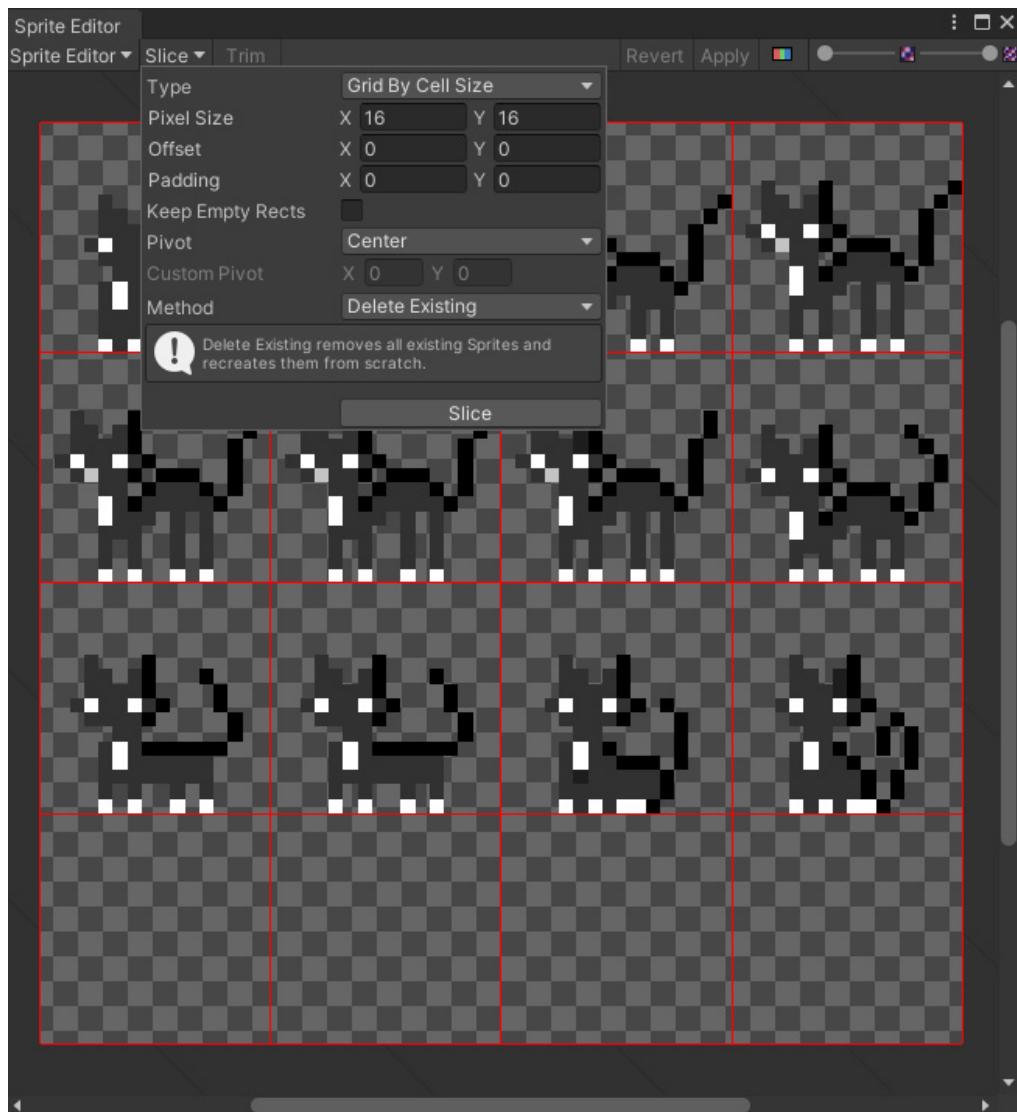


Figure 18.19: The sliced sprite sheet

8. Select the **Slice** button to confirm the changes, then select **Apply** to commit the changes. You can now close the **Sprite Editor**.
9. Now that we have our sprite imported properly, we're ready to build out the Editor UI in the UI Builder. Open the UI Builder with **Window | UI Toolkit | UI Builder**.
10. Select the `<unsaved file>.uxml` UI Document in the **Hierarchy** to bring up its properties in the **Inspector**.

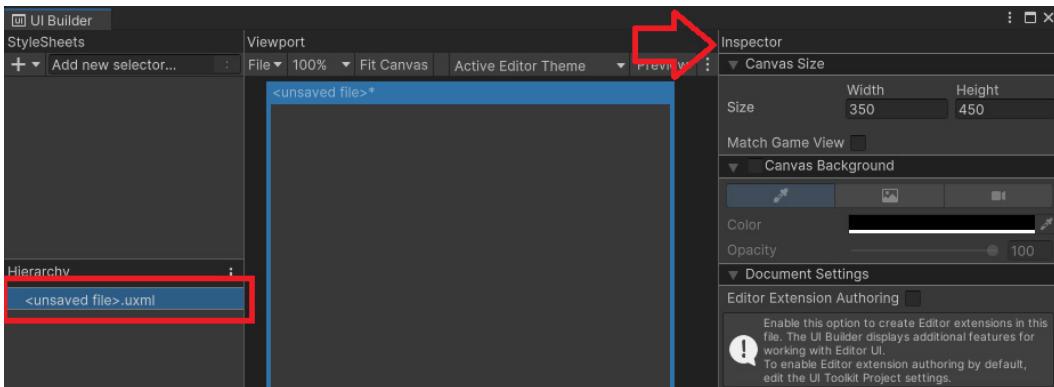


Figure 18.20: The UXML file's Inspector

11. Now change **Canvas Size** to 100 x 100.
12. Save the UI Document by selecting **File | Save** from within the **Viewport**. Save the file in the **Assets/Editor/Resources** folder and name the file `IdleCat.uxml` (the `.uxml` extension is automatically added for you).
13. Now, let's add the cat to the **Viewport**. Because we want to make the cat clickable, the easiest way to achieve this is to use a Button. Drag the icon from the **Controls Panel** to the **Viewport**. Note once you do so the Button will be positioned at the top of the container.

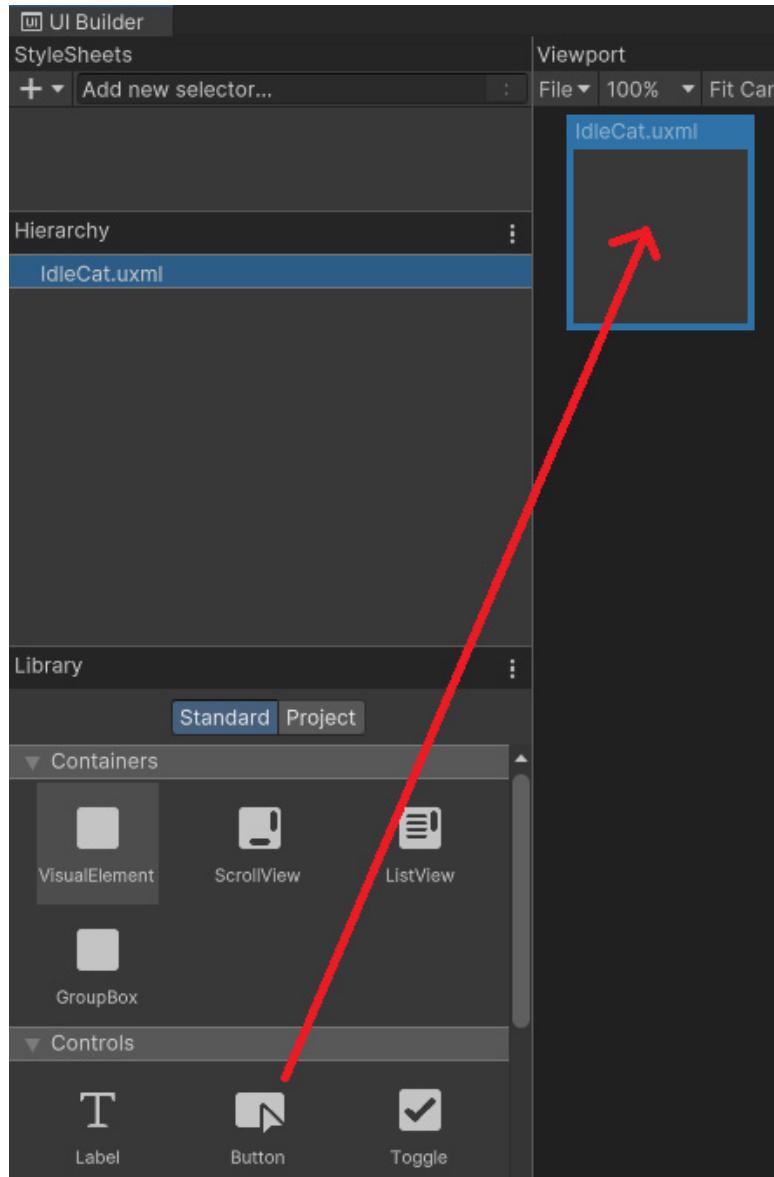


Figure 18.21: Adding a Button

14. Select the **Button** in the **Hierarchy** to open its Inspector.
15. Expand the **Background** property so that you can change the look of the button.
16. Select **Sprite** from the dropdown menu next to the **Image** property. This will allow you to select images of **Sprite** type.

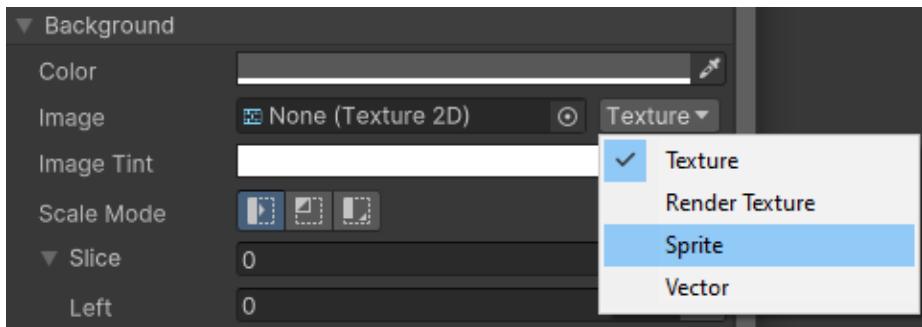


Figure 18.22: Changing the Background Image type to Sprite

17. Select the circle in the **Image** property field to search for sprites and select `idleCat_0`. You should see the following in your **Viewport** once you do so:

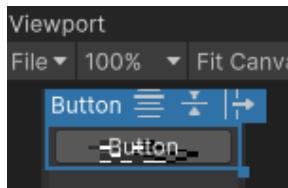


Figure 18.23: The Button with the cat image applied

18. Let's get that text out of the way. At the top of the **Inspector**, under the **Button** properties, delete the word **Button** from the **Text** property.

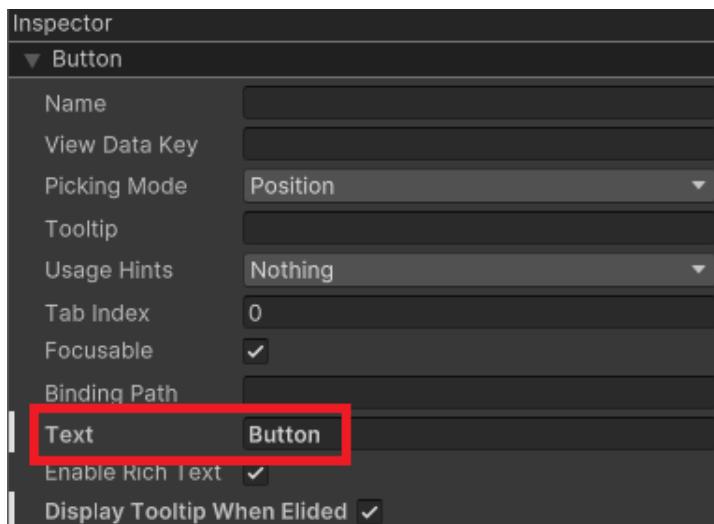


Figure 18.24: Removing the Text from the button

19. Now, let's make it the appropriate dimensions. Expand the **Size** property and change the **Size** from `auto x auto` to `64 x 64`.

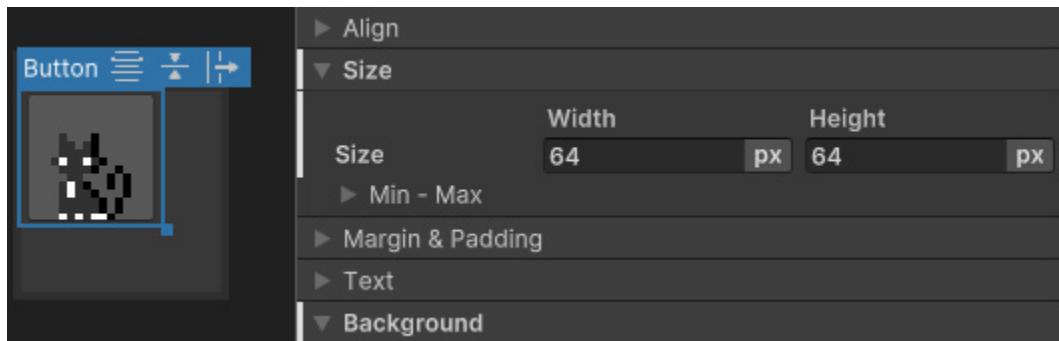


Figure 18.25: Changing the button size

20. Visual Elements always initialize in the UI Document root container at the top-left corner. If you want to be able to position them within the container, you have to create a high-level Visual Element that acts as the “shell” in which all the other items can be positioned within. So, to make this Button centered within the window, we will need to add a Visual Element to contain it. Drag **VisualElement** from the **Library** into the **Hierarchy**.

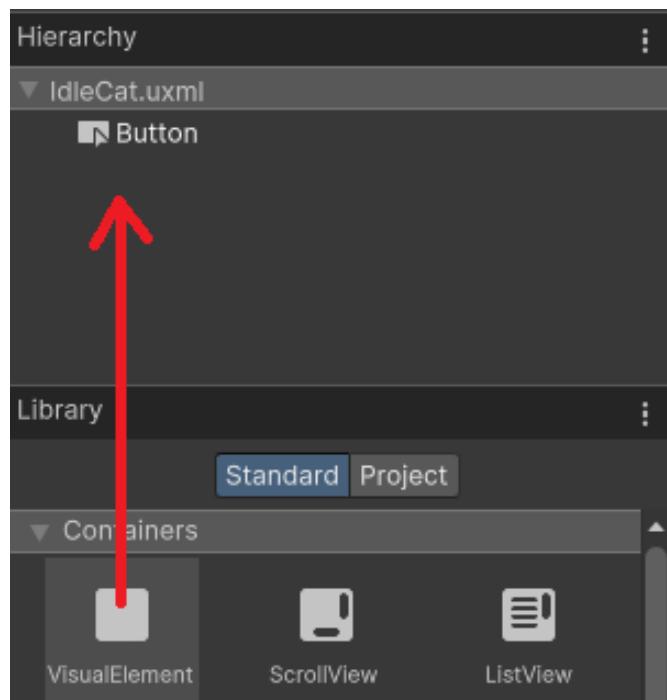


Figure 18.26: Adding a Visual Element to the Hierarchy

21. Within the **Hierarchy**, drag and drop the **Button** onto the **VisualElement**. This will cause the **Button** to be a child of the **VisualElement**.

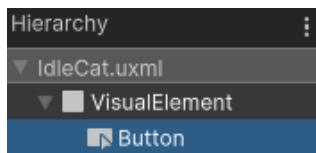


Figure 18.27: Making the Button a child of VisualElement

22. Select **VisualElement** from the **Hierarchy** to open its **Inspector**.
23. You may notice that the **VisualElement** does not fully fit within the root container. Expand the **Flex** property and change the **Grow** property to 1. This will cause the **VisualElement** to fill the whole root container.

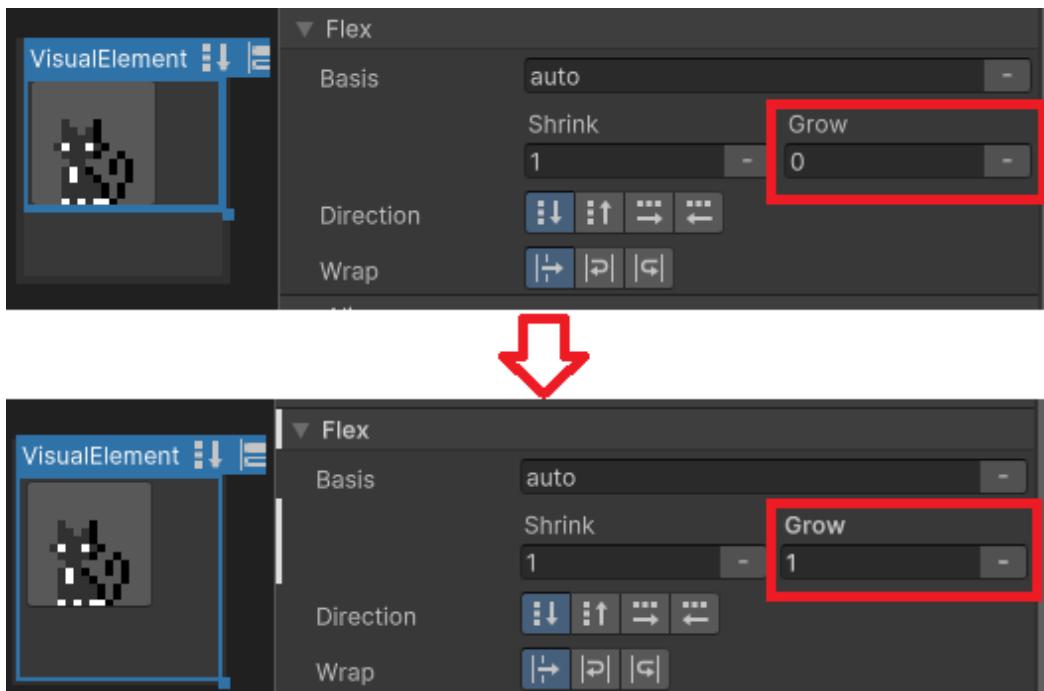


Figure 18.28: Updating the Grow property

24. Now, let's center the cat within the **VisualElement**. Expand the **Align** property of the **VisualElement** and center for both the **Align Items** and **Justify Content** properties.

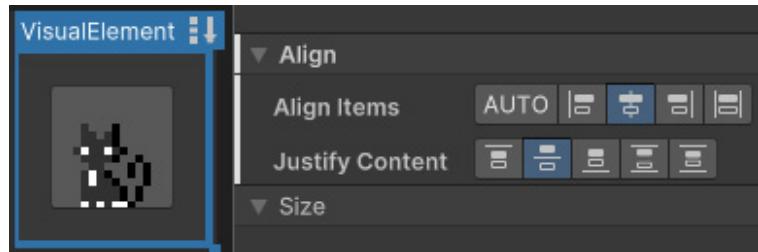


Figure 18.29: Adjusting the Align properties of the VisualElement

25. Let's change the background color of the VisualElement. In the **Background** property, change the color to BE8D8D by selecting the color block and typing BE8D8D into the **Hexidecimal** property.

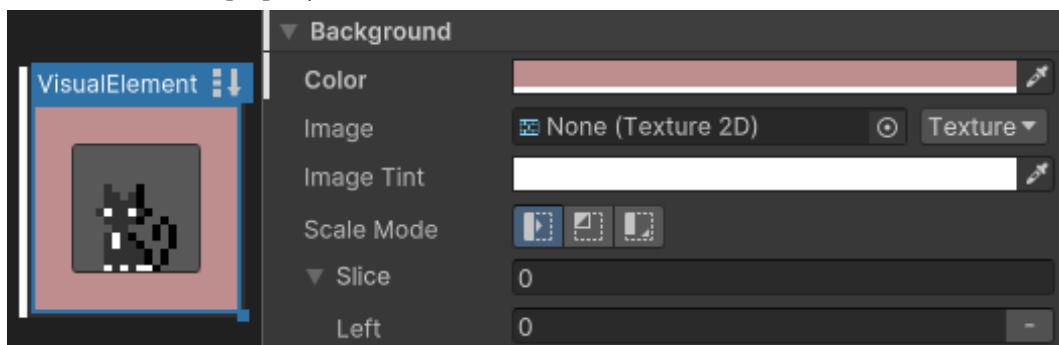


Figure 18.30: Adjusting the VisualElement's background color

26. Now, it's easy to see there's some background and border around our cat Button that we don't want. To remove this, select the Button and change its background color so that the alpha is 0.

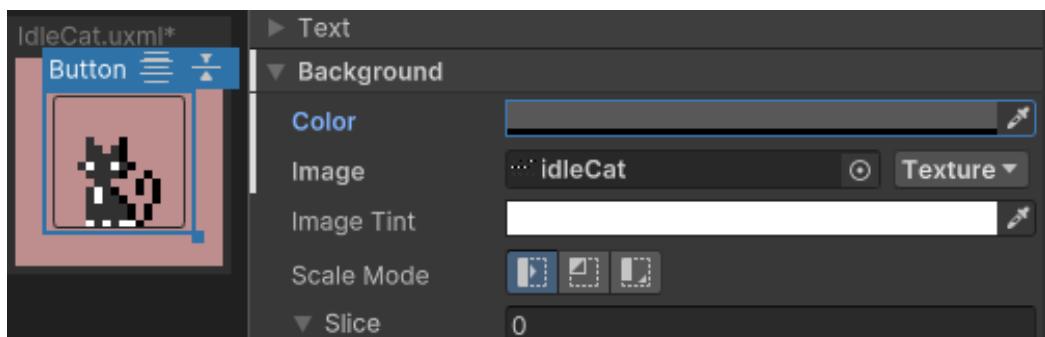


Figure 18.31: Removing the Button's background color

27. There is still a border around the Button that we don't want. Expand the **Border** property and change the color's alpha to 0. You should now have the following:



Figure 18.32: The final UI look

28. We're almost done with the UI Builder. The last thing we need to do is rename **Button** so that it will be easily accessible in our C# code. Double-click on **Button** in the **Hierarchy** and rename it **CatButton**.
29. Save your work by pressing *Ctrl + S* and we're done! You should have the following:

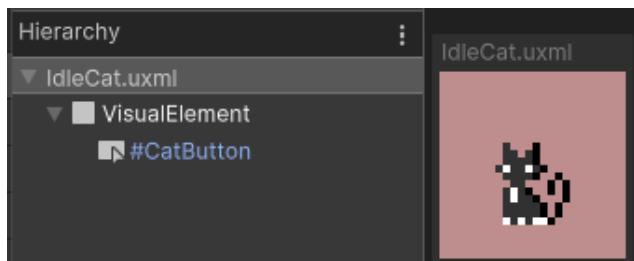


Figure 18.33: The IdleCat.uxml Hierarchy with appropriate names

Note

If a **UIDocument** **GameObject** was added to your currently open scene, you may see a warning about a **Panel Setting** missing in your **Console**. You can just delete this **GameObject** and dismiss the error message.

Now that our UI is fully laid out, we can write the code to start adding in the functionality!

Writing C# code to make our Editor Window and add functionality

To get our cat to appear as a window in our Unity Editor and have functionality, we need to write some C# Editor code. There are a few things we want the cat to do:

- Appear in a floating window when you use a **Tools** menu or use pre-defined hotkeys
- Say “meow” when you hover over it
- Compliment you in the **Console** when you click on it
- Sit with a looping idle animation
- Play a standing animation when you click on it (aka pet it)

We can achieve all of this in one C# script. To have your virtual pet appear in a window within your Unity Editor, complete the following steps:

1. Within your **Editor** folder, create a C# script called `IdleCat.cs` and open it.
2. The first thing we need to do is change our script from a `MonoBehaviour` to an `EditorWindow`. Change the class definition to the following:

```
public class IdleCat : EditorWindow {
```

This will automatically add the `UnityEditor` namespace.

3. We will have the window appear by selecting **Tools | I'm Lonely** from the Menu Bar within Unity. It will also open if we type *Ctrl + Shift + K* (*k* for “kitty”). To achieve this, write the following code in your `IdleCat.cs` script:

```
[MenuItem("Tools/I'm Lonely _%#K")]
public static void ShowIdleCat() {
    EditorWindow window = GetWindow<IdleCat>();
    window.titleContent = new GUIContent("Kitty");
}
```

In the preceding code, the `[MenuItem("Tools/I'm Lonely _%#K")]` line creates the **Tools | I'm Lonely** menu item and runs the `ShowIdleCat()` method when it is selected. `_%#K` signifies that this can also be achieved with the *Ctrl + Shift + K* shortcut.

The following lines instantiate the window and set its title to **Kitty**:

```
EditorWindow window = GetWindow<IdleCat>();
window.titleContent = new GUIContent("Kitty");
```

Save your code and you should see the menu item in your Editor. Clicking it or typing *Ctrl + Shift + K* should bring up a window named **Kitty**.

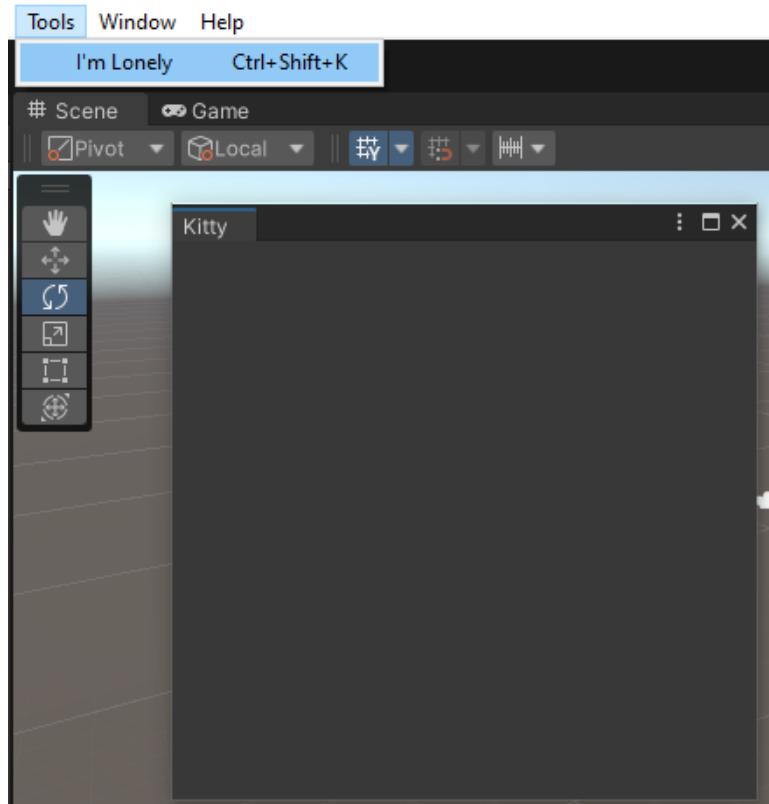


Figure 18.34: The Tools menu and the Kitty window

4. Your window may be a different size than mine. Don't worry. Let's set the size via code. Add the following two lines of code to your `ShowIdleCat()` method:

```
window.maxSize = new Vector2(100, 100);  
window.minSize = window.maxSize;
```

This will not automatically change the size of your currently open window, because this function only runs when you use the menu item or the shortcut keys. You can now resize your window by doing one of those actions now. You do not need to close the window. You can do this while it is still open.

One thing to note is that Editor Windows can be docked throughout the Unity Editor. So, the size of your window can change based on where the user docks it.

5. Now, we need to reference the UI Document we created with the UI Builder in this script. Since this is an Editor script, it can't be attached to a GameObject. That means we can't assign the variable via drag and drop or using `GetComponent`. Instead, we will have to load it via from our `Editor/Resources` folder.

When working with Editor UI, the `CreateGUI()` method is an Event Function used to initialize the UI.

Write the following code, making sure to add the `UnityEngine.UIElements` namespace to the top of the script:

```
private void CreateGUI() {
    var root = rootVisualElement;
    var quickToolVisualTree = Resources.
Load<VisualTreeAsset>("IdleCat");
    quickToolVisualTree.CloneTree(root);
}
```

This code block does a few things. First, it gets the `rootVisualElement` of the Editor Window we have created. Then, it finds the `IdleCat.uxml` file by searching for a `VisualTreeAsset` named `IdleCat` within the `Editor/Resources` folder. Then, it clones the `IdleCat.uxml` UI Document and places it within the window's root.

If you return to the Editor, you should now see your Kitty on the pink background in your open window.



Figure 18.35: The window with the appropriate size and UI Document

6. Now, we need to get access to the button. Add the following variable declaration to your script:

```
private Button catButton;
```

7. To assign the `catButton`, we need to `Query` the `root` object. Add the following line to your `CreateGUI()` method:

```
catButton = root.Q<Button>("CatButton");
```

8. Next, let's have the tooltip appear when we hover over the button. Add the following line of code to the `CreateGUI()` method:

```
catButton.tooltip = "meow";
```

After you save the script, you should be able to immediately see these changes reflected in your Editor. (Note that my screenshot tool doesn't capture the mouse cursor, so it is not shown in the following figure.)



Figure 18.36: The meow tooltip

- Now, we can have the cat compliment you when you click on it. First, let's create a new method called `OnCatButtonClicked()` that shows the "You're doing great!" message in the Console. Add the following code to your script:

```
private void OnCatButtonClicked()
{
    Debug.Log("You're doing great!");
}
```

- We need to have the `OnCatButtonClicked()` method subscribe and unsubscribe to the click event on `catButton`. Add the following line to your `CreateGUI()` method:

```
catButton.clicked += OnCatButtonClicked;
```

Also, create the following `OnDisable()` method:

```
private void OnDisable()
{
    catButton.clicked -= OnCatButtonClicked;
}
```

Once you save, you can already see the cat compliment you in the Console when you click on it.

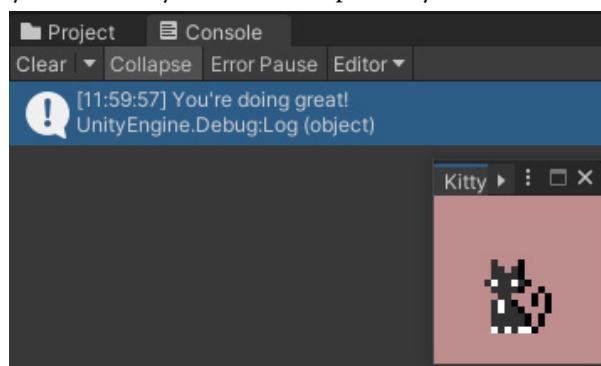


Figure 18.37: The complimenting cat

11. Our last two goals for our virtual pet involve animating it. This takes a bit of effort. We can't use a Unity animation to achieve it because we're in the Editor. So, we have to write code that will swap out the background on our `catButton` with the appropriate image via code on some kind of timer.

One way to do this is with **coroutines**, but unfortunately, coroutines do not work by default in the Editor. However, we can use coroutines in our Editor by importing the Editor Coroutines package from Unity.

To do that, select **Window | Package Manager** to open the **Package Manager**.

12. Select **Package: Unity Registry** from the dropdown to view all available Unity packages.

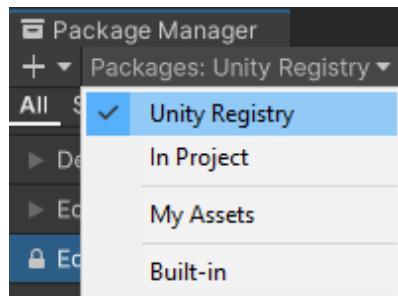


Figure 18.38: Changing which packages appear in the Package Manager

13. Now, scroll until you see **Editor Coroutines**. It will likely be locked. If it is, select **Unlock**.

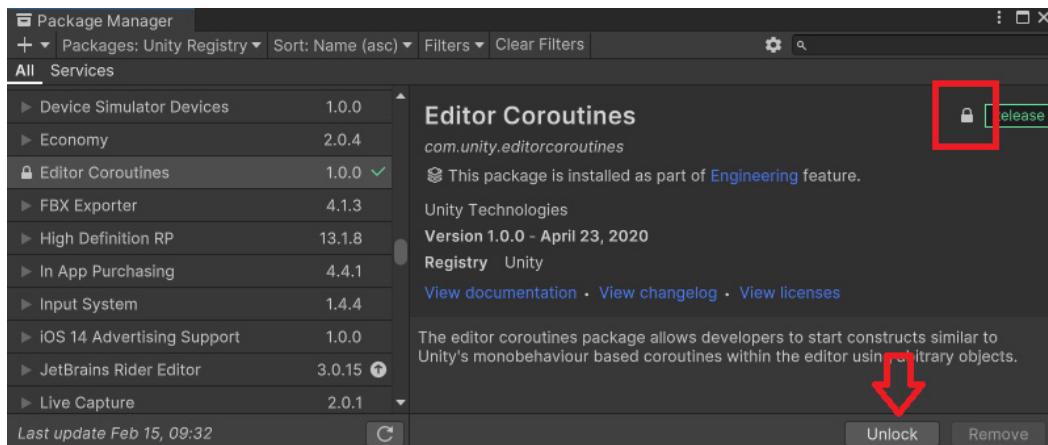


Figure 18.39: Unlocking the Editor Coroutines package

14. Now that you've unlocked the Editor Coroutines package, you can add the following namespace to the top of your `IdleCat.cs` script:

```
using Unity.EditorCoroutines.Editor;
```

Now, we can use `EditorCoroutines` in our code! They work pretty similarly to regular coroutines. But, before we write our first Editor coroutine to control our animations, let's get access to the images we'll want to use in our animation and write some helper functions.

15. I want the cat to have two animations: an idle animation and a petting animation (which will be triggered by the mouse click). I'll store the images for these animations in two separate Lists. To use them as background images, I'll need to store them in lists of `StyleBackgrounds`. Add the following to your class:

```
private List<StyleBackground> idleBackgrounds = new  
List<StyleBackground>();  
private List<StyleBackground> pettingBackgrounds = new  
List<StyleBackground>();
```

16. Before we can store the appropriate sprites in those Lists, we need to get a reference to all of the sprites from the sprite sheet. We'll do this similarly to how we found the UI Document file earlier. Add the following line of code to your `CreateGUI()` method:

```
Sprite[] allSprites = Resources.LoadAll<Sprite>("idleCat");
```

This will look in the `Editor/Resources` folder and store all of the sprites with the name `idleCat` to the `allSprites` array.

17. The following frames will represent the specific animations:



Petting Animation

Idle Animation

Figure 18.40: Which sprites go to which animation

Now, we need to loop through the `allSprites` array and divvy those sprites up into the correct list. We can do so by adding the following code to the `CreateGUI()` method:

```
for (int i = 0; i <= allSprites.Length - 1; i++)
{
    StyleBackground backgroundImage = new
    StyleBackground(allSprites[i]);

    if (i < 11)
    {
        pettingBackgrounds.Add(backgroundImage);
    }

    if (i >= 10)
    {
        idleBackgrounds.Add(backgroundImage);
    }
}
```

18. Before we can write the methods that swap out the background with the correct image, we need to add a variable that will keep track of which frame of which animation the button is currently displaying. Add the following variable initialization to your class:

```
private int animationIndex = 0;
```

19. OK, now we need to write methods that will increase the `animationIndex` and set the `catButton`'s `backgroundImage` to the appropriate sprite. Add the following method to your class to control the animation for the idle animation:

```
private void IdleAnimation()
{
    animationIndex++;
    if (animationIndex >= idleBackgrounds.Count)
    {
        animationIndex = 0;
    }

    catButton.style.backgroundImage =
    idleBackgrounds[animationIndex];
}
```

Notice that when the index is out of range, it loops back around to 0.

20. Add a similar method to control the petting animation:

```
private void PettingAnimation()
{
    animationIndex++;
    if (animationIndex >= pettingBackgrounds.Count)
    {
        animationIndex = 0;
    }

    catButton.style.backgroundImage =
pettingBackgrounds[animationIndex];
}
```

Note

The `IdleAnimation()` and `PettingAnimation()` methods have some reusable code and could be refactored for brevity; however, for clarity, I will keep it the way it is.

21. Now, we're ready to use an `EditorCoroutine` to call these methods and get the animations playing. Let's start by creating the following Editor coroutine:

```
IEnumerator NextAnimationFrame()
{
    var waitForOneSecond = new EditorWaitForSeconds(1f);

    while (true)
    {
        yield return waitForOneSecond;
        IdleAnimation();
    }
}
```

This will create an infinite loop that runs the `IdleAnimation()` method every second.

22. We need to start the Editor coroutine that we wrote in the previous step. Add the following code to your `CreateGUI()` method:

```
EditorCoroutineUtility.StartCoroutine(NextAnimationFrame(),
this);
```

23. If you return to your Editor, you should be able to see the cat flicking its tail while it loops through its idle animation. Now, we need to write some code that will make the petting animation play when you click on the cat. To do this, we need another variable. Add the following to your class:

```
private bool idle = true;
```

This variable will keep track of whether or not the cat should be idling.

24. Add the following to the `OnCatButtonClicked()` method to signify cat should no longer be idle once it is clicked:

```
idle = false;
```

25. Update the `NextAnimationFrame()` Editor coroutine so that it switches between the idle and petting animations based on the `idle` variable. The code in bold is new:

```
IEnumerator NextAnimationFrame()
{
    var waitForOneSecond = new EditorWaitForSeconds(1f);

    while (true)
    {
        yield return waitForOneSecond;

        if (idle)
        {
            IdleAnimation();
        }
        else
        {
            PettingAnimation();
        }
    }
}
```

26. Currently, if you click on the cat, it will enter the petting animation, but it will keep looping through that animation indefinitely. We need it to go back to the idle animation once the petting animation completes. Update the `PettingAnimation()` method to include `idle = true` inside the `if` statement. This will reset the `idle` variable to `true` when the animation is complete.
27. We've officially completed everything we set out to do, but there is one problem with our code. If you put a `Debug.Log` in the coroutine's `while` loop and then close the **Kitty** window, you'll see that the `Debug.Log` keeps printing in the Console! We need to stop that coroutine's `while` loop when the window closes. To do that, add another variable to your class:

```
private bool windowOpen = true;
```

28. Change the `while` loop within the `NextAnimationFrame()` coroutine to the following:

```
while (windowOpen) {
```

29. Now, we just need to set `windowOpen` to `false` when the window closes. So, add the following to the `OnDisable()` method:

```
windowOpen = false;
```

That's it! You should now have a little kitty friend that hangs out with you while you work. Click on her whenever you need a little pick-me-up.

Using the UI Toolkit to make a menu with style sheets and animation transitions

We'll look at how to use the UI Builder to create a basic menu that uses style sheets and transition animations, then we'll hook up buttons within the menu to access web data and randomly generate content in this example.

Since the last example was a virtual pet meant to bring you happiness, I decided to stick with the theme of "self-care" for this example, as well. The following figure is a screenshot of the UI we will make with a randomly generated image of a cat and a quote retrieved from the web.

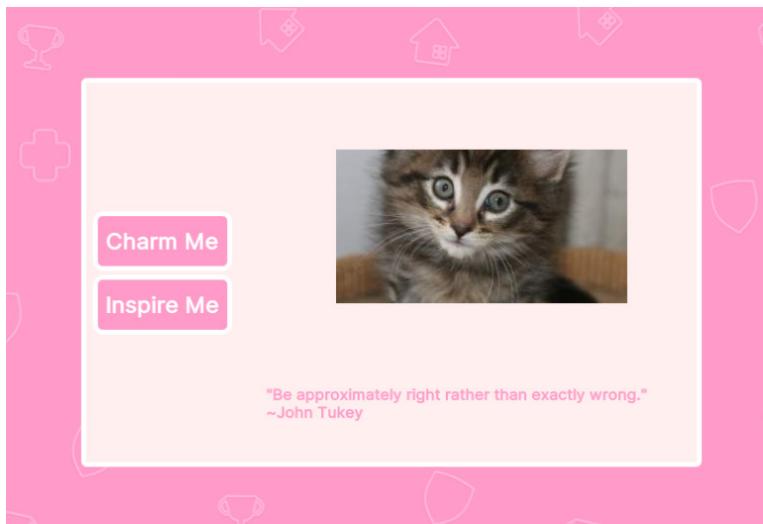


Figure 18.41: The UI menu with randomly generated content

Pressing the **Charm Me** button will randomly generate a cat picture and pressing the **Inspire Me** button will randomly generate an inspirational quote. Additionally, the buttons change color when hovered over and clicked on while animating to a slightly bigger shape.

As with the previous example, we will start by laying out the UI with the UI Builder.

Using the UI Builder to lay out a menu, make style sheets, and make animation transitions

To create the UI shown in *Figure 18.41*, complete the following steps:

1. Create a new scene called Chapter18-Examples.
2. Right-click in the Scene Hierarchy and select **UI Toolkit | UI Document**. If you don't already have one, this will automatically create a new **Assets** folder called **UI Toolkit**. It will contain a **PanelSettings** asset and a **Unity Themes** folder.
3. Within the **UI Toolkit** folder, right-click and select **Create | UI Toolkit | UI Document**. Name it **InspirationalMenu.uxml**.
4. Drag the new UXML file into the **Source Asset** property of the **UI Document** component on the **UIDocument** GameObject.
5. Just so that we have the same Game view resolution, set your Game scene view to **840x630**.

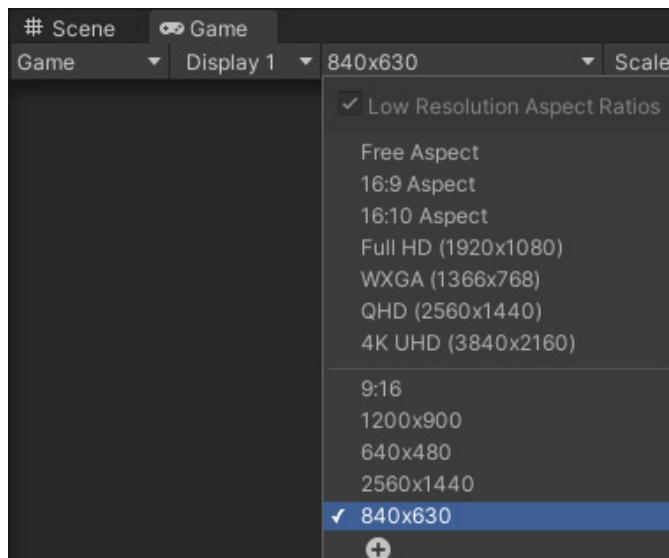


Figure 18.42: Setting your Game view resolution

6. Double-click on the **InspirationalMenu.uxml** file to open the UI Builder.
7. Select **InspirationalMenu.uxml** from the **Hierarchy** to view its **Inspector**.
8. Change the **Canvas Size** to **Match Game View**. This will cause the container in the **Viewport** to be the same size as your Game view.

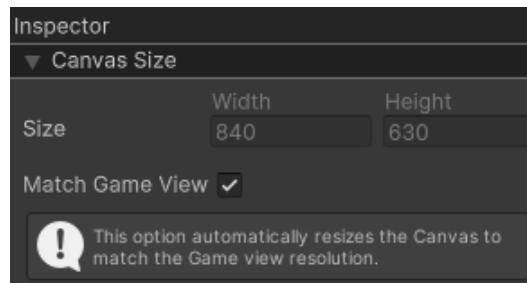


Figure 18.43: Setting the Canvas Size to Match Game View

9. Drag a **Button** from the **Controls** window into the **Viewport**. It should stretch all the way across the container.
10. The **Button** should have a blue outline around it and a handle in its bottom-right corner. Select that handle to resize the button to a pleasing “button-like” size. It’s up to you what size.

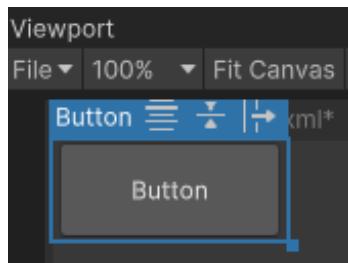


Figure 18.44: The Button element resized in the Viewport

11. Change the text on **Button** to **Charm Me** by adjusting the **Text** property.

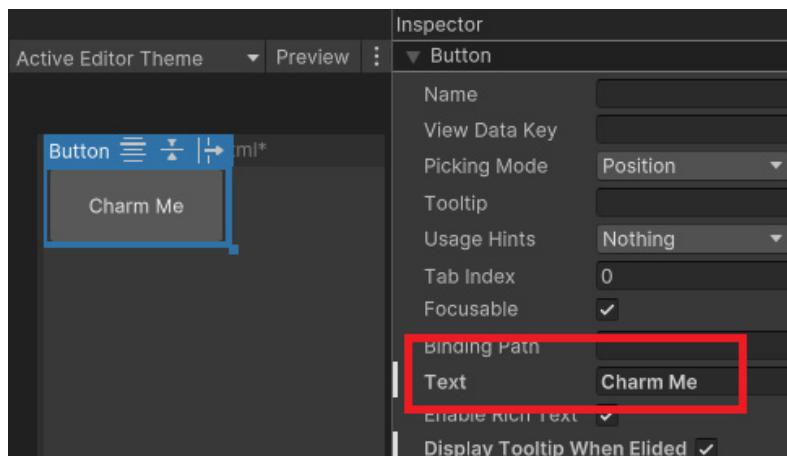


Figure 18.45: How to change the text that displays on the button

12. Expand the **Text** property and change the **Font Style** to **Bold**, the **Size** to 22, and the **Alignment** to center vertically and horizontally. Drag the handle of the button to make it a bit bigger or smaller as needed to fit the new text size.

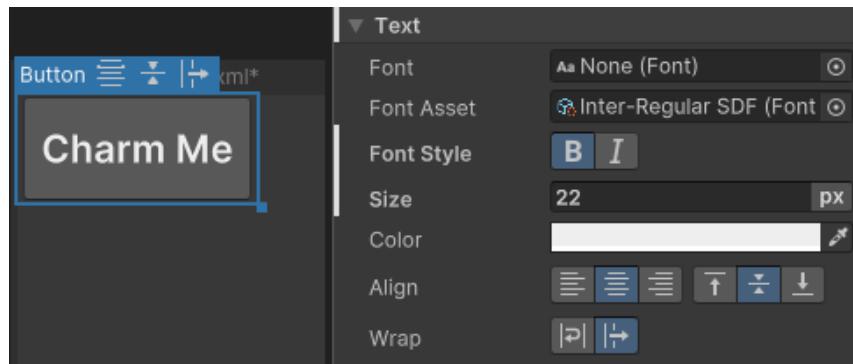


Figure 18.46: The Text settings of Button

13. Expand the **Background** property and set the **Color** to FF9BC8.

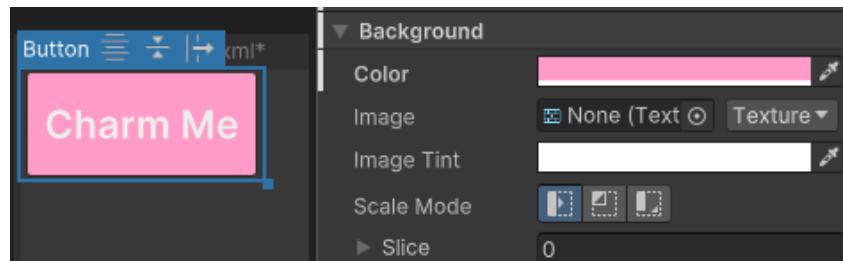


Figure 18.47: Changing the background color of the button

14. Expand the **Border** property and set the **Color** to white, the **Width** to 5, and the **Radius** to 10.

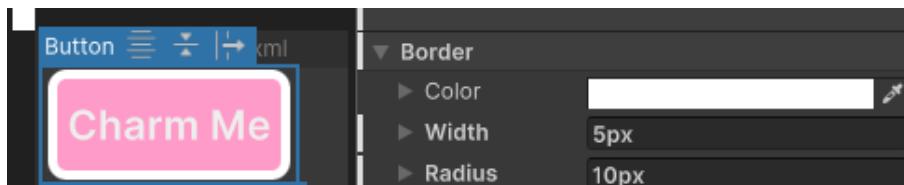


Figure 18.48: Adjusting the Border properties

15. If you go to your **Game** view, you'll notice that the colors aren't exactly the same as they are in the UI Builder.



UI Builder Viewport

Figure 18.49: The difference between the UI Builder Viewport and Editor Game View

To make sure what you are seeing in the UI Builder matches your **Game** view, select **Unity Default Runtime Theme** from the dropdown in the top-right corner of the UI Builder Viewport.

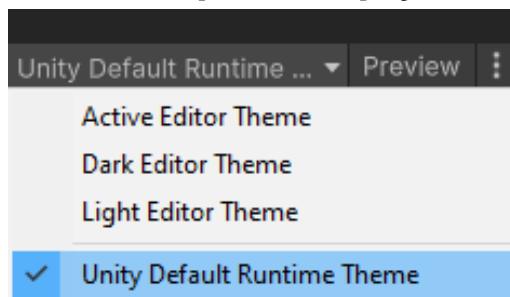


Figure 18.50: Setting the Viewport to Unity Default Runtime Theme

16. Change the color of the Button's **Text** to white.

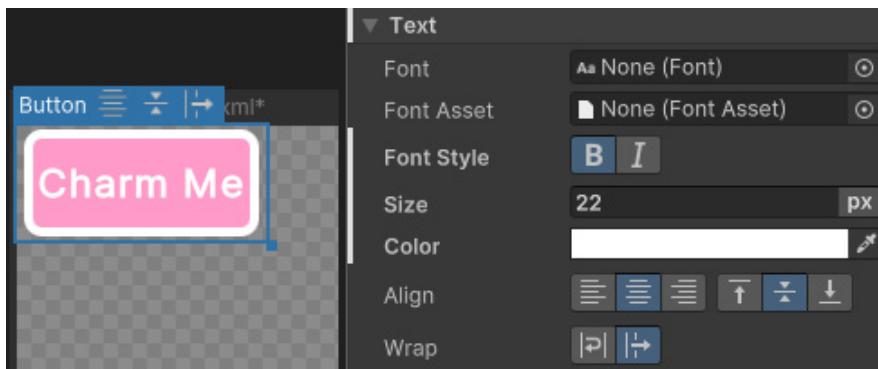


Figure 18.51: Changing the Text color

17. Now, let's create a Panel that will hold the Button. We will create the second button momentarily.
Drag a **VisualElement** from the **Containers Library** to the **Viewport**.
18. In the same way that you resized the Button, resize the VisualElement to a reasonable Panel size.

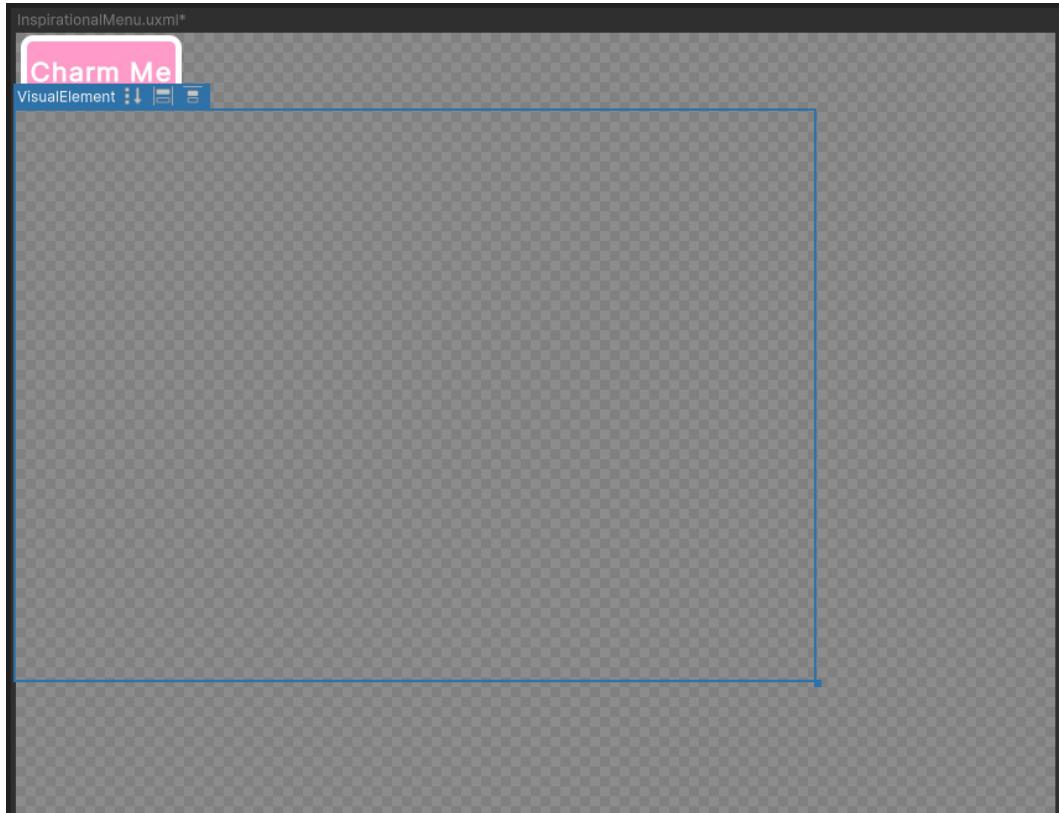


Figure 18.52: Resizing the VisualElement

19. Set the background color of the VisualElement to FFEFEF.
20. Set the **Border** color to white with a **Width** of 5 and a **Radius** of 6. You should see the following in your Viewport:

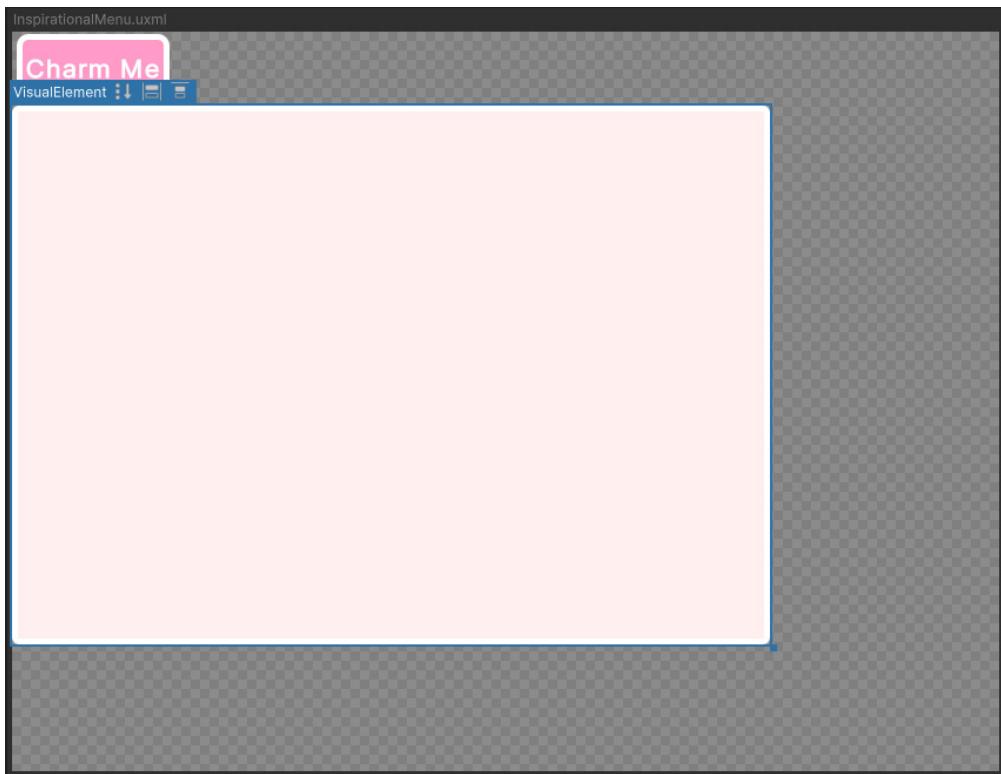


Figure 18.53: The Panel with its new settings

21. Now, let's make the Button a child of the VisualElement Panel. In the **Hierarchy** drag and drop Button onto VisualElement.

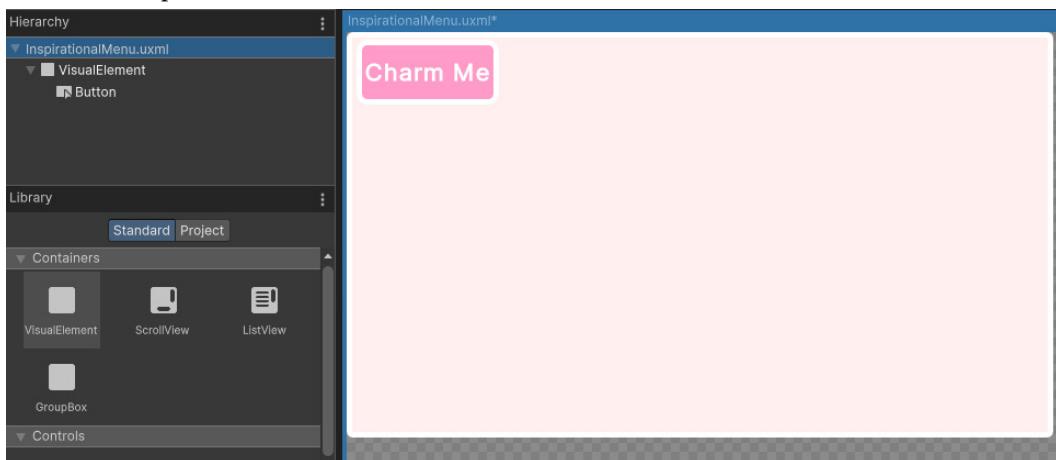


Figure 18.54: Parenting the Button to the VisualElement

22. I want to have two sections of the Panel: the left section that holds the buttons and the right section that holds the image of the cat and the inspirational quote. Add a `VisualElement` as a child of the `VisualElement` that represents the Panel. You'll notice that it automatically stacks under the `Button` in the Viewport.

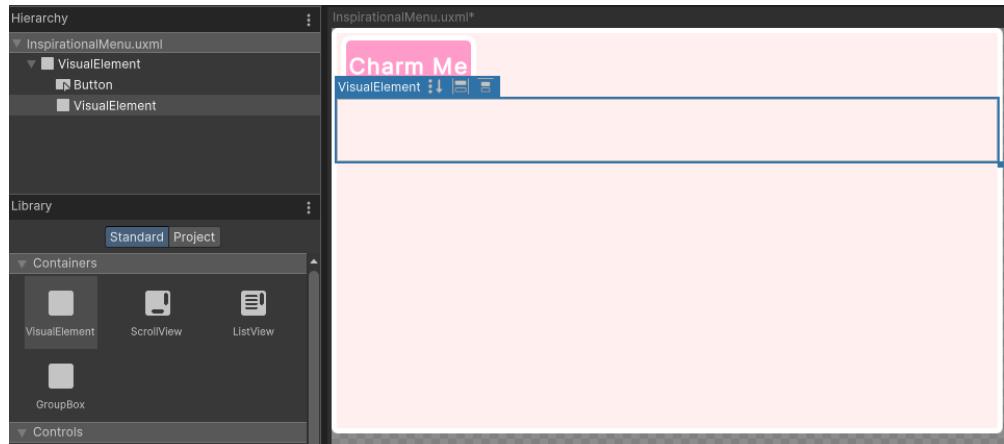


Figure 18.55: Adding in a new `VisualElement`

23. Make the `Button` a child of the new `VisualElement`.
24. Now, resize the `VisualElement` so that it takes up a portion of the left side of the Panel by clicking and dragging its blue handle. You should have something that looks like the following:

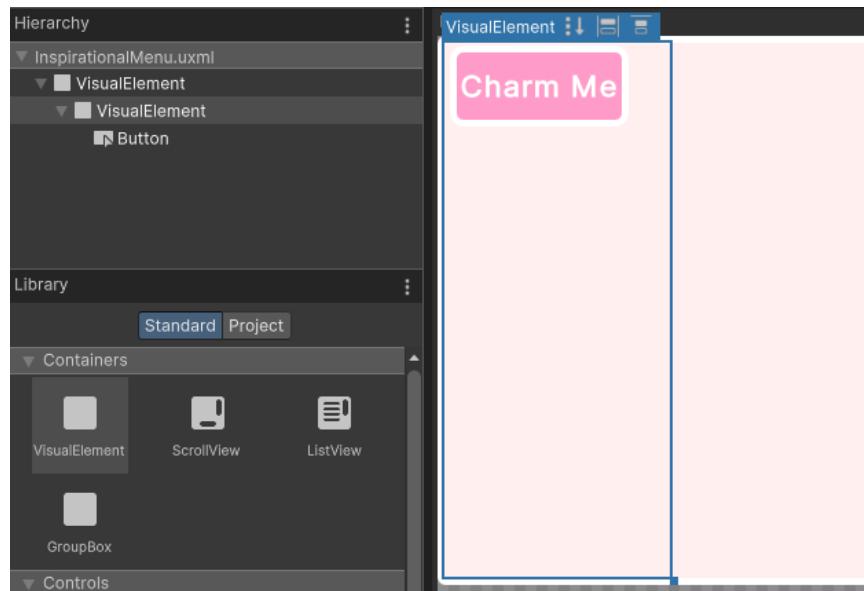


Figure 18.56: Resizing the `VisualElement`

25. Now, add another VisualElement as a child of the highest-level VisualElement. You will see something like the following:

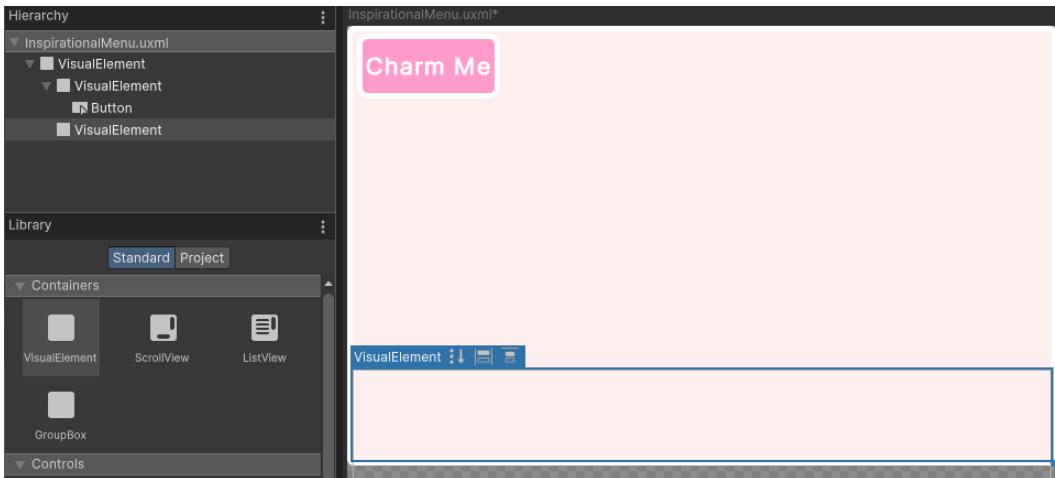


Figure 18.57: Adding another VisualElement to the Hierarchy

26. Select the top-most VisualElement and change the **Flex Direction** to row. You can do this by either selecting the Flex button on the top of the VisualElement or in the Inspector.

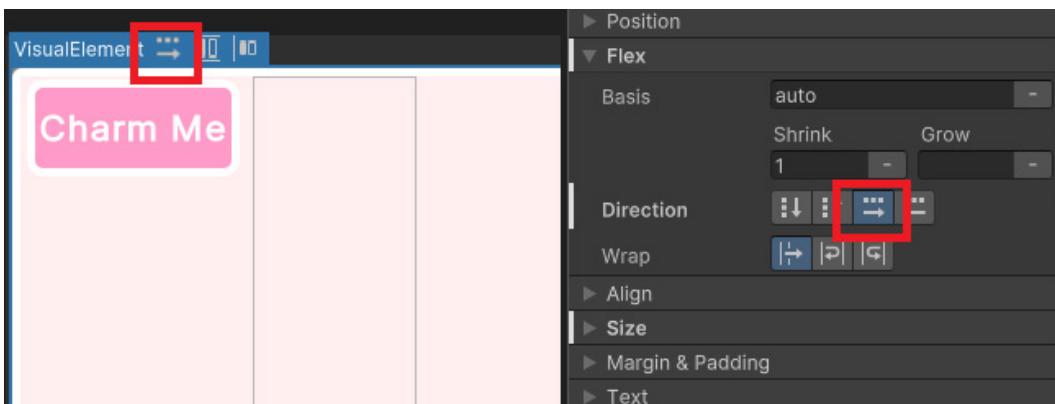


Figure 18.58: Setting the Flex Direction to row

Doing this will cause its two VisualElement children to line up left to right rather than top to bottom.

27. Select the bottom-most VisualElement from the Hierarchy. In its **Flex** property within the Inspector, change the **Grow** property to 1. This will cause the VisualElement to fill up the available space within its parent.

28. Now, we want to make the Panel centered on the screen. To do this, we need to create another `VisualElement` that will hold it so that we can center it within its parent. Add another `VisualElement` to the Hierarchy.
29. Drag the `VisualElement` that represents the Panel in the Hierarchy so that it is a child of the new `VisualElement`. In the following figure, the elements encircled by the red rectangle in the Hierarchy are the elements that represent the Panel and its children.

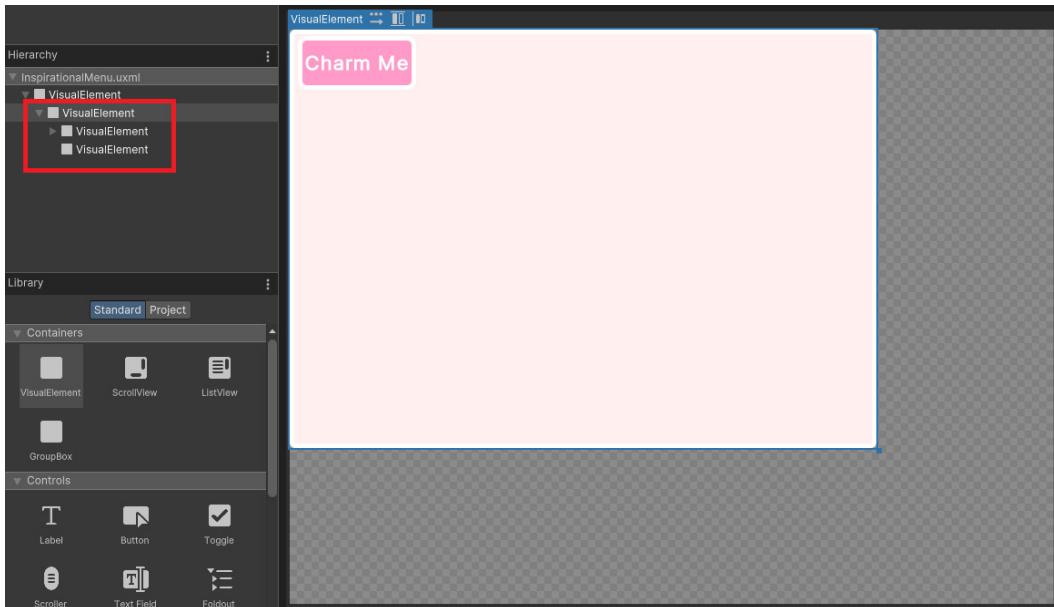


Figure 18.59: The Hierarchy of our current layout

30. Select the top-most `VisualElement` in the Hierarchy. You'll see it doesn't fill the entire root container. Set its **Flex Grow** property to 1 so it will fill the available space.
31. Expand the **Align** property and select center for both the **Align Items** and **Justify Content** properties. The Panel should now be centered in the container.
32. With the top-most `VisualElement` still selected, change the background image by expanding the **Background** property, changing the **Image** type to **Sprite**, and assigning `pinkBackground` to the **Image**. You should see the following at this point:

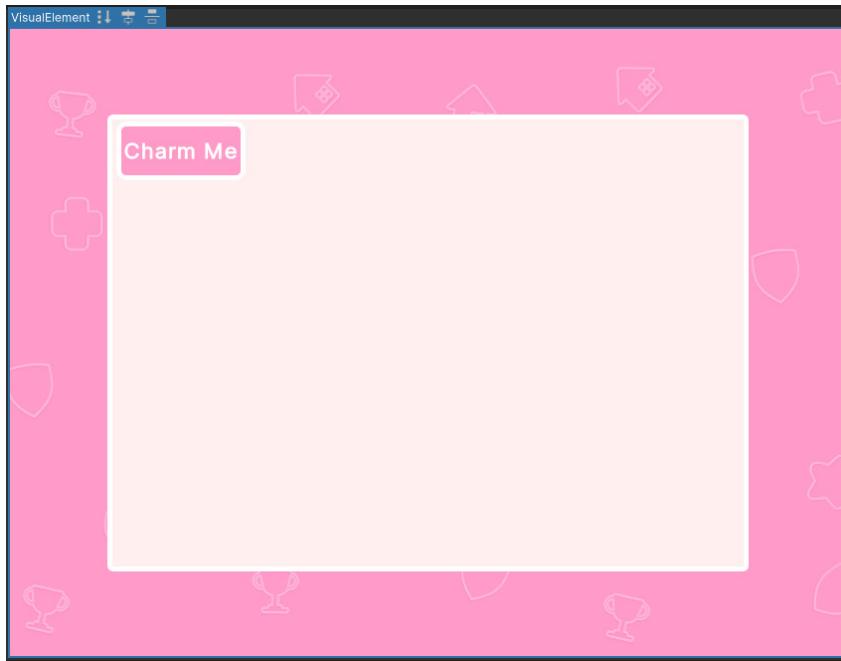


Figure 18.60: Setting the background image on the background

33. Let's add the second button to the Panel. Drag a **Button** from the **Controls Library** to the **Hierarchy** so that it is a sibling to the other Button. You should see something like the following:

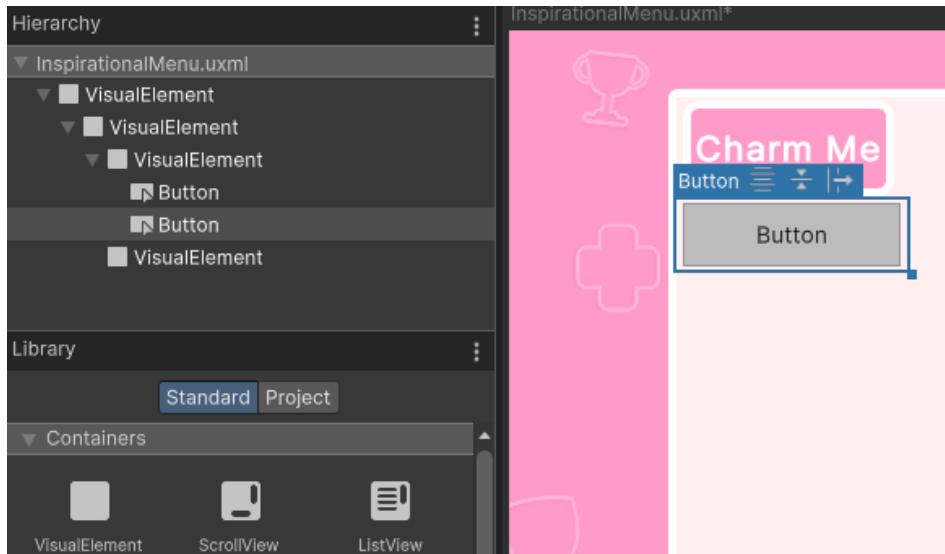


Figure 18.61: Adding a new Button

34. Instead of manually setting all the properties of the new Button to match the other, we can use a style sheet applied to both. This will make sure they both always have the same properties. Select the first Button (the one with the properties we want) and expand its **StyleSheet** property in its **Inspector**.

We can create a style sheet directly from its properties by putting a name within the textbox and then selecting **Extract Inlined Styles to New Class**.

Type `button-class` into the textbox and then press the **Extract Inlined Styles to New Class** button.

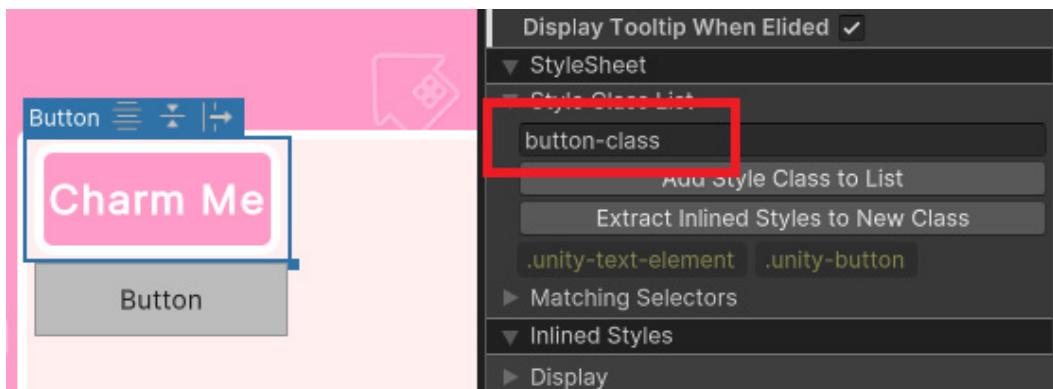


Figure 18.62: Creating a style sheet with Extract Inlined Styles to New Class

35. Select **Add to New USS** from the popup that appears.
36. Save the file in Asset/UI Toolkit as `ButtonStyle.uss`.
37. The `ButtonStyle.uss` style sheet should appear in the **StyleSheets** Panel now.

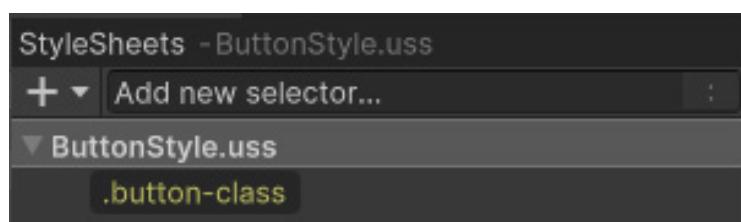


Figure 18.63: The new style sheet

You can now apply this style to the unstyled button by dragging `ButtonStyle.uss` from the **StyleSheet** window onto the Button. Now, the two Buttons will have the same properties.



Figure 18.64: The two buttons with the same style sheet

38. Change the text on the bottom Button to `Inspire Me`.
39. Now, let's align these buttons in their `VisualElement` parent. Select their `VisualElement` parent and set its **Align Items** and **Justify Content** properties both to center. You can do this either from the blue box on top of the `VisualElement` in the Viewport or from the **Align** property in the Inspector.

Your UI Builder should look like the following:

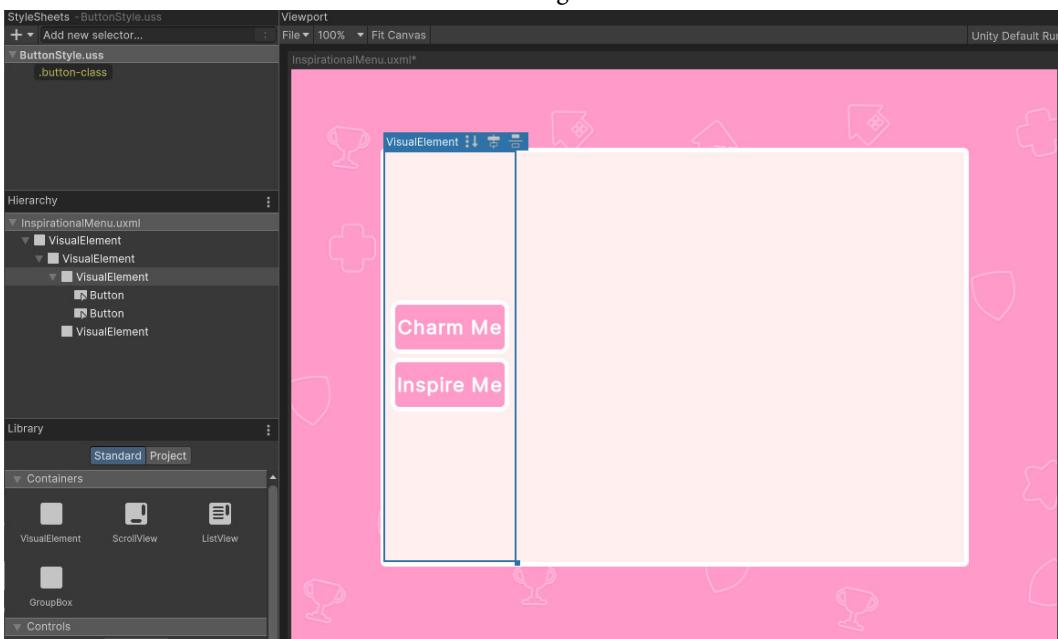


Figure 18.65: The Buttons centered in their `VisualElement` parent

40. Now, let's add in the holders for the cat pictures and the quotes that will be generated from the internet. Add a `VisualElement` so that it is a child of the bottom-most `VisualElement`.
41. Resize it so that it is smaller than its `VisualElement` parent.

42. Expand the **Position** property on its Inspector and change the position from **Relative** to **Absolute**. This will allow you to position it manually by either entering the numbers or dragging it into position in the Viewport. Position it like so:

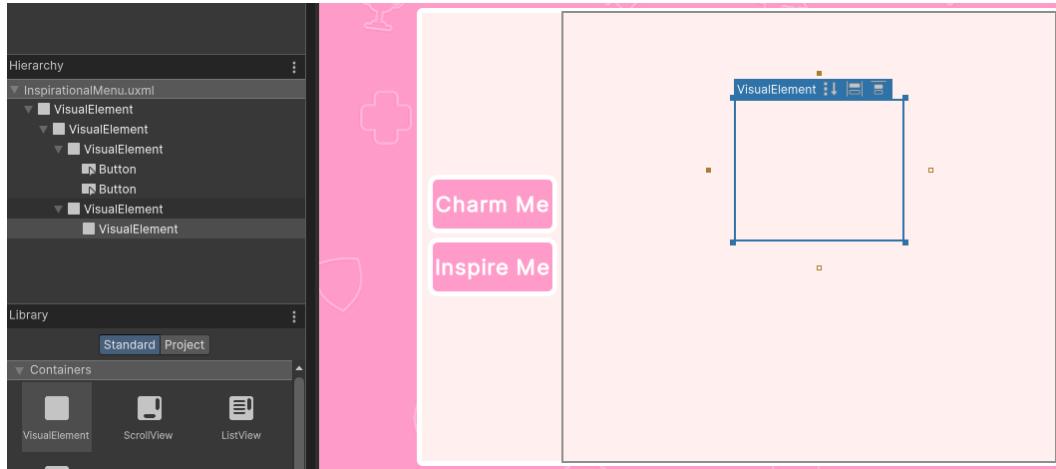


Figure 18.66: Manually positioning the VisualElement

43. Add a Label as a sibling of the VisualElement we just created.
44. Change its **Position** property to **Absolute**, then scale and position it as follows.

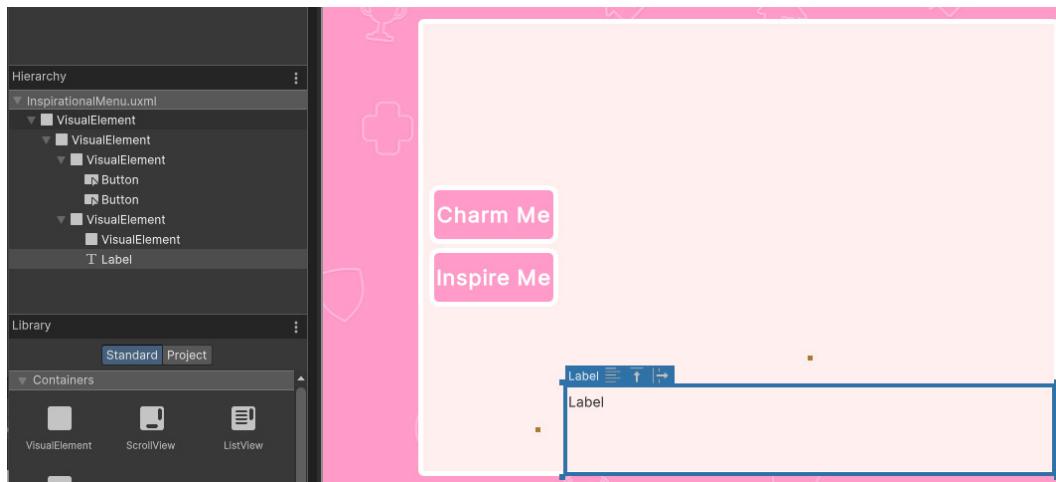


Figure 18.67: Adding a Label to the UI

-
45. Change the **Text** properties to what is shown in the following figure. The font color is FF9BC8.

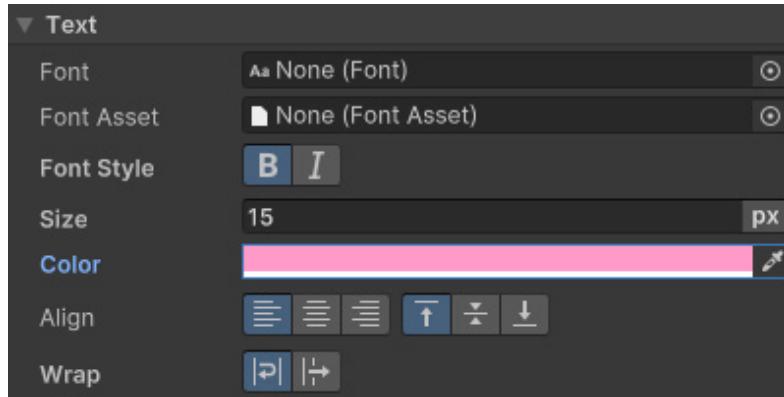


Figure 18.68: The Text property of the Label

46. Delete the text so that there is nothing displayed there.
47. Currently, when you click on the buttons, there is not much of a reaction. This makes it difficult for the user to tell if they are clicking the buttons. So, let's give the buttons hover and active states.
48. Select the `.button-class` in the **StyleSheets** Panel. Right-click on it and select **Duplicate**.
49. Right-click on the duplicate and rename it to `.button-class:hover`. This naming convention indicates that this style will be applied to the button's hover state.
50. Duplicate it again and rename the new duplicate to `.button-class:active`. This naming convention indicates that this style will be applied to the button's click state.
51. Select the `.button-class:hover` style. In the Inspector, change the **Background** color to E96FA6 and change its size to 200 x 90.
52. Select the `.button-class:active` style. In the Inspector, change the **Background** color to C35D8B and change its size to 200 x 90.
53. You can now preview the changes you made by either playing your game or by using the **Viewport Preview**. This will show you the hover and active state changes.

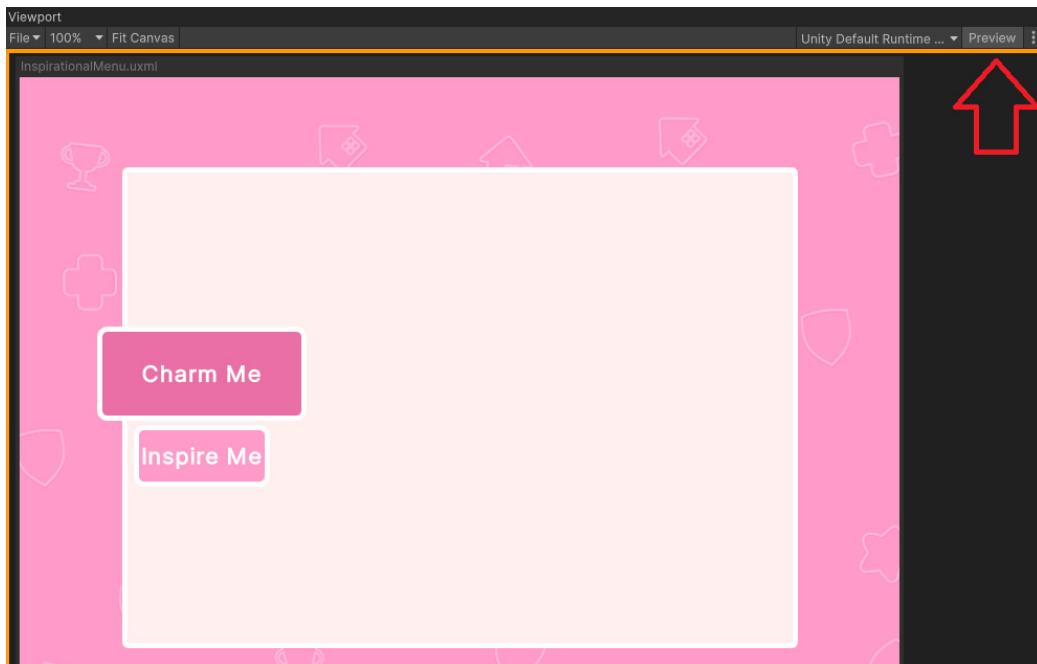


Figure 18.69: Using Preview in the Viewport to see the Buttons style changes

The buttons now get darker and larger when you hover and click on them.

54. These style changes on the buttons are fine, but the transition from its standard state to the others is a bit drastic. We can add transition animations to the `.button-class` style, so when it transitions to the other styles, it will animate more smoothly.

Select the `.button-class` style from the **StyleSheet** Panel. Within its Inspector, expand the **Transition Animation** property. We want it to animate when it scales. So, change the **Property** from **all** to **width** using the dropdown. Also, set the **Duration** to **0 . 5** and the **Easing** to **Ease In**. This indicates that when the button changes width, it will do so over 0.5 seconds and will ease into it.

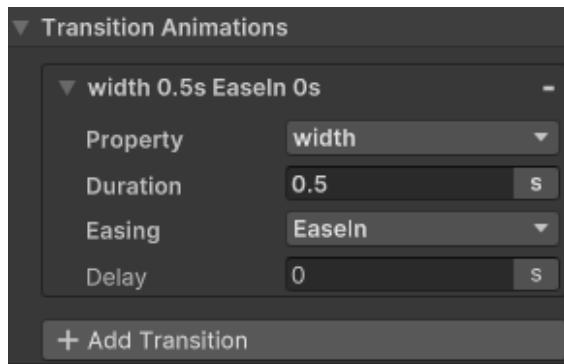


Figure 18.70: The width transition animation

55. Select the **+ Add Transition** button and change the properties on the new transition to **height**, **0 . 5**, and **EaseIn**. Your **Transition Animations** should now look like the following. This will ensure that when the width and height of the buttons change, it will do so smoothly over 0.5 seconds.

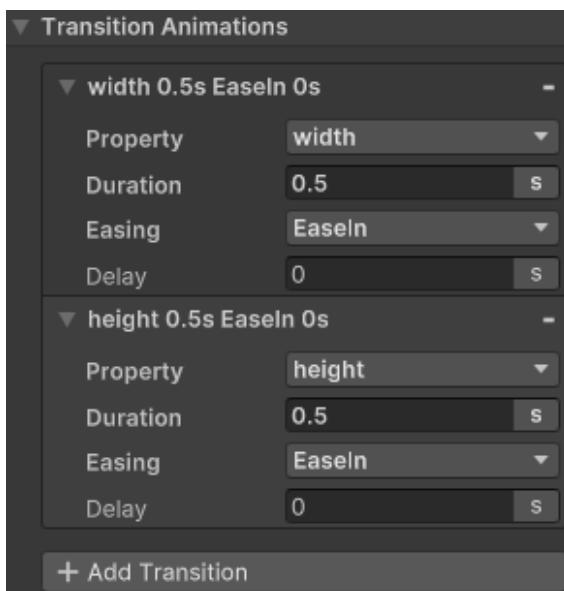


Figure 18.71: The Buttons' transition animations

Play the Preview to watch the buttons grow gradually as you hover over them.

56. The last thing we need to do in the UI Builder is give variable names for each of these Visual Elements so that we can find them via code. Change the names of the various Visual Elements to the following:

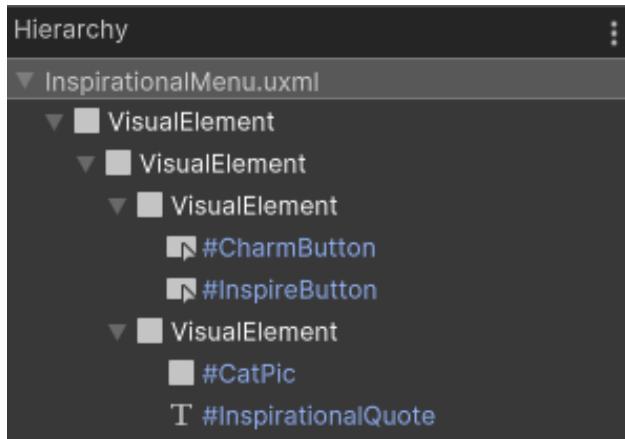


Figure 18.72: Renaming the Visual Elements we'll want to access via code

Whew. That was a lot of steps! But we are officially done with the UI Builder. We can now work on our C# scripts to get the functionality we want.

Using C# code to set VisualElement and Label properties with web data

Currently, our Buttons do not do anything but animate when we hover over them or click them. We now need to hook them up to some functionality. Our goals for the two Buttons when clicked are as follows:

- **CharmButton:** This should get cute cat pictures from the internet and replace the background of the CatPic Visual Element with the cute cat picture.
- **InspireButton:** This should get inspirational quotes from the web, format the text, and place it in the InspirationalQuote Label.

Honestly, what we've done up to this point for this example was all I wanted to cover when I initially started planning it. However, I got a bit excited about randomly generating the VisualElements properties and I possibly made the example a bit too complicated. We engineers love to over-engineer, after all. This example uses concepts of web requests and JSON manipulation. Since web development is not a focus of this book, I won't belabor the code that performs these functions. I want the focus of this example to be on the UI-specific code, not the web requests. I will explain what each section of code does, but I will not necessarily explain it line-by-line.

Before we get to the actual code, I do want to give a brief rundown of the two sources we will be getting our data from. We will use the following resources to get cat pictures and inspirational quotes, respectively:

- <https://placekitten.com/>
- <https://zenquotes.io/>

The Place Kitten website allows you to get a picture of a kitten with a specific dimension by simply adding the dimension to the end of the URL. For example, <https://placekitten.com/300/300> displays an image of a kitten that is 300 x 300. This website is great for adding placeholder images when you are developing websites and just need something to fill a specific place. Plus, it's cute!

The Zen Quotes website hosts an API that allows you to get inspirational quotes. An API is a collection of functions that let you interact with its data. What types of functions the API has will depend on the use of the API and the design paradigm the engineers chose to create it. But generally, an API will have the ability to GET data. Zen Quotes allows you to GET inspirational quotes. You can GET a single quote or a whole set of them. Visit their site to see what other options it has for retrieving data from its database. Zen Quotes returns the data you request from it in JSON format. JSON is a format standardization that provides information. It will look like the following when you get it:

```
{ "string": "HelloWorld", "boolean": false, "number": 123, "array": [1, 2, 3], "-object": { "prop1": "a", "prob2": "b" } }
```

But it can be formatted to look like the following:

```
{
  "string": "HelloWorld",
  "boolean": false,
  "number": 123,
  "array": [
    1,
    2,
    3
  ],
  "object": {
    "prop1": "a",
    "prob2": "b"
  }
}
```

I won't get into the specifics of what the preceding code means, but I will point out the important details of the example we cover. If you'd like to learn more about JSON format, see the following resource: https://www.w3schools.com/js/js_json_intro.asp

To finish out this example and have the buttons generate cat pictures and inspirational quotes, complete the following steps:

1. Create a new C# script in your `Scripts` folder and call it `InspriationalPanel.cs`.
2. Let's start with getting a reference to the `UIDocument` we created in the previous section. Add the following code to your class, making sure to import the `UnityEngine.UIElements` namespace:

```
private UIDocument uiDocument;

void Start()
{
    uiDocument = GetComponent<UIDocument>();
}
```

This code should look familiar after our Editor code example; the main difference is we got the reference to the `UIDocument` by using `GetComponent`. So, we'll need to put this script on the same `GameObject` that contains our `UI Document` component.

3. Go ahead and drag this script onto our `UIDocument` `GameObject`.
4. Now, create an instance variable for the root Visual Element on the UI Documents with the following code in the `Start()` method:

```
var root = uiDocument.rootVisualElement;
```

5. Now let's hook up our charm button. Add the following variable declaration:

```
private Button charmButton;
```

6. Add the following code to the `Start()` method to initialize it:

```
charmButton = root.Q<Button>("CharmButton");
```

7. We need a method that will run when the `charmButton` is clicked. Update your code to appear as follows. The new code is bold. This should all look familiar after our previous example:

```
public class InspriationalPanel : MonoBehaviour
{
    private UIDocument uiDocument;
    private Button charmButton;

    void Start()
    {
        uiDocument = GetComponent<UIDocument>();
        var root = uiDocument.rootVisualElement;

        charmButton = root.Q<Button>("CharmButton");
```

```

        charmButton.clicked += OnCharmClicked;
    }

    private void OnCharmClicked()
    {
        // Handle charm button click
    }

    private void OnDisable()
    {
        charmButton.clicked -= OnCharmClicked;
    }
}

```

I wanted to add the preceding large block of code because it can essentially act as a template for how you will code buttons with the UI Toolkit.

- Now, we need a reference to the Visual Element called `CatPic` so that we can change its background. Add the following code to the `Start()` method:

```
VisualElement catPic = root.Q<VisualElement>("CatPic");
```

- You may notice that I didn't make `catPic` a class variable but instead made it an instance variable within the `Start()` method. This is because the thing we will need to reference the most in this code is the `CatPic`'s style, not the `CatPic` itself. So, add the following variable to your class:

```
private IStyle catPicStyle;
```

- Now, add the following to the `Start()` method:

```
catPicStyle = catPic.style;
```

- To get the image for the `catPic` background, we'll use `https://placekitten.com/`. We'll randomly pick a width and height between 150 and 300, then get a picture from `placekitten.com` that fits those dimensions. To do so, we will need to use a coroutine since web requests require coroutines. Start by creating the following method:

```
IEnumerator GetCatPic()
{
    // Implement coroutine logic here
}
```

Note

This method is going to show an error in your IDE until you put in the web request.

Make sure to add the following namespace to your code so it recognizes what an `IEnumerator` is:

```
using System.Collections;
```

12. Now, let's randomly generate the width and height of the image we will request. Add the following to your `GetCatPic()` coroutine to get random numbers and then change the `catPic`'s dimensions to match it:

```
int randomWidth = Random.Range(150, 300);
int randomHeight = Random.Range(150, 300);

catPicStyle.width = randomWidth;
catPicStyle.height = randomHeight;
```

13. Remember, the format of the URL for a placekitten image is `https://placekitten.com/300/300`. So, we need to create a string that replaces the two `300` values with `randomWidth` and `randomHeight`. Add the following line of code to your `GetCatPic()` coroutine:

```
string uri = "https://placekitten.com/" + randomWidth + "/" +
randomHeight;
```

14. Now, let's send a web request to the `uri`. Add the following code to your `GetCatPic()` coroutine:

```
UnityWebRequest request = UnityWebRequestTexture.
GetTexture(uri);
yield return request.SendWebRequest();
if (request.result != UnityWebRequest.Result.Success) {
    Debug.Log(request.error);
} else {
    // Do stuff here with returned data
}
```

The preceding code tries to get a texture from a website. The `if` prints an error if the request fails. Make sure to import the following namespace so your script understands what a `UnityWebRequest` is:

```
using UnityEngine.Networking;
```

15. Now, let's replace the `// Do stuff here with returned data` part with the following code:

```
Texture2D myTexture = ((DownloadHandlerTexture)request.
downloadHandler).texture;
Debug.Log("Texture Acquired");
catPicStyle.backgroundImage = new StyleBackground(myTexture);
```

This code will take the picture returned by the web request and store it as a `Texture2D`. It then sets the background image of `catPicStyle` to that `Texture2D` image.

16. Now, we need to have our button to actually call the `GetCatPic()` coroutine. Update the `OnCharmClicked()` method to call the `GetCatPic()` coroutine:

```
private void OnCharmClicked()
{
    StartCoroutine(GetCatPic());
}
```

If you play the game, you can now click the **Charm Me** button to get random cat pictures.

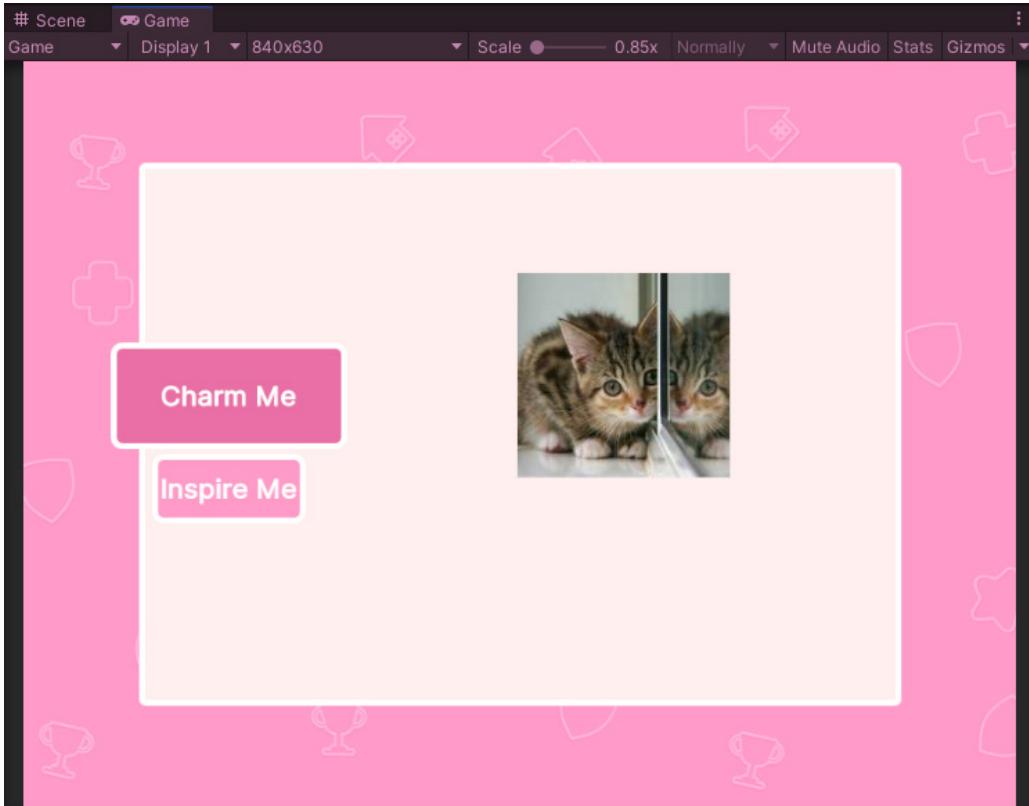


Figure 18.73: Random cat pictures appearing in our UI

17. You may notice a little funkiness with the image if you click the button a second time. Because the `GetCatPic()` method contains a web request, it takes a moment to fetch the image and then turn it into a texture, so whatever image is currently in its place will resize, becoming distorted, before the new image loads in.

Instead of letting this happen, let's remove the original image while the new image loads in. Add the following code as the first line of your `GetCatPic()` coroutine:

```
catPicStyle.backgroundImage = null;
```

18. Now, let's set up the functionality of the **Inspire Me** button. Add the following variable declarations to your class:

```
private Button inspireButton;  
private Label inspirationalQuote;
```

19. Add the following method that will be called when the `inspireButton` is clicked:

```
private void OnInspireClicked()  
{  
    // Implement functionality for when the inspireButton is  
    // clicked  
}
```

20. Add the following code to the `Start()` method. This should all look familiar to you by now:

```
inspireButton = root.Q<Button>("InspireButton");  
inspireButton.clicked += OnInspireClicked;  
inspirationalQuote = root.Q<Label>("InspirationalQuote");
```

21. Now, let's move on to filling in the data for the inspirational quote. We will use the `https://zenquotes.io/api/random` request to retrieve a random quote. If you navigate to that URL in your web browser, you can see how the data it returns is formatted. Add the following code to your `InspirationalPanel.cs` script:

```
IEnumerator GetInspiringQuote() {  
    UnityWebRequest request = UnityWebRequest.Get("https://zenquotes.io/api/random");  
    yield return request.SendWebRequest();  
  
    if (request.result != UnityWebRequest.Result.Success) {  
        Debug.Log(request.error);  
    } else {  
        Debug.Log("Quote Acquired");  
        string response = request.downloadHandler.text;  
        Debug.Log(response.ToString());  
  
        // more code will go here  
    }  
}
```

You'll notice this is structured similarly to the web request we used to get our cat image. Before we proceed, let's explore how the data is returned.

22. Update your `OnInspireClicked()` method with the following code:

```
private void OnInspireClicked()
{
    StartCoroutine(GetInspirationalQuote());
}
```

23. Play the game and click the **Inspire Me** button. You will see something like the following in the console:

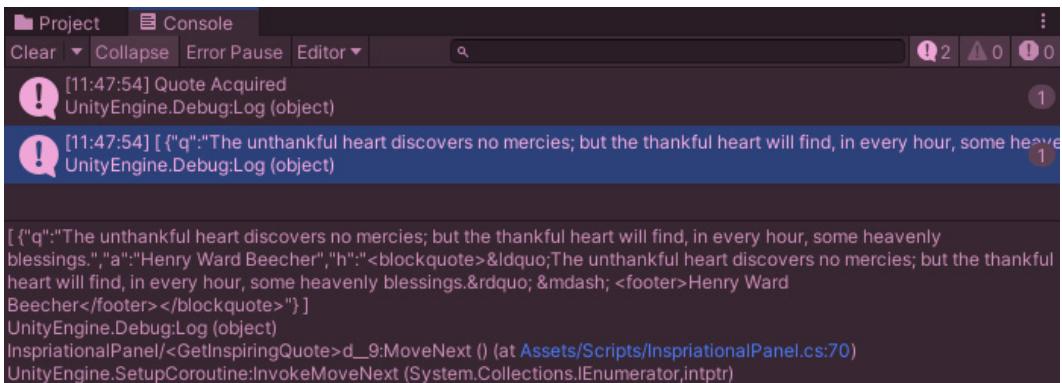


Figure 18.74: The API request response

As you can see, we can't exactly put the full result into our Label's text field. We need to format this into something usable and get only the information we want. To do this most simply, we'll need another package.

Open the Package Manager with **Window | Package Manager**.

24. Select the plus sign and select **Add package from git URL....**
25. Type `com.unity.nuget.newtonsoft-json` into the textbox and select **Add**. After some loading, you should see **Newtonsoft Json** in your packages list. This will allow you to easily manipulate JSON data.
26. Return to your script and add the following namespace:

```
using Newtonsoft.Json.Linq;
```

If **Newtonsoft** is not recognized by your IDE, close the IDE and Unity and reopen it.

27. Now, we can parse the data we receive from our API call and get just the information we want. The data that is returned by the API call starts with `[`. This means that the data we are receiving is coming to us in an array format. So, we need to convert the data to an array. Replace `// more code will go here` with the following:

```
JArray jArray = JArray.Parse(response);
```

28. We only want the strings assigned to the "q" property and the "a" property, as these represent the quote and the author respectively. To get this data, we need our data to be in an object format. The array that is returned to us only has a single array item, so use the following code to convert the array item to an object:

```
JObject jObject = JObject.Parse(jArray[0].ToString());
```

29. Now, we can get the strings we want. Add the following lines to get the quote and the author:

```
string quote = (string)jObject["q"];
string author = (string)jObject["a"];
```

30. The last thing to do is change the label's text to the information we want and make sure it's formatted correctly. I want the quote to appear in quotes. Then, on the next line, I want to see something like ~ Author's Name. The following line of code will do that:

```
inspirationalQuote.text = "\"" + quote + "\" \n~" + author;
```

Play the game and you should now be able to get pictures of kittens and inspirational quotes!

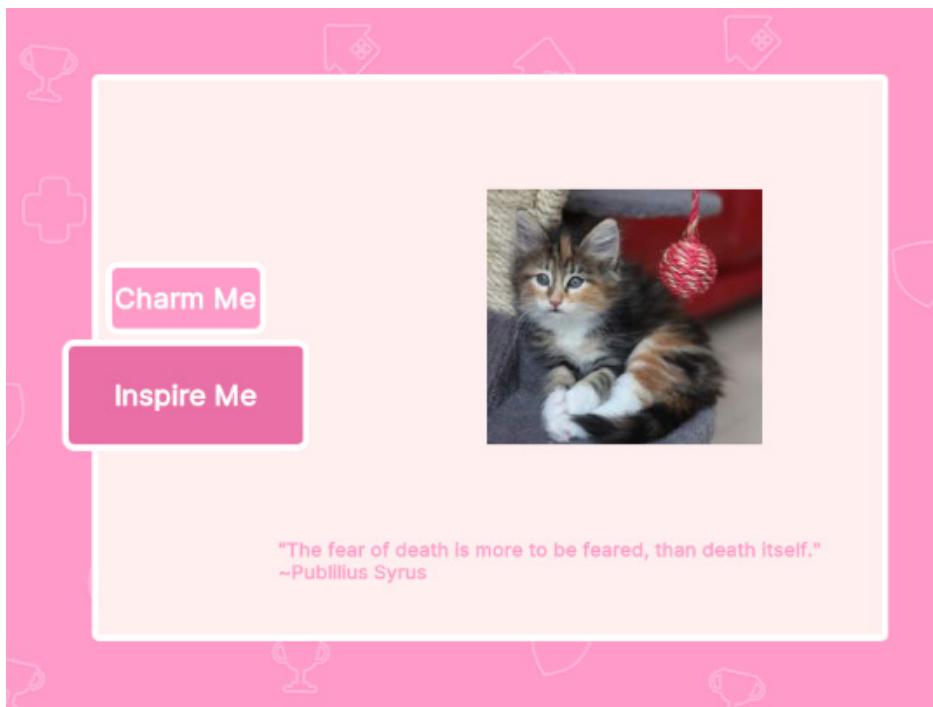


Figure 18.75: The final version of our example

Because zenquotes is an API, the developers have put a limit of 5 calls per 30 seconds. So, that means if you try clicking the button more than 5 times in 30 seconds, you will get an error message.

You could expand on this example by getting a set of the data at the start and storing it locally. This could reduce some of the load times of the images and reduce the number of API calls needed.

That's all for the examples that I will cover for the UI Toolkit. As I said previously, this is a big topic and a whole new way of thinking about developing UI. I wasn't able to cover even half of what I would have liked to. If you enjoy working with the UI Toolkit, I recommend viewing the resources section for suggested further reading and tutorials.

Resources

If you're looking for more documentation, I recommend the following resources provided by Unity:

- <https://unity.com/resources/user-interface-design-and-implementation-in-unity>
- <https://docs.unity3d.com/Manual/UIE-Transitioning-From-UGUI.html>

If you're interested in tutorials, here are some great resources. It's important to note that most of the tutorials at this point are for using the UI Toolkit for Editor UI since the runtime support is still new:

- <https://docs.unity3d.com/Manual/UIE-examples.html>
- <https://docs.unity3d.com/2021.2/Documentation/Manual/UIE-HowTo>CreateRuntimeUI.html>
- <https://docs.unity3d.com/2023.3/Documentation/Manual/UIE-simple-ui-toolkit-workflow.html>
- <https://learn.unity.com/tutorial/ui-toolkit-first-steps#61df0f23edbc2a2bf49579a2>
- <https://docs.unity3d.com/Manual/UIE-HowTo>CreateCustomInspector.html>
- <https://docs.unity3d.com/Manual/UIE-HowTo>CreateEditorWindow.html>
- <https://youtu.be/J2KNj3bw0Bw?feature=shared>

Lastly, Unity has provided some excellent pre-made projects that use the UI Toolkit. You can find these projects on the asset store here:

- <https://assetstore.unity.com/packages/essentials/tutorial-projects/quizu-a-ui-toolkit-sample-268492>
- <https://assetstore.unity.com/packages/essentials/tutorial-projects/ui-toolkit-sample-dragon-crashers-231178>

Summary

The UI Toolkit is still in development, but Unity is actively working on it as a system to replace the current uGUI system (that the rest of this book has focused on up to this point). It uses the concepts of web development to develop UI and can provide a cleaner, more performant UI than uGUI. However, since it is still in development, it doesn't do everything uGUI does ... yet. While you might not be able to fully transition over to the UI Toolkit system for your UI needs, it is helpful to have an idea of how it works, since one day, it will be your only option.

To help introduce the concepts of the UI Toolkit to you, we discussed the general concepts of the system as well as how to use it to make UI in the Editor and Runtime UI.

In the next chapter, we will discuss yet another UI system used within Unity: IMGUI.

19

Working with IMGUI

The last UI system I will cover is **IMGUI** or **Immediate Mode Graphical User Interface**. The primary usage for IMGUI is to create tools that assist developers during development and debugging. While IMGUI can technically make runtime UI, it is strongly discouraged by Unity. So, for example, you can use it to make Editor extensions or debug menus that will run in your game's view, and can be accessed when you play your game outside of the Editor. However, keep in mind these debug in-game menus are meant to be developer-facing, not player-facing. IMGUI is most commonly used for developing Editor UI extensions.

Since this book's primary focus is on runtime, player-facing UI, I won't delve too deep into this system; however, I'll cover the very basics of making developer-facing with IMGUI.

In this chapter, I will discuss the following:

- A general overview of how to use IMGUI
- The most commonly used IMGUI Controls
- How to use IMGUI for Inspector UI
- How to show debug frame rate text UI in your game
- How to put a button on an Inspector component and import data to a ScriptableObject

It is important to note that IMGUI is not a recommended system by Unity. When it comes to runtime UI, they recommend uGUI (which is what the majority of this book is about), and when it comes to Editor UI, they recommend UI Toolkit (which was covered in *Chapter 18*). So, learning IMGUI is not necessarily a required skill for anyone developing in Unity. It is especially not a required skill for non-programmers. However, it does give programmers a very quick way to build out UI to assist them in development, so the skills are not useless.

Let's review some basic information about IMGUI.

Technical requirements

You can find the relevant codes and asset files of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2019>

IMGUI overview

As I stated earlier, IMGUI gives programmers a quick way to build out UI that can assist them in their development. This is because IMGUI is built exclusively via code. It is not connected to GameObjects, and all objects are rendered via calls to an `OnGUI()` or `OnInspectorGUI()` method. The `OnGUI()` method is called every frame, similar to the `Update()` method.

If you want your IMGUI to appear within your scene, you write all your UI building code in an `OnGUI()` method within a `MonoBehaviour` inheriting script. Because IMGUI items are created via code, any UI you create with it on a `MonoBehaviour` script will not render until the game is run.

If you want your UI to appear in an Editor window, you will write all your UI building code in an `OnGUI()` method within an `EditorWindow` inheriting script. If you want your UI to appear in the Inspector, you write all your UI building code in an `OnInspectorGUI()` method within an `Editor` inheriting script.

All IMGUI items are created by calling their unique method within an `OnGUI()` or `OnInspectorGUI()` method. Each of their unique methods can be positioned and sized using rectangular position. When positioning an object with rectangular position, you first create a new Rectangle using the `Rect()` method. The `Rect()` method takes four parameters: `x` position, `y` position, `width`, and `height`. So, for example, you can create a new rectangle with the following line of code:

```
Rect rect = new Rect(10, 10, 50, 50);
```

This would create a rectangle at screen coordinate `(10, 10)` with a width and height of `50`.

Remember, screen coordinates put position `(0, 0)` in the top-left corner of the screen.

You can also use `Vector2s` to specify the parameters. For example, you could do something like this to achieve the same results:

```
Vector2 position = new Vector2(10, 10);
Vector2 dimensions = new Vector2(50, 50);
Rect rect = new Rect(position, dimensions);
```

Now that we know how to create IMGUI items, let's review the types of items that you can draw using `OnGUI()`.

IMGUI Controls

The items drawn using IMGUI are called **Controls**. While the term “Controls” may imply the item is interactable, not all Controls are interactable. There are multiple Controls available with the IMGUI system, but you can accomplish most of your IMGUI goals with the following:

- Label
- Button
- RepeatButton
- TextField
- TextArea
- Toggle

Note

You can find a comprehensive list of all IMGUI controls here: <https://docs.unity3d.com/2023.3/Documentation/Manual/gui-Controls.html>.

A **Label** is a non-interactable item. It can be either text or an image. To create a **Label**, you call the `GUI.Label()` method. The `GUI.Label()` method has multiple overloads, but the primary ones you will use are the following:

```
public static void Label(Rect position, string text);
public static void Label(Rect position, Texture image);
```

Where the first is used to define a text label, and the second is used to define an image label.

For example, the following code will display a text label and an image label in a scene, when it is attached to a GameObject and the `labelTexture` variable is assigned in the Inspector:

```
public class Chapter19Labels : MonoBehaviour {
    public Texture2D labelTexture;

    private void OnGUI() {
        GUI.Label(new Rect(10, 10, 100, 50), "Text Label");
        GUI.Label(new Rect(10, 80, 50, 50), labelTexture);
    }
}
```

The labels will look as follows:



Figure 19.1: Using IMGUI Labels

A **Button** Control performs a function with a single click. It only triggers once if the button is clicked and held. It can be a text button or an image button. To create a Button, you call the `GUI.Button()` method. The `GUI.Button()` method has multiple overloads, but the primary ones you will use are the following:

```
public static bool Button(Rect position, string text);
public static bool Button(Rect position, Texture image);
```

Where the first will be used to define a text button, and the second will be used to define an image button.

To execute code when a button is clicked, you create the button within an `if` statement within the `OnGUI()` method.

For example, the following code will display a text button and an image button in a scene:

```
public class Chapter19Buttons : MonoBehaviour {
    public Texture2D buttonTexture;

    private void OnGUI() {
        if (GUI.Button(new Rect(10, 10, 100, 50), "Text Button")) {
            Debug.Log("Text Button Clicked");
        }

        if (GUI.Button(new Rect(10, 80, 50, 50), buttonTexture)) {
            Debug.Log("Image Button Clicked");
        }
    }
}
```

Both buttons will write a message in the console each time they are clicked. The code must be attached to a GameObject and the `buttonTexture` variable must be assigned in the Inspector.

The buttons created by the preceding code will look as follows:

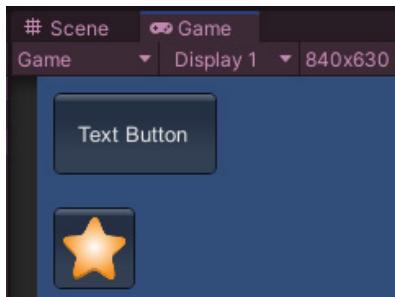


Figure 19.2: Using IMGUI Buttons

If you want to create a button that calls a method as it is clicked and held, you can use a **RepeatButton**. Its creation code is nearly identical to the creation of a **Button**. You can replace all instances of **GUI.Button** in the preceding code with **GUI.RepeatButton** and achieve a similar result, except the button will execute the method as long as the button is held.

The **TextField** and **TextArea** Controls allow you to create an interactive box with editable text. **TextField** is used when you want only a single line of editable text, whereas **TextArea** is used when you want multiple lines. To create a **TextField** or **TextArea**, you call the **GUI.TextField()** and **GUI.TextArea()** methods, respectively. The **GUI.TextField()** method has multiple overloads, but the primary ones you will use are the following:

```
public static string TextField(Rect position, string text);
public static string TextArea(Rect position, string text);
```

The first method creates a **TextField**, and the second creates a **TextArea**. In both cases, the **string** parameter is the text that will be displayed before the user begins editing the text. Note that the method returns a **string** type. You can get the value of what is entered by the user by assigning the constructed object to a **string** variable.

For example, the following code will display a **TextField** and **TextArea** you can interact with in your scene when it is attached to a **GameObject**:

```
public class Chapter10TextFieldAndArea : MonoBehaviour {
    private string textFieldText = "Enter text";
    private string textAreaText = "Enter text";
    private void OnGUI() {
        textFieldText = GUI.TextField(new Rect(10, 10, 100, 50),
textFieldText);
        textAreaText = GUI.TextArea(new Rect(10, 80, 100, 100),
textAreaText);
    }
}
```

The items rendered in the scene appear as follows before the user starts editing the text:

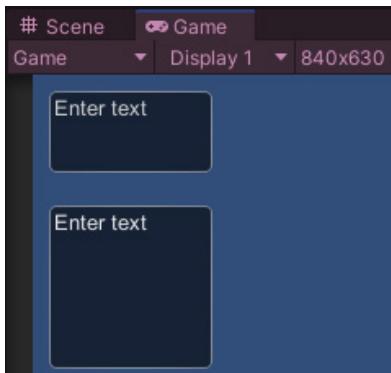


Figure 19.3: Using the IMGUI TextField and TextArea Controls

A **Toggle** Control is a checkbox that changes its state with a click. To create a Toggle, you call the `GUI.Toggle()` method. The `GUI.Toggle()` method has multiple overloads, but the primary ones you will use are the following:

```
public static bool Toggle(Rect position, bool value, string text);
```

Note that the method returns a `bool` type. You can get the value of the toggle by assigning the created object to a `bool` variable.

For example, the following code will create a toggle whose value is assigned to a Boolean variable whenever the toggle is clicked. This example, like the other examples, would need to be attached to a `GameObject` in your scene to render:

```
public class Chapter19Toggle : MonoBehaviour {
    private bool toggleBool = true;
    private void OnGUI() {
        toggleBool = GUI.Toggle(new Rect(10, 10, 100, 50),
        toggleBool, "Toggle Me");
    }
}
```

The toggle will initially render in the scene as follows until the user interacts with it, in which case the toggle will turn on and off with each click.



Figure 19.4: Using the IMGUI Toggle Control

All the examples I have shown were in the scene. However, if you want to display your UI in an Editor window, your code will work similarly. You simply put your code in an `EditorWindow` inheriting script. (See the example covered in *Chapter 18*.) However, it is slightly different if you want to create IMGUI in your Inspector. Let's look at that now.

IMGUI in the Inspector

Using IMGUI to enhance your components works very similarly to the way it does with in-game IMGUI and `EditorWindow` IMGUI. However, there are some small changes. First, you write scripts that inherit from `Editor`. Second, you use the `OnInspectorGUI()` method, not the `OnGUI()` method. Third, if you want the Inspector to also contain all its usual data, you need to call the `DrawDefaultInspector()` method within your `OnInspectorGUI()` method. Lastly, if you want the IMGUI button to appear in line with the various default Inspector elements, you use the `GUILayout` base class, rather than the `GUI` base class. So, for example, you wouldn't create a button with the following code:

```
GUI.Button(new Rect(10, 10, 100, 50), "Text Button");
```

Instead, you'd create it with this:

```
GUILayout.Button("Text Button");
```

Buttons created with `GUILayout` do not require a rectangular position and will automatically be positioned within the UI.

I will cover an example of this in the *Examples* section.

Now, you have enough basic background information to begin developing developer-facing UI with IMGUI. Let's look at some basic examples to get you started on using the system.

Examples

For the examples in this chapter, I will cover two types of IMGUI usages: one for an in-game debug menu and another for an Inspector UI. All the examples covered up to this point have all been in-game debug menu UI.

Using IMGUI to show framerate in-game

Let's create a very simple script that will show the frame rate of our game in the scene. It will change color if the framerate drops below a certain value.

To display the framerate in your game, complete the following steps:

1. Create a new scene called Chapter19-Examples.unity.
2. Within the new scene, create a new GameObject called DebugMenu.
3. Create a new script called DebugFrameRate.cs.
4. Attach the DebugFrameRate.cs script to your DebugMenu GameObject as a component.
5. Open the new script and add the following variable declarations:

```
int fps;
[SerializeField] int fpsThreshold;
```

The `fps` variable is used to get an estimate of our game's frame rate, while the `fpsThreshold` variable is assigned in the Inspector and used to determine the threshold for when our `fps` will display as red in the scene.

6. Create an IMGUI Label that will display the `fps` with the following code:

```
private void OnGUI() {
    GUI.Label(new Rect(10, 10, 50, 50), "fps = " + fps.
    ToString());
}
```

7. Now, let's calculate the `fps`. We can get an estimate of the `fps` with the following:

```
private void Update() {
    fps = (int)(1f / Time.unscaledDeltaTime);
}
```

8. Play your game, and you should see the `fps` displayed in the scene. If you open the **Stats** window, you should see that the values are relatively close.

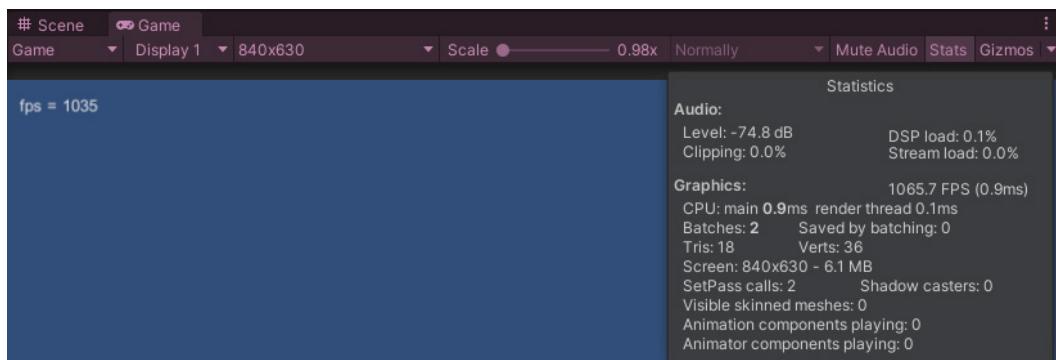


Figure 19.5: The frame rate displaying in-game

It's slightly difficult to see what the value is without pausing the game. So, let's make it change color when it drops below a certain value. This will make it easier to see when there is a framerate we find worrisome. We'll use the `fpsThreshold` value for this. Your game running in your Editor may run at a different framerate than mine, so please use a value that makes sense for your system to be able to see the code execute. I'm going to use 1000. Enter the value for your `fpsThreshold` in the Inspector.

- Now, let's make it change color when it goes below that threshold. Add the following `if/else` statement to the top of your `OnGUI()` method, before the label is created:

```
if (fps < fpsThreshold) {  
    GUI.contentColor = Color.red;  
}  
else {  
    GUI.contentColor = Color.white;  
}
```

And that's all you need to have a framerate estimate display in your game.

Displaying a framerate for your game is a great example of something you might want to make with an in-game debug UI. This will let you easily see the general performance of your game, even when you run it outside of the Editor. You could extend this to only appear when you perform specific tasks or only update every second. The possibilities are endless.

There are lots of other reasons you may want to use an in-game debug menu. For example, you might want buttons that call functions that help you skip to sections of your game. Or maybe you want a button that clears your saved data. The important thing to remember about making in-game UI with IMGUI is this should all be to help you, the developer, but should not be used to display information to your player.

Now, let's look at an example of something you can do in your Editor to assist your development.

Using IMGUI to make an Inspector button that imports data

Often in game development, you will have data stored in some external source and need to import it into a usable format within your game's code. For example, you may have a writer on your team who creates all dialogue in an Excel sheet, which you then need to figure out how to import into your game. This example will show a basic example that uses an Inspector button to read text from a file and distribute it to the appropriate place within your game, specifically a `ScriptableObject`.

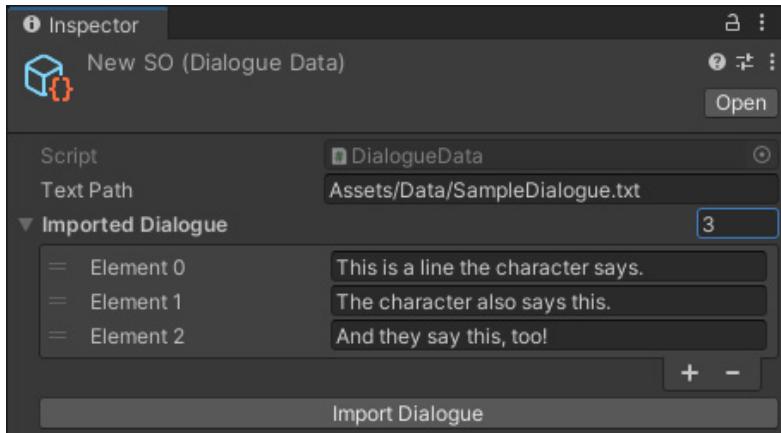


Figure 19.6: A custom Inspector with an import button

Note

We haven't really discussed ScriptableObjects in this text. A ScriptableObject is essentially a data container within Unity.

To create a button that imports data into your ScriptableObject, complete the following steps:

1. Create a new C# script called `DialogueData.cs`.
2. Create a new folder in your `Assets` folder called `Data`.
3. From the code files, find the text file called `SampleDialogue.txt`. Import it into your `Data` folder. Alternatively, you can create a text file with at least three lines of text and place it into this folder.
4. Open your `DialogueData.cs` script and change it so that it inherits from `Monobehavior` to `ScriptableObject`, like so:

```
public class DialogueData : ScriptableObject {
```

5. Add the following two lines of code to the class:

```
public string textPath;
public List<string> importedDialogue;
```

We'll use the `textPath` variable to define where the text file being imported is within our project and the `importedDialogue` variable to hold all the imported dialogue.

6. Add the following line above the class definition, to create a menu that makes `DialogueData` ScriptableObjects:

```
[CreateAssetMenu(fileName = "New SO", menuName = "DialogueData",
order = 1)]
```

7. Now, return to your Editor and create a new ScriptableObject by right-clicking within your Data folder and selecting **Create | DialogueData**.

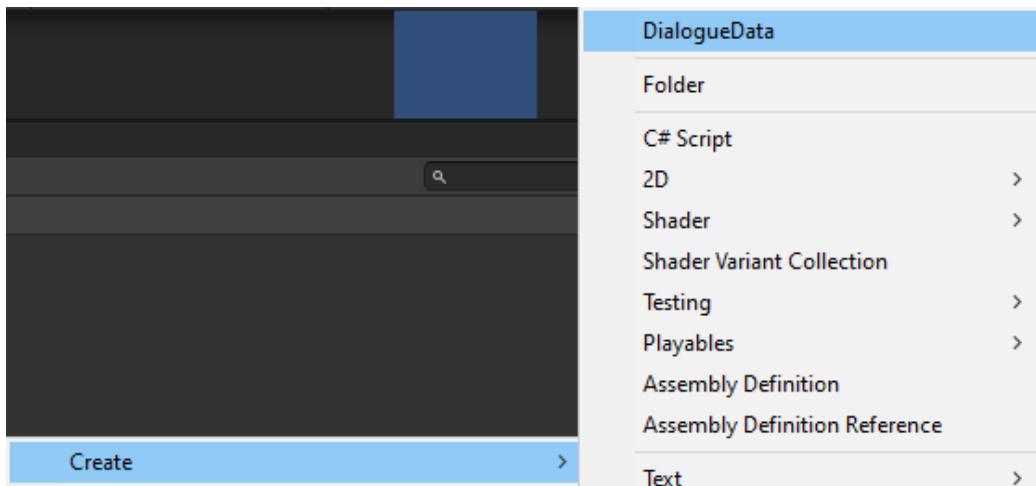


Figure 19.7: Creating the DialogueData ScriptableObject

This will create a new ScriptableObject called `New SO` with the following Inspector:

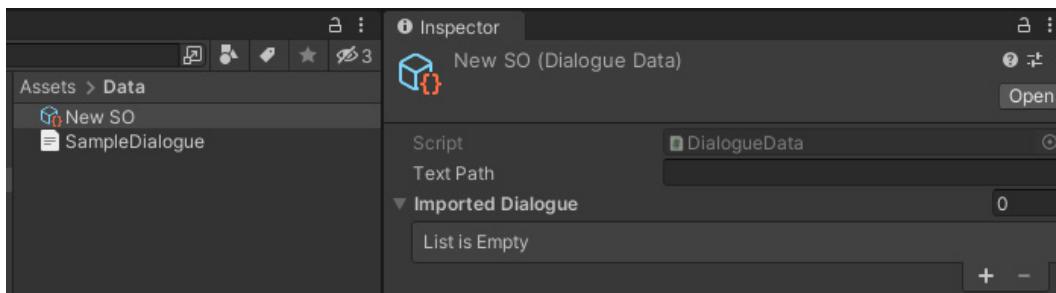


Figure 19.8: The ScriptableObject's Inspector

8. Now, let's customize the Inspector with an import button. Create a new script called `DialogueDataCustomEditor.cs` and save it in your `Editor` folder.
9. Open the script and make it inherit from `Editor` instead of `Monobehavior`, like so:

```
public class DialogueDataCustomEditor : Editor {
```

10. Make sure to add the following:

```
using UnityEditor;
```

11. Now, to let the script know that it is a custom Inspector for the DialogueData class, add the following above the class definition:

```
[CustomEditor(typeof(DialogueData))]
```

12. Add the following code to your script to display a button in the Inspector:

```
public override void OnInspectorGUI()
{
    if (GUI.Button(new Rect(10, 10, 100, 50), "Import Button"))
    {
        // Handle button click logic here
    }
}
```

If you return to your New SO, you'll see that the Inspector only contains a button now.

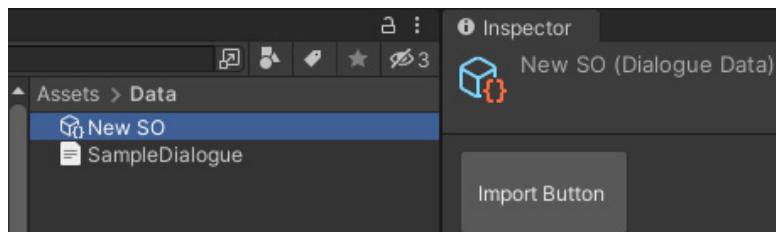


Figure 19.9: The button in the Inspector

13. It no longer displays all of the data that we want to show in the Inspector. So, add the following, above your button code:

```
DrawDefaultInspector();
```

14. It should now draw the button on top of the default Inspector information, which is not what we want.

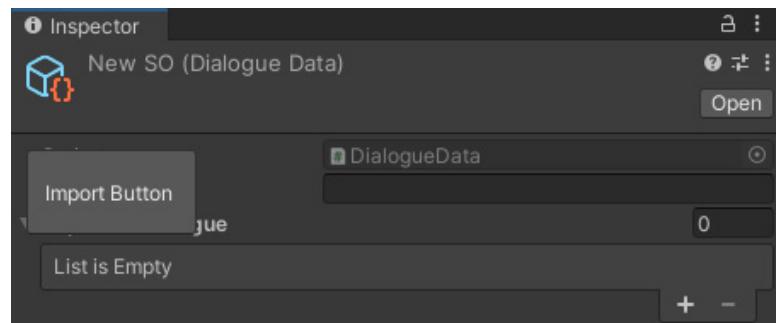


Figure 19.10: The button in the Inspector over the default information

Edit the button code to the following:

```
if (GUILayout.Button("Import Dialogue")) {  
}
```

This will cause the button to display at the bottom of the component, utilizing the layout of the Unity GUI instead of being explicitly positioned.

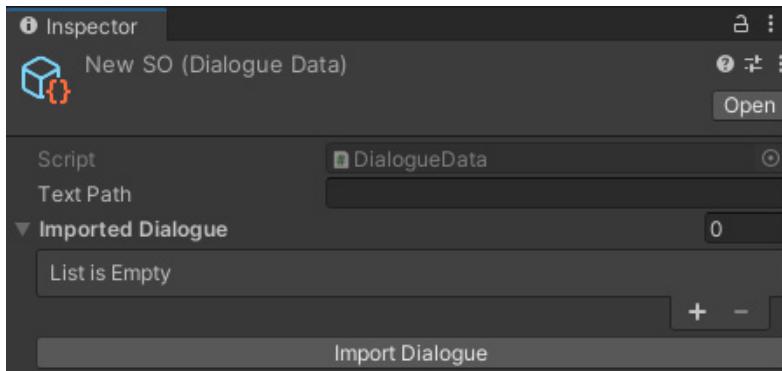


Figure 19.11: The button using the GUILayout base class

- Now, we just need to import the dialogue. Add the following variable declaration to your code:

```
private string[] splitTags = { "\r\n", "\r", "\n"};
```

We will use this variable to correctly parse the data in the text file, making sure each new line is a new line of dialogue.

- Now, we need to add code that will first get the data from the file and then send it to the ScriptableObject:

```
private void ReadString() {  
    DialogueData dialogueDataScript = (DialogueData)target;  
    StreamReader reader = new StreamReader(dialogueDataScript.  
textPath);  
    ParseFile(reader.ReadToEnd());  
    reader.Close();  
}  
  
private void ParseFile(string theFileText) {  
    Debug.Log(theFileText);  
    DialogueData dialogueDataScript = (DialogueData)target;  
    dialogueDataScript.importedDialogue.Clear();  
  
    string[] lines = theFileText.Split(splitTags,  
StringSplitOptions.None);
```

```

        foreach (var line in lines) {
            dialogueDataScript.importedDialogue.Add(line);
            EditorUtility.SetDirty(target);
        }
    }
}

```

17. Now, update your `OnInspectorGUI()` method to call the `ReadString()` method:

```

public override void OnInspectorGUI() {
    DrawDefaultInspector();
    if (GUILayout.Button("Import Dialogue")) {
        ReadString();
    }
}

```

18. We need to add the location of the `SampleDialogue.txt` file to our `ScriptableObject`. Right-click on `SampleDialogue.txt` and select **Copy Path**. Paste it into the **Text Path** slot.

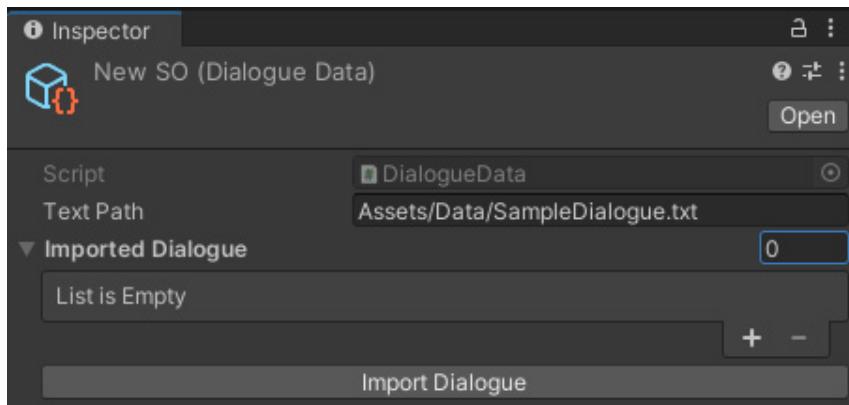


Figure 19.12: The `textPath` variable assigned in the Inspector

19. We should be good to go. Click on the **Import Dialogue** button and the dialogue will now import. Your Inspector should appear like *Figure 19.6*.

And that's it for using IMGUI to create a button within the Editor.

Summary

In this chapter, we discussed how to use the IMGUI system to build UI for both developers, in-game debug displays as well as Editor extensions. While IMGUI is not necessarily a recommended UI system, it is extremely helpful for creating super-quick tools to assist developers during the development process.

In the next chapter, we will look at the other input system provided by Unity: the New Input System.

20

The New Input System

Unity's **Input System** (colloquially referred to as *the new Input System*) allows you to make the code concerning how your game reacts to interactions from various input devices more modular. The old Input System used the Input Manager to allow you to define various Axes that could be referenced in code (see *Chapter 8*). You would create checks in your `Update()` method to determine if a device received an input. This usually results in multiple if-else branches within your `Update()` method that control what happens for each input received.

The Input System divorces the handling of individual input devices from code by using an event-based programming methodology. Instead of having to create reference to each of your input devices within your code, you create code that reacts to specific actions. The Input System then controls which Interaction from which Input devices trigger these actions.

This separation of input handling and code allows you to make more customizable controls, more easily handle different Input Devices, and more easily handle controls from different consoles that have extremely different control schemes. Additionally, the modularity of the Input System allows you to easily copy your control schemes to other projects, since the information is all stored on an asset.

This chapter is meant to be an introduction to the Input System and will give you an overview of the key concepts so that you can start using the Input System within your projects.

In this chapter, I will discuss the following:

- How to install the Input System
- The differences in polling vs subscribing
- The basic elements involved with the Input System
- How to write code that uses the Input System's to control your game
- How to convert a project that uses the old Input System (the Input Manager) to one that uses the new Input System
- Two different ways to connect your Input System to your code

Before we begin looking at how to work with the Input System, it needs to be imported into your project. Let's look at how to do that.

Technical requirements

You can find the asset files and codes of this chapter here: <https://github.com/PacktPublishing/Mastering-UI-Development-with-Unity-2nd-Edition/tree/main/Chapter%2020>

Installing the Input System

The Input System is no longer in preview and is officially part of Unity. However, it does not come pre-packaged in unity and must be installed. You can install the Input System into your Project via the Package Manager, by completing the following steps.

1. Select **Unity Registry** from the dropdown menu.

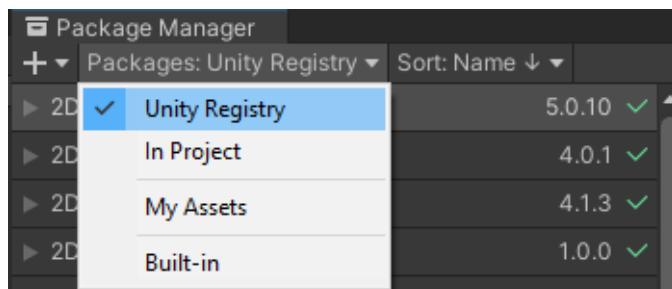


Figure 20.1: Changing the Package filter in the Package Manager

2. Search for the **Input System** in the list and select **Install**.

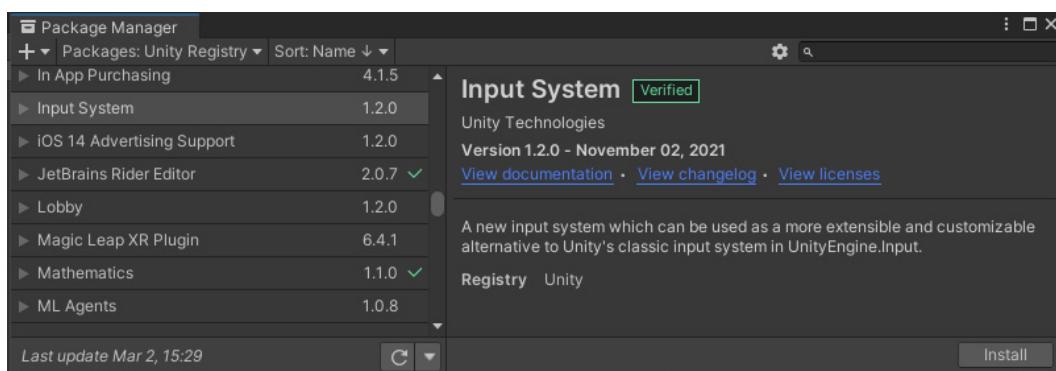


Figure 20.2: Finding the Input System in the Package Manager

You will see the following pop up which you must agree to in order to proceed.

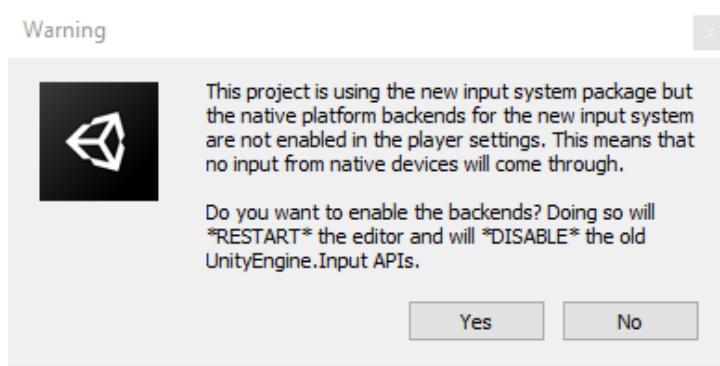


Figure 20.3: A Warning about changing input systems

The warning is indicating that any code you wrote using the old Input System will no longer function. This is a destructive action and may break your game.

Note

I recommend you create a new project to explore the Input System as installing it may be destructive to your project. It will cause all code written using the Input Manager to cease functioning. Don't install it in a pre-existing project until you are comfortable doing so.

Once you reopen your project, you may want to return to the Package Manager and install the various Samples provided by Unity with the Package. I recommend you install the Simple Demo and UI vs Game Input.

Since this is a new Input System, it does use a new methodology to access inputs than the old Input System. Let's first discuss the differences in these methodologies before we begin looking at the various elements of the Input System.

Polling vs subscribing

We discussed the Input Manager in *Chapter 8* and up until recently, it was the only way to track inputs from devices in a Unity-built game. When using the Input Manager, you assign specific Axes to different input actions. You then write a C# script that constantly checks to see if that action is performed.

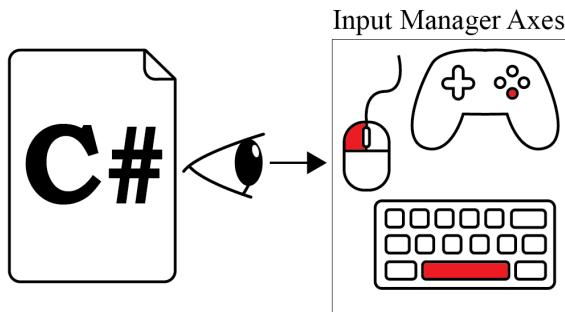


Figure 20.4: C# script watching for specific inputs defined by the Input Manager

To accomplish this, your code would look something like the following pseudocode:

```
void Update () {
    if (some input happened) {
        do something
    }
}
```

This technique of requesting information at a regular interval, is called **polling**. You can use the polling pattern for accessing information from the Input System in a similar way that you could with the Input Manager. However, to make your code more modular, most of your code when using the Input System will use a **publisher-subscriber (pub-sub)** pattern.

Consider the following analogy: you really enjoy content from a specific YouTube content creator. You check their channel every day to see if they have released new content. However, you find this cumbersome and instead decide to subscribe to their channel. Now the YouTube channel will alert you whenever new content is posted, saving you the work. Imagine how much work you save yourself from if you had hundreds of YouTube channels you wanted to be updated on by subscribing to their channels instead of constantly checking in on them.

Let's tie the analogy to coding. Instead of your C# script having to constantly check in on the various inputs in the `Update()` method, you can instead **subscribe** to specific events defined by the **Input System**. In more technical speak, you will write event subscriber methods that listen for specific events raised by the Input System.

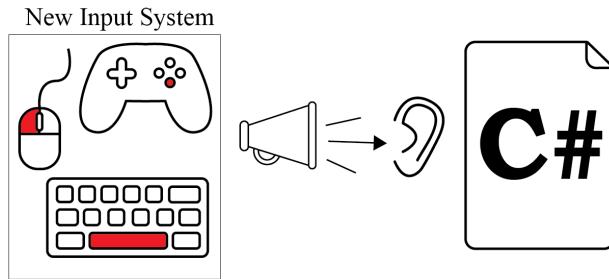


Figure 20.5: C# script listening for specific events defined by the New Input System

So, when using the New Input System, your code may look something like the following pseudocode:

```
void OnEnable() {
    subscribe DoSomething() to the event
}
private DoSomething() {
    do something
}
void OnDisable() {
    unsubscribe DoSomething() from the event
}
```

The Input System will determine which events to call based on inputs you specify and your code will subscribe to those events, performing the appropriate functions whenever those events happen.

So, how do you tell the Input System which events to call based on which input? And what types of events can it call from those inputs? You do this through the use of Actions, Interactions, and Input Binding. Let's explore these concepts now.

Input System elements

In the Input Manager, you list various Axes and define what buttons can trigger these axes. In the Input System, you define a set of **Actions** and describe what **Interactions** the various **Controls** of the **Input Device** can trigger those Actions through **Input Bindings**.

For example, you could create the Input Binding that binds the Space Bar on a keyboard to a jump Action through a pressed Interaction, where the Input device would be the keyboard and the Controls would be all the keys on the keyboard.

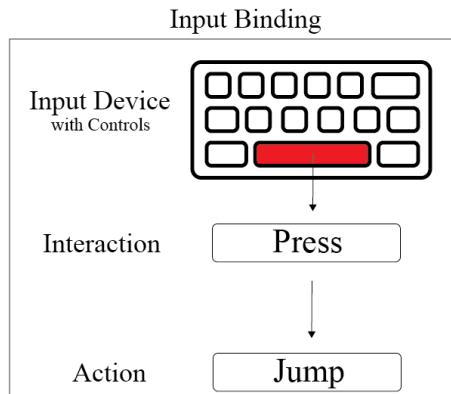


Figure 20.6: An example of an Input Binding

You can create these sets of **Actions** within what is known as an **Action Map**. Action Maps contain a list of Actions. The Actions contain the information about the Input Bindings. The general idea is to create Action Maps that contain groups of actions based on their purpose for example all character control Actions may go in one Action Map and all UI Actions may go in a separate one.

You can view your Action Maps, Actions, Bindings, and Interactions in the Action Editor. For example, here is the **Action Editor** with two Action Maps called Player and UI.

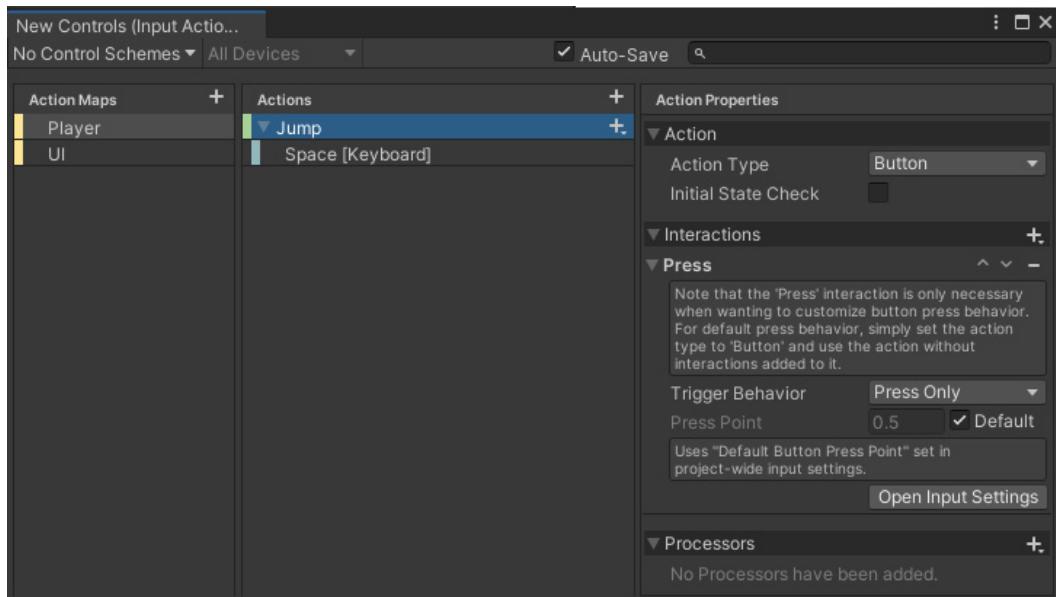


Figure 20.7: The Action Editor

Within the Player Action Map, you can see the Jump Action, with the Space [Keyboard] Binding, and the Press Interaction.

To start creating your Actions and Action Maps, right click within an Assets folder and select **Create | Input Actions** and name the new Input Action Asset whatever you wish. I recommend you save these assets in a folder called Inputs within your Assets folder. When you do so, you should get something that looks like the following:

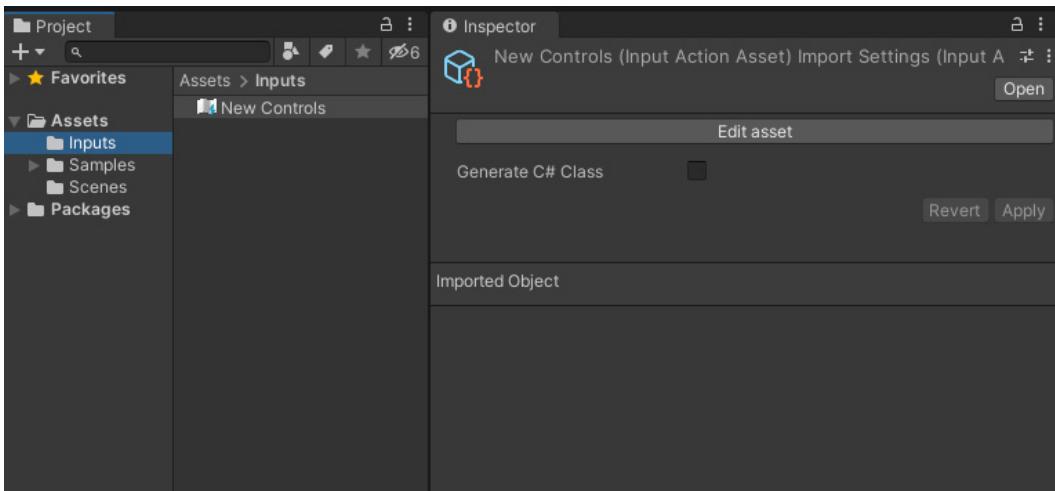


Figure 20.8: An Action Asset's Inspector and icon representation

You can either double click on the asset you just created or select **Edit asset** from its Inspector to view the Action Editor. The Action Editor will contain your list of Action Maps and Actions while displaying each of the Action's properties.

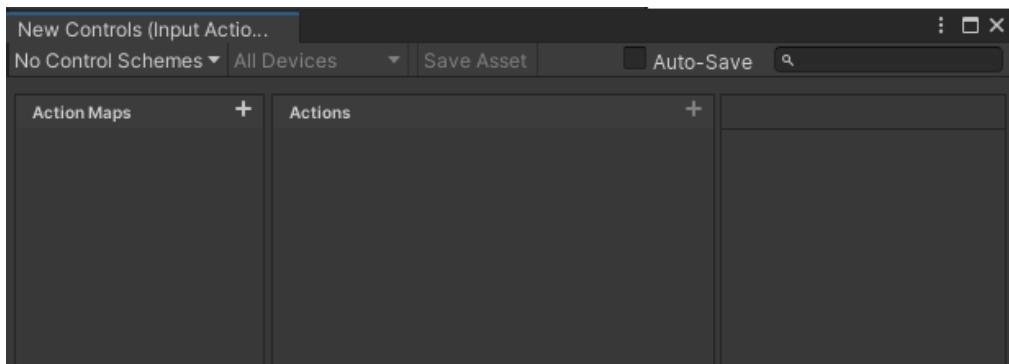


Figure 20.9: The Action Editor of a newly created Action Asset

Selecting the + sign in the **Action Maps** section will create a new Action Map.

A new Action Map will automatically come with a **New action**, which you can rename.

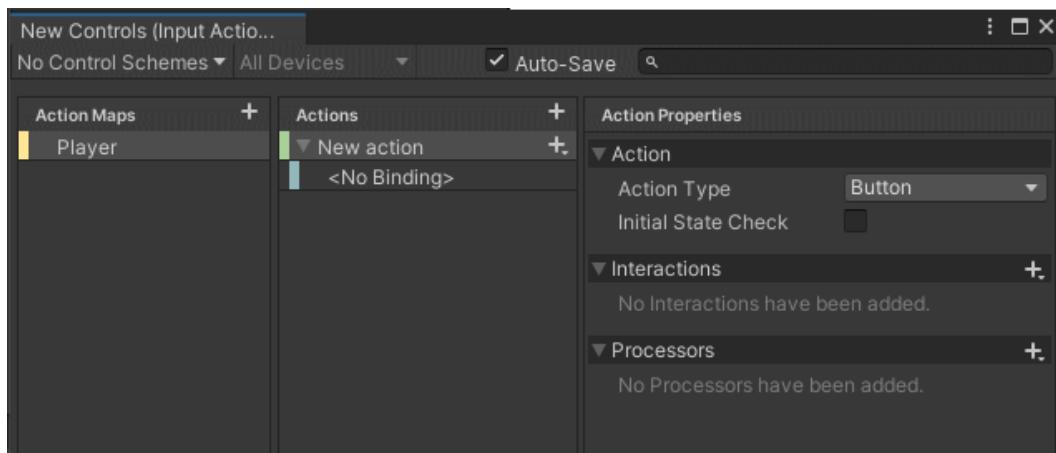


Figure 20.10: An Action Map with its New action

Selecting <No Binding> will allow you to create an Input Binding.

I will cover an example of how to select Binding and Interactions in the Examples section at the end of this chapter, but for now, let's look at how you can hook these Actions to your code.

Connecting Actions to code

There are multiple ways to work with the Input System's Actions in your code. In this section, I will give a general overview of the most important topics that should allow you to get started working with the Input System.

To connect your Actions to your code, you will need to import the `InputSystem` with the following statement:

```
using UnityEngine.InputSystem;
```

There are two ways in which I will discuss connecting Actions to your Code:

- Referencing the Action Asset
- Using the `PlayerInput` component

Note

For more information about alternate ways you can connect Actions to your code, see the following Unity documentation:

<https://docs.unity3d.com/Packages/com.unity.inputsystem@1.7/manual/Workflows.html>

You can reference the Action Asset creating a variable of type `InputActionsAsset` like so:

```
[SerializeField] private InputActionAsset actions;
```

You can then assign the value of `actions` in the Inspector.

And to reference the specific Action within a `InputActionAsset`, you could create a variable of type `InputAction`, like so:

```
private InputAction playerAction;
```

You could then assign it by finding the specific Action Map and Action within the Action Asset, like so:

```
playerMoveAction = actions.FindActionMap("Player").FindAction("Move");
```

Once you find the reference to the Action Asset, you will need to enable and disable the appropriate Action Maps. For Example, if you had an Action Map named `Player`, you could do the following:

```
private void OnEnable()
{
    actions.FindActionMap("Player").Enable();
}

private void OnDisable()
{
    actions.FindActionMap("Player").Disable();
}
```

Once you have a reference to the `InputActionAsset`, you can have your methods subscribe to the various callbacks of the Action. Each Action has the following callbacks that you can subscribe to:

- **Performed:** The Action's Interaction is complete
- **Started:** The Action's Interaction has started
- **Waiting:** The Action is enabled and waiting for an Input to cause trigger an Interaction
- **Canceled:** The Action's Interaction has been canceled
- **Disabled:** The Action cannot receive any input as it is disabled

So, for example, you could subscribe to an Action named `Jump` on an Action Map named `Player` that is triggered by a button press with something like the following:

```
actions.FindActionMap("Player").FindAction("Jump").performed +=  
OnJump;
```

If you wish to poll the Actions rather than subscribe to the callback events, you use the `ReadValue< TValue >()` method like so:

```
Vector2 moveVector = playerMoveAction.ReadValue<Vector2>();
```

If you're not a big fan of writing code, you can instead use the `PlayerInput` component. The `PlayerInput` component will allow you to specify which methods in your C# scripts are called by the various Actions.

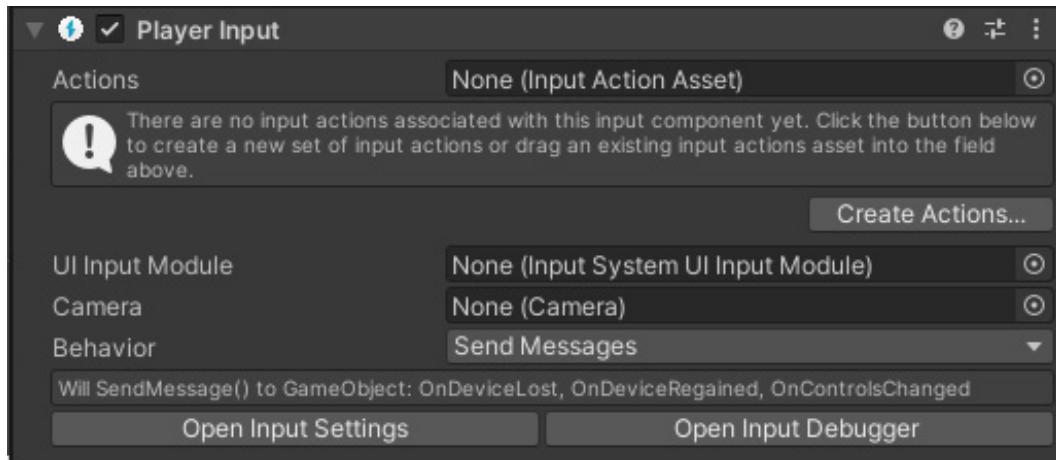


Figure 20.11: The default Player Input component

Using this method vs directly referencing the Actions in your code is mostly up to preference. It takes less code, but it takes more Inspector work. As a programmer by trade, I personally prefer the method that references the Actions in code. I find it easier to debug, more customizable, and quicker to edit when there are multiple objects using Actions. However, when I am working on a project that has designers making changes in the Inspector, who don't want to work in code, using the `PlayerInput` component is a good solution as it allows them to do things without me. You can also use a combination and it's all based on your needs and preferences.

Note

To learn more about the various ways you can access Actions with your code, see the following documentation:

<https://docs.unity3d.com/Packages/com.unity.inputsystem@1.7/manual/Workflow-ActionsAsset.html>

Now that we have a general idea of how the Unity Input System works, let's look at some examples of how to work with it.

Examples

Now that we've reviewed the basics of getting started with the Input System, let's look at some examples of how to implement it. We'll look at one very basic example of a character controller. We'll start with an example that uses the old Input Manager and then adjust it to use the Input System.

Note

This is a super basic character controller. It is simplified to make the process of converting it to the Input System easier to understand.

The example we will use is just a cat that jumps and moves around.



Figure 20.12: The character controller example using the old Input System

Before you begin these examples, complete the following steps:

1. Create a new 2D Unity Project.
2. Import the Chapter 20 - Example 1 - Start Package provided by the book's source code using **Assets | Import Package | Custom Package**.

After you've imported the package, play the scene Chapter 20 - Example 1 to get a feel of how the cat moves around. It's nothing fancy or particularly impressive, but the cat will jump with the space bar and move back and forth with the arrow keys and *A-D* keys.

3. Open the `InputManagerBasicCharacterController.cs` class and review the code. The key section of code is the `Update()` method:

```
void Update()
{
    movement = Input.GetAxis("Horizontal");
    catRigidbody.velocity = new Vector2(speed * movement,
    catRigidbody.velocity.y);

    if (grounded && Input.GetButtonDown("Jump"))
    {
        catRigidbody.AddForce(new Vector2(catRigidbody.
    velocity.x, jumpHeight));
    }
}
```

Notice the use of `Input.GetAxis("Horizontal")` and `Input.GetButtonDown ("Jump")`. These are defined within the Input Manager which you can find in **Edit | Project Settings | Input Manager**.

4. Now that you've played the game. Install the Input System Package and restart when requested. See the *Installing the Input System* section for steps.

If you try to play the game now, you will see the following error in your console.

```
InvalidOperationException: You are trying to read Input using
the UnityEngine.Input class, but you have switched active Input
handling to Input System package in Player Settings.
InputManagerBasicCharacterController.Update () (at Assets/
Scripts/InputManagerBasicCharacterController.cs:20)
```

This is because when we installed the Input System, the project stopped accepting inputs that use `UnityEngine.Input`. The cat will no longer move in response to our key presses.

Ok, now that we have our project started and set up, we can convert that code to something that can be used by the Input System! Let's start by setting up our Actions.

Creating basic character controller Actions

Before we start adjusting our code, we first have to set up our Action Map and Actions.

To set up your basic character controller Actions, complete the following steps:

1. Create a new folder called `Inputs` within your `Assets` folder.
2. Right-click within the folder and select **Create | Input Actions**.
3. Rename the new Action Asset to `CatActions`.
4. Double-click `CatActions.inputActions` to open the Action Editor.
5. Check the **Auto-Save** box so that any changes you make will automatically save.

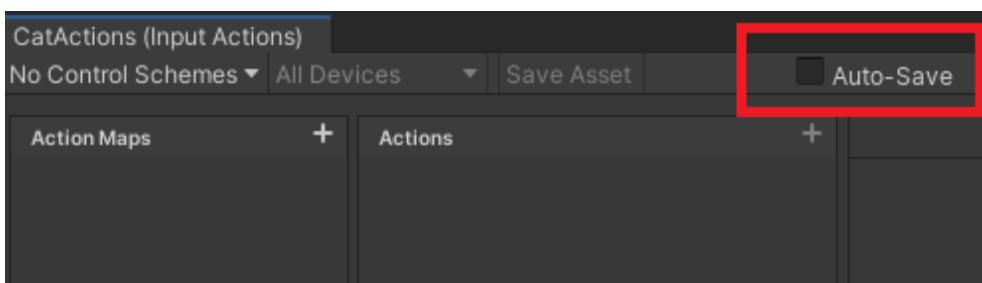


Figure 20.13: Selecting Auto-Save

6. Now we need to add an Action Map that will hold all the Actions for our character. Create a new Action Map by selecting the plus sign and then naming the Action Map `Player`. You should now see the following:

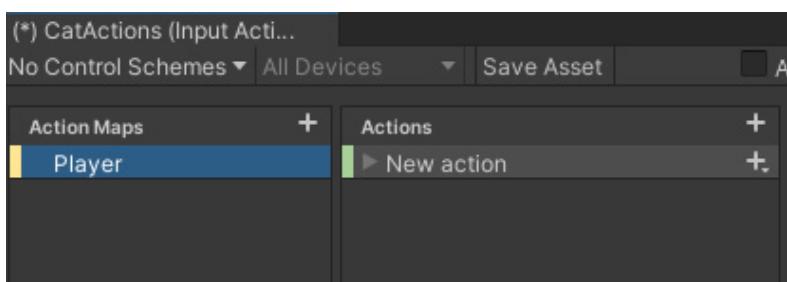


Figure 20.14: The Player Action Map

7. Let's create the Jump action by renaming New action to Jump. You can do so by double-clicking on the words **New action**. Notice that the **Action Type** is set to **Button**. This means the Jump Action will be triggered by button-like inputs.

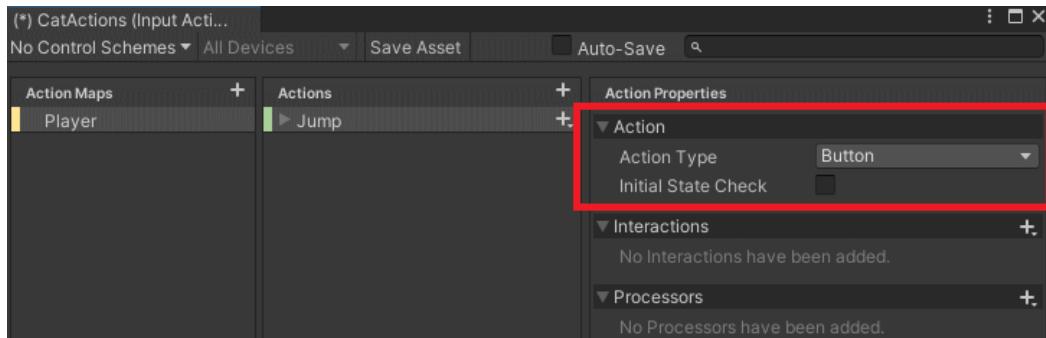


Figure 20.15: The Jump Action's properties

8. Click on the arrow next to the Jump Action to view all of its bindings. You should see **<No Binding>**.

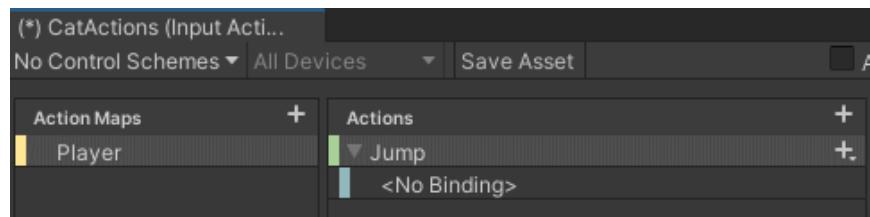


Figure 20.16: The Jump Action's bindings

9. Click on **<No Binding>** and select the dropdown next to the **Path** property. Within the text field that appears, type Space, then select **Space [Keyboard]**. This will bind the spacebar on the keyboard to this Action. Notice it automatically names the binding for you.
10. Now let's tell this binding what Interaction it can accept. Select the plus sign next to **Interactions** and select **Press** from the dropdown. Your Player Action Map should now look as follows:

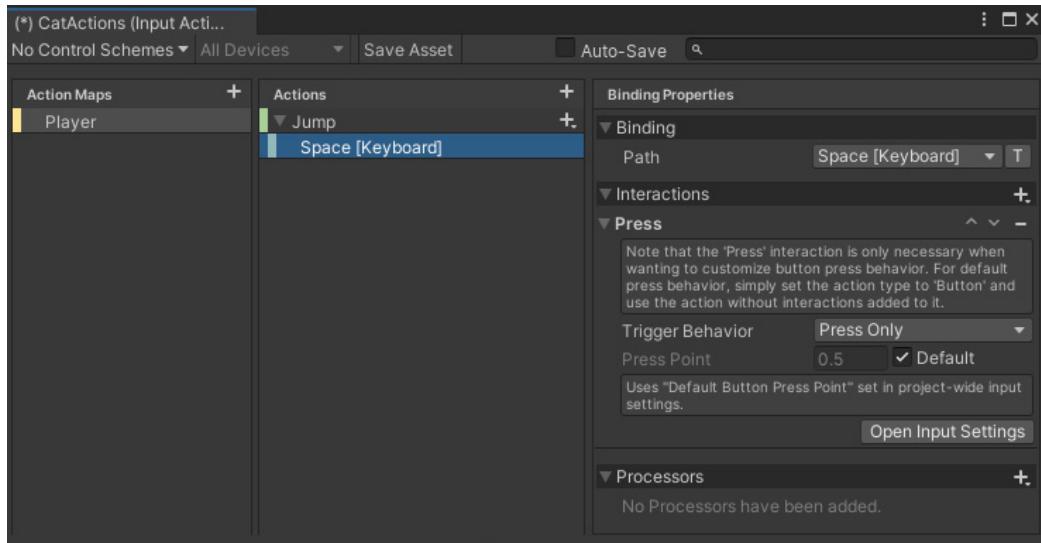


Figure 20.17: The Press Interaction on the Space [Keyboard] binding

11. Now we want to add the Action that will be bound to the keys that were previously referenced by "Horizontal" in the Input Manager. Select the plus sign next to **Actions** to create a new **Action**. Call it Move.
12. Set its **Action Type** to **Value** and its **Control Type** to **Vector2**.
13. Now we need to add the key bindings. Select the plus sign next to the Move Action and you'll notice, since we set Move as a different Action Type than Jump, it has different **Binding Options**. Select **Add Up\Down\Left\Right Composite**. You should now see the following:

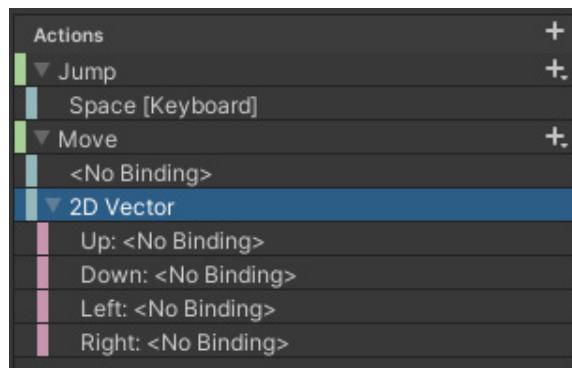


Figure 20.18: The 2D vector bindings

14. Delete the **Up** and **Down** Bindings as well as the one marked **<No Binding>**, because we do not need them. You can do this by right-clicking on them and selecting **Delete**.

15. Select **Left: <No Binding>**. Start typing **Left Arrow** in the **Path** to find **Left Arrow [Keyboard]**.
16. Select **Right: <No Binding>**. Start typing **Right Arrow** in the **Path** to find **Right Arrow [Keyboard]**. You should now have the following Actions and Bindings.



Figure 20.19: The Left and Right Bindings

We have now bound our arrow keys to the move action.

17. Now we need to bind the **A** and **D** keys. Duplicate the **Left: Left Arrow [Keyboard]** and **Right: Right Arrow [Keyboard]** bindings by right clicking them and selecting **Duplicate**.
18. Select the duplicate **Left: Left Arrow [Keyboard]** and start typing **a** keyboard into the **Path** to find **A [Keyboard]**.
19. Select the duplicate **Right: Right Arrow [Keyboard]** and start typing **d** keyboard into the **Path** to find **D [Keyboard]**. You should now see the following Actions and Bindings:

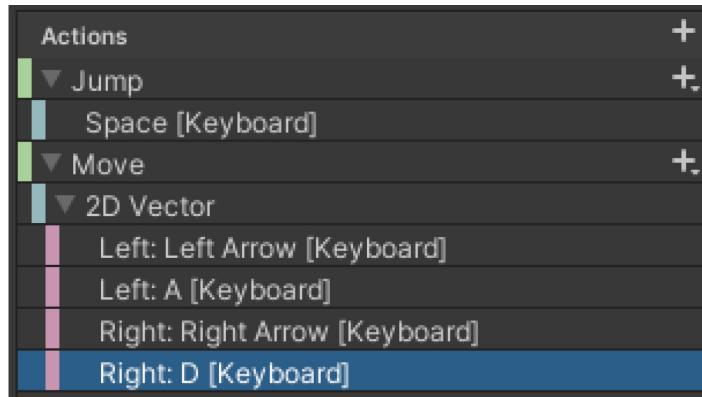


Figure 20.20: All the necessary Actions and Bindings

Now that we've done hooking up our Actions, we can start using them with our code! I'll show you two ways to do this: with the `PlayerInput` Component and by referencing the Action in our script.

Creating a basic character controller with the PlayerInput Component

Switching our code to use Actions and the `PlayerInput` component, requires a bit of code adjustment and some Inspector work.

To use Actions with the `PlayerInput` Component, complete the following steps:

1. To preserve the previous example, I am going to duplicate the scene and call it `Chapter 20 - Example 2`; duplicate the `InputManagerBasicCharacterController.cs` script; and rename the duplicate to `PlayerInputBasicCharacterController.cs`. If you'd rather just work in the same scene with the same script, you can skip this step. However, if you do want to do this, make sure to also change the name of the script in the class definition and add the script as a component to the Cat in your new scene.
2. Let's start by adjusting the script. We'll be removing the code within the `Update()` method that checks for the Input Axes and instead use public methods that can be hooked up in the Inspector. Comment out all the code within the `Update()` method. Don't delete it, b/c we'll cut and paste some of it to other methods later.
3. Add the following statement to the top of your script:

```
using UnityEngine.InputSystem;
```

4. Create a new method called `OnJump()`. This will be the method that is called when the `Jump` Action is triggered. It should look as follows:

```
public void OnJump(InputAction.CallbackContext context) {  
}
```

5. We want our `OnJump()` method to perform similarly to the following statement we have commented out in our `Update()` method.

```
if (grounded && Input.GetButtonDown("Jump"))  
{  
    catRigidbody.AddForce(new Vector2(catRigidbody.velocity.x,  
    jumpHeight));  
}
```

Cut and paste it into the `OnJump()` method.

6. Remove `&& Input.GetButtonDown("Jump")` from the `if` statement. Your `OnJump()` method should now appear as follows:

```
public void OnJump(InputAction.CallbackContext context) {  
    if (grounded) {  
        catRigidbody.AddForce(new Vector2(catRigidbody.  
velocity.x, jumpHeight));  
    }  
}
```

7. Now let's hook this method up in the Inspector. Select your Cat from the Hierarchy and add the PlayerInput Component. Your Cat's Inspector should have the following components:

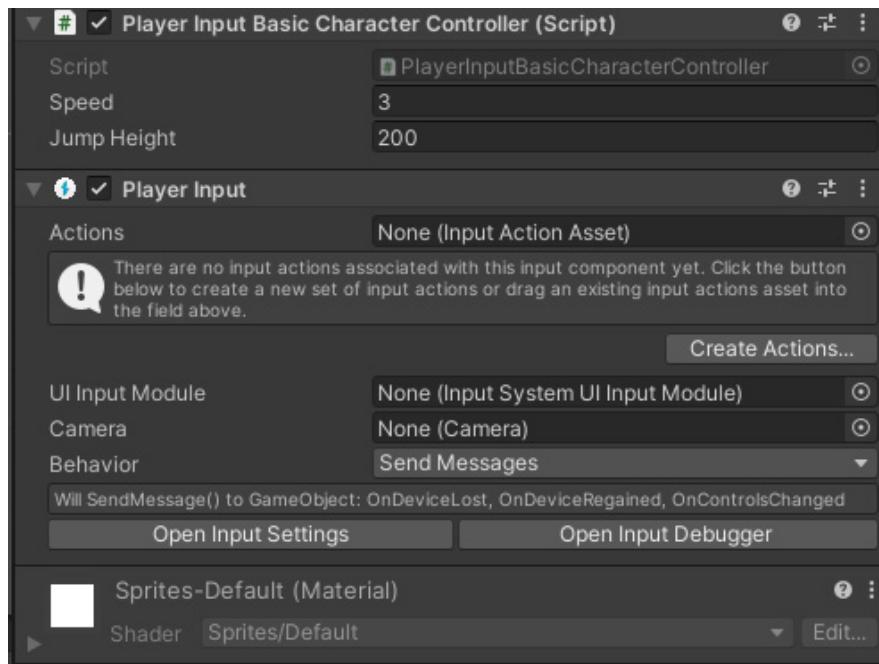


Figure 20.21: Some of the components on Cat

8. Now let's add our Actions to the **Player Input** component. Drag your CatActions from the Project folder into the **Actions** slot. Your component should now look as follows:

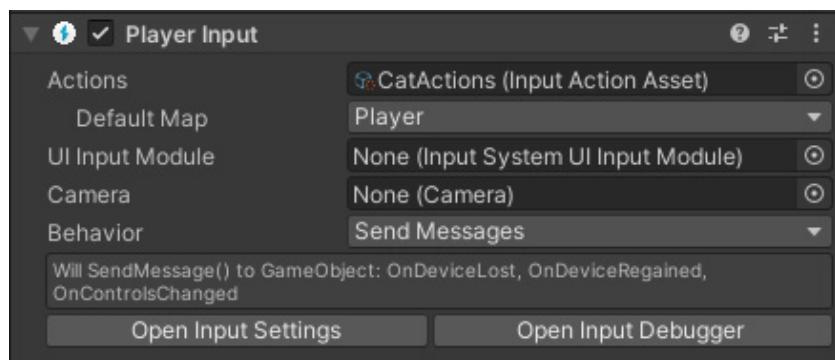


Figure 20.22: Assigning CatActions to the PlayerInput component

Notice it already found our **Player** Action Map and added it as the **Default Map**. If we had more Action Maps, there would be others in the dropdown we could choose from.

9. From the **Behavior** dropdown, select **Invoke Unity Events**. This will add an **Events** setting that can be expanded to reveal a list of **Player** Events. Expanding **Player** will show the two Actions we have defined within the Player Map along with some other useful Actions.

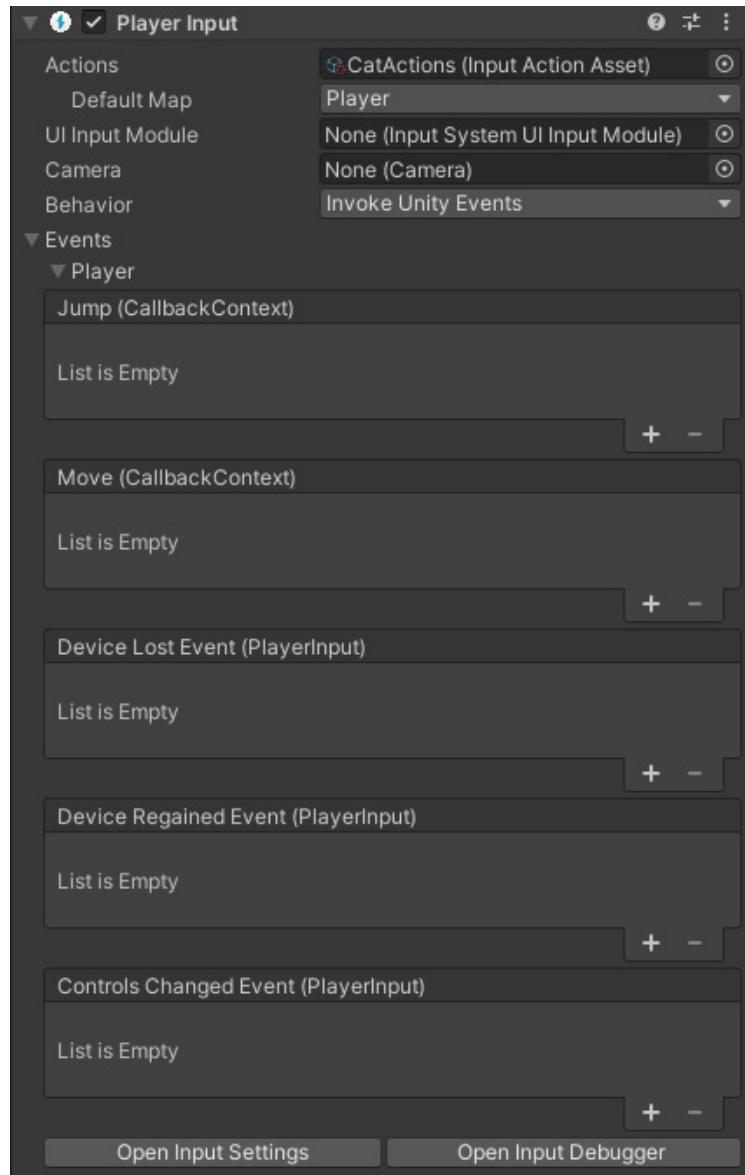


Figure 20.23: The Various Player Action Events

10. We can hook up our `OnJump()` method to the **Jump** Event by selecting the plus sign under **Jump(CallbackContext)**, dragging the **Cat** into the Object slot, and then selecting **PlayerInputBasicCharacterController | OnJump**. Your **Jump** Event should now appear as follows:

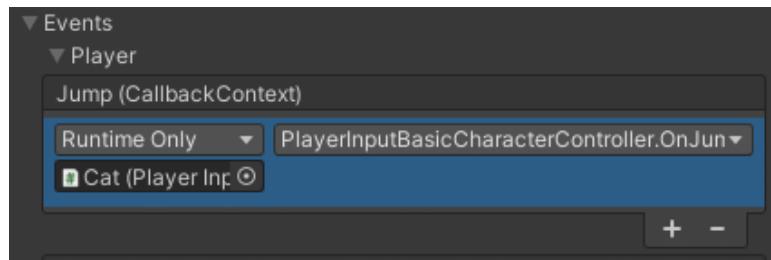


Figure 20.24: The Player Jump Event calling our `OnJump` method

Play the game and you should see the cat jump when you press the spacebar.

11. Now let's hook up the Move Action. Return to your code and create the following method:

```
public void OnMove(InputAction.CallbackContext context) {}
```

12. Add the following variable declaration to your code:

```
private Vector2 moveVector = new Vector2();
```

13. When we got the movement variable from the Input Axis, we used the following line:

```
movement = Input.GetAxis("Horizontal");
```

In the preceding code, `movement` was a `float` variable. While the `Horizontal` Axis returned a `float`, the Move Action sends `Vector2` value. Add the following line to your `OnMove()` method:

```
moveVector = context.ReadValue<Vector2>();
```

This will get the value of the Move Action whenever the `OnMove()` method is called.

14. Return to your `Update()` method and uncomment the remaining code. Your `Update()` method should appear as follows:

```
void Update()
{
    movement = Input.GetAxis("Horizontal");
    catRigidbody.velocity = new Vector2(speed * movement,
    catRigidbody.velocity.y);
}
```

15. Delete the following line that uses the old Input System.

```
Movement = Input.GetAxis("Horizontal");
```

16. Edit the remaining line to use the moveVector instead of the movement float.

```
catRigidbody.velocity = new Vector2(speed * moveVector.x,  
catRigidbody.velocity.y);
```

17. Remove the movement float variable declaration since we no longer need it.

18. Return to your Inspector and add the OnMove () method to the Move Event by selecting the plus sign under Move(CallbackContext), dragging the Cat into the Object slot, and then selecting PlayerInputBasicCharacterController | OnMove. Your Move Event should now appear as follows:

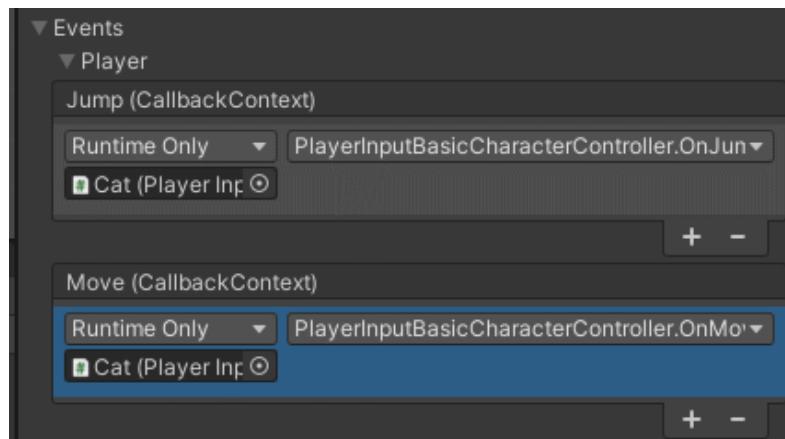


Figure 20.25: The Player Move Event calling our OnMove method

Play the game and your cat should now be able to move and jump!

Personally, I'm not a fan of using the PlayerInput component. Since I am a programmer by trade, I prefer to spend most of my time in my code editor rather than the Unity Editor. I find debugging code that is hooked up in the inspector difficult to debug. When I do use this functionality, I try to make life easier for my future self (or my other programming coworkers) by adding comments to the code that states what components call what methods. This can make debugging and understanding the way the code works a little easier. Using a code editor like Jetbrains Rider can relieve some of this stress, as it will indicate where code is hooked up in the Inspector, but it does not always show all necessary information and you can't count on your coworkers also having the Rider IDE. If you have a similar preference, I will show you in the next example how to do this in a code-centric way.

Creating a basic character controller by referencing Actions in your code

To redo our previous example so that it references the Actions in code, complete the following steps:

1. To preserve the previous example, I am going to duplicate the scene and call it Chapter 20 – Example 3; duplicate the `PlayerInputBasicCharacterController.cs` script; and rename the duplicate to `ActionReferenceBasicCharacterController.cs`. If you'd rather just work in the same scene with the same script, you can skip this step. However, if you do want to do this, make sure to also change the name of the script in the class definition and add the script as a component to the Cat in your new scene.
2. Delete the **PlayerInput** component from your Cat's Inspector. Notice if you try to play the game now, the cat will not move or jump, since we removed the component that tied the inputs to our code.
3. Open the `ActionReferenceBasicCharacterController.cs` script.
4. Comment out the `OnMove()` method. We will not be calling this via an Event in the Inspector, so we do not need the method. However, we will write code similar to it, so I'll comment it out for now so I can copy and paste it into the correct place later.
5. The first thing we need to do is get a reference to the Action Asset that holds all of our Actions. Create the following variable declaration:

```
[SerializeField] private InputActionAsset actions;
```

6. We can go ahead and hook this up in the Inspector by dragging the `CatActions` asset into the **Actions** slot.
7. Return to the script. Add an empty `OnEnable()` method and `OnDisable()` method to your script. We'll add some code to them momentarily.

I prefer to put the `OnEnable()` method right under the `Awake()` method since it executes after it and the `OnDisable()` method at the bottom of the script since it will execute after all of our other methods.

8. Since we are referencing our Actions via code, we will need to enable and disable our Action Map. Add the following boldened lines of code to your `OnEnable()` and `OnDisable()` methods.

```
private void OnEnable()
{
    actions.FindActionMap("Player").Enable();
}

private void OnDisable()
{
    actions.FindActionMap("Player").Disable();
}
```

9. Now let's hook up our Jump Action. We will accomplish this by subscribing to the Jump Action's performed event. Add the following code to your `OnEnable()` method.

```
actions.FindActionMap("Player").FindAction("Jump").performed +=  
    OnJump;
```

This should be enough to get the `OnJump()` method we already wrote working with your Action Map. Play the game and you can now see the cat jump when you press the space bar.

10. While not entirely necessary for this example, I prefer to make a habit of always unsubscribing from any events I subscribe to in `OnDisable()`. Making a habit of doing it will save me from problems in the future when it is necessary. So, add the following line of code to the top of your `OnDisable()` method:

```
actions.FindActionMap("Player").FindAction("Jump").performed -=  
    OnJump;
```

11. Now let's hook up our Move Action. To do this, we can create a reference to the `InputAction` that represents the Move Action to make it easily accessible via code. Add the following variable declaration to the top of your code:

```
private InputAction playerMoveAction;
```

12. Now let's initialize it. Add the following line of code to your `OnEnable()` method.

```
playerMoveAction = actions.FindActionMap("Player").  
    FindAction("Move");
```

13. Instead of subscribing to the Move Action, we'll poll it in our `Update()` method. Cut and paste the following code from within your commented out `OnMove()` method to the top of your `Update()` method. You can delete the `OnMove()` method now.

```
moveVector = context.ReadValue<Vector2>();
```

14. You will get an error on the `context` variable. It was a parameter for the `OnMove()` method and does not exist within the `Update()` method. Instead of reading the value of `context`, we will read the value of `playerMoveAction`. Replace `context` with `playerMoveAction` so that your `Update()` method appears as follows:

```
void Update()  
{  
    moveVector = playerMoveAction.ReadValue<Vector2>();  
    catRigidbody.velocity = new Vector2(speed * moveVector.x,  
    catRigidbody.velocity.y);  
}
```

And that should be sufficient to get our cat moving with Actions and without the `PlayerInput` component. Play the game and watch the cat move around and jump when you press the appropriate keys.

I know that setting our specific project up to use the Input System rather than using the Input Manager was a lot more work and may not seem worth it. And for this tiny example, that is probably true. However, if we wanted to create a cross platform version of this project that accepted inputs from multiple types of devices, adding new Bindings to an Action within an Action Map is significantly less work than adding in a new line of code for each possible input configuration we want to work with.

Summary

In this chapter, we covered how to use the New Input System to collect input for your game. This allows us to make an easily customizable input system that can make controlling our game via various input devices significantly easier in the long run. It may take a bit of work to set up, but it can save you a lot of effort in the long run.

And so, we come to an end of the book! I have no more User Interface knowledge to impart to you.

Index

Symbols

2D game background image placing 99-103
2D World Space status indicators 478-481
3D hovering health bars 481-487

A

accessibility design 40, 45 cognitive and emotional 48 hearing and speech 47 mobility 47 reference link 48 vision 46, 47
Action Editor 584
Action Map 584
Actions connecting, to code 586-589
American Standard Code for Information Interchange (ASCII) 247
Anchor Handles 80-83
Android screen resolutions reference link 19
animated text creating 259

Animation Clips 394-397 Animation Events 398, 399
Animation Events 398, 399
Animator Controller 399-405 Animation Parameters. setting in scripts 407, 408
Animator layers 406, 407 Animator of Transition Animations 405, 406 Behaviours 409, 411
Animator's Parameters setting, with code 424-427
ASCII Codes Table reference link 247
augmented reality (AR) 31-33 used, for designing user interface (UI) 37
automatic layout groups Grid Layout Group 116 Horizontal Layout Group 110 types 108-110 Vertical Layout Group 115

B

Background Canvas prefab creating 259, 260
BaseInputModule 162
benchmarking 491

billboard effect 477
button Animation Transition
 adding 224-229
Button Control 568
Button presses
 used, for loading scenes 222-224
buttons
 explicit navigation, selecting 217-221
 First Selected, using 210
 laying out 211-217
 navigating through 209, 210
 sizes 25-27

C

C#
 UI, making interactable with 513

Canvas component 64
 Screen Space-Camera 65, 66
 Screen Space-Overlay 64, 65
 text, scaling in 474-477
 World Space 66, 67

Canvas Group component 83, 84

Canvas Renderer component 74

Canvas Scalar component 68
 Constant Physical Size 71
 Constant Pixel Size 68, 69
 Scale With Screen Size 69, 70
 World 72

caret 375

C# code
 using, to make VisualElement and Label properties with web data 554-563

Central Processing Unit (CPU) 490

character spacing
 adjusting 279-281

circular progress bar
 creating 299-302

complex loot box
 animating 428, 429
 animations, setting up 429-436
 animations, timing 436-455
 State Machine, building 436, 437

Controls 567

coroutines 532

custom font 250-252, 273-279
 character spacing, adjusting 279-281
 font size, modifying 279-281

D

DaFont
 URL 245

designing for inconvenience concept 44

Device Simulator 19-21

device-specific resources 29

dialogue
 translating 267-273

diegetic interface 5

draw call 491

Dropdown component 362, 363
 caption properties 364
 On Value Changed (Int32) 365
 option properties 364
 template properties 364

dropdown menu
 creating, with images 380
 information, using from dropdown selection 388-390
 laying out, with caption and item images 380-383

Dropdown Template 360-362

E

eight-directional virtual analog stick
 creating 316-321
 making to float 321-325

event inputs 168-170

Event System 149-153

Event System Manager 153
 Drag Threshold property 154
 First Selected property 153
 Send Navigation Events property 154

Event Trigger component 163
 action, adding to event 166, 167
 event types 164

event types 164
 drag and drop events 165
 other events 165
 pointer events 164
 selection events 165

extended reality (XR) 31

F

field of view (FOV) 34

Fitters 126
 Aspect Ratio Fitter component 127, 128
 Content Size Fitter component 126, 127

Fitts' Law 7

Font Asset Creator
 reference link 253

font assets
 reference link 252

font color
 modifying, with markup 255

fonts
 Ascent Calculation Mode 248
 Character property 247
 custom fonts 250-252

dynamic font settings 248, 249
 font assets 253
 Font Size setting 246
 font styles, importing 250
 importing 245, 246
 Rendering Mode setting 247
 working with 245

font size
 modifying 279-281
 modifying, with markup 255

Font Squirrel
 URL 245

font style
 modifying, with markup 254

Frame Rate 490

frames per second (fps) 490

full screen taps 27, 28

G

game
 pausing 176, 177

game aspect ratio 7, 8
 changing 8-13

game resolution 7, 8
 changing 8-13
 single resolution, building 14, 15

Game view
 mobile resolution, setting 18, 19

GetAxis() function 159

GetButton() function 158

GetKey() function 159

GetMouseButton() function 160

Glyph metrics
 reference link 248

Google Fonts
 URL 245

graphical user interface (GUI) defining 3, 4
Graphic Raycaster component 72, 73, 170
Graphics Processing Unit (GPU) 490
grid inventory
 laying out 138-148
Grid Layout Group 116, 117
 Cell Size 117
 Constraint property 120, 121
 Start Axis property 119, 120
 Start Corner property 118, 120

H

heads-up-display (HUD) 4, 33, 64
 layout, creating 85-130
horizontal health bar
 creating 296-299
Horizontal Layout Group 110, 111
 Child Alignment property 112, 113
 Child Force Expand property 114, 115
 Control Child Size property 113
 Padding property 111, 112
 Reverse Arrangement property 113
 Spacing property 112
 Use Child Scale property 115
HUD selection menu
 laying out 130-138

I

Image Type property, UI Image
 Filled 291, 292
 Simple 289
 Sliced 289, 290
 Tiled 291

Immediate Mode Graphical User Interface (IMGUI) 502
 Controls 567-570
 examples 571
 in Inspector 571
 overview 566
 using, to create Inspector button 573-578
 using, to show framerate in-game 572, 573
input
 for accelerometer and gyroscope 162, 163
 for multi-touch 162
Input Field component 366
 Character Limit property 367
 Character Validation options 373-375
 Content Type property 367, 368
 Hide Mobile Input property 367
 Input Types 369
 Keyboard Types 369-372
 Line Type option 369
 On Value Changed (String) and On End Edit (String) 375, 376
 Placeholder property 367
 properties, of caret and selection 375
 properties, of entered text and onscreen keyboards 367
 Read Only property 367
 TextMeshPro 377, 378
 Text property 367
input functions
 for buttons and key presses 158
 GetAxis 159
 GetButton 158
 GetKey 159
 GetMouseButton 160
Input Manager 54, 154-157
 reference link 157
 using, with Pause Panel 174, 176

input modules 160

- BaseInputModule 162
- PointerInputModule 162
- Standalone Input Module 161

Input System 54, 579

- Actions to code, connecting 586-589
- elements 583-586
- examples 589, 590
- installing 580, 581
- polling, versus subscribing 581-583

Input System, examples

- basic character controller Actions, creating 591-594
- basic character controller, creating by referencing Actions in your code 600-602
- basic character controller, creating with PlayerInput Component 595-599

Input System 53

- and new Input System, selecting between 54, 55
- Input Manager 54
- new Input System 54

interactable UI

- consideration 35, 36
- placement 35, 36

interface 168

- types 4, 5

inventory items

- dragging and dropping 178-188

Inventory Panel

- KeyCode, using with 172-174

invisible button zones 207, 208**K****KeyCode**

- using, with Inventory Panel 172-174

keyframes 226**L****Layout Element 121, 122**

- Flexible Width and Flexible Height properties 125
- Ignore Layout property 122, 123
- Min Width and Min Height properties 123
- Preferred Width and Preferred Height properties 124
- Width and Height properties 123

M**MagicLeap**

- reference link 37

markup format

- exploring 254
- used, for modifying font color and size 255
- used, for modifying font style 254

masks

- component 328, 329
- using 328

meta interface 5**Milky Coffee font**

- reference link 261

mixed reality (MR) 31, 32

- reference link 37

- used, for designing user interface (UI) 37

mobile aspect ratio

- Device Simulator 19-21
- setting 18

mobile inputs 29

mobile orientation

- building 22-25
- setting 18

mobile resolution

- setting 18
- setting, in Game view 18, 19

multi-touch input 162**mute Buttons**

- with sprite swap 302-308

N**namespace 150****new Input System 54****nodes 400****non-diegetic category 5****Noto fonts**

- reference link 247

O**object pooling 496****optimization basics 490**

- CPU 491

- Frame Rate 490

- GPU 490

P**pan and zoom**

- implementing, with mouse and multi-touch input 188-194

Panel Settings 507**Particle System**

- timing, to play in loot box animation 467

Particle System, that displays in UI

- creating 459-466

Pause Panel

- Input Manager, using with 174-176

Physics 2D Raycaster 170**Pivot Points 80-83****PointerInputModule 162****polling 582**

- versus subscribing 581-583

pop in and out animation

- setting up 412- 424

pop-up menus

- hiding, with keypress 171

- setting up 103-106

- showing, with keypress 171

pop-up windows

- animating, to fade in and out 411

prefab 259**press-and-hold/long-press functionality**

- adding 308-312

publisher-subscriber (pub-sub) pattern 582**R****raycasters 170**

- Graphic Raycaster 170

- Physics 2D Raycaster 170

raycasting 170**Rect Mask 2D**

- component 329, 330

Rect Tool 76

- positioning modes 76, 77

Rect Transform component 63, 64, 76-78

- edit modes 78, 79

- Rect Tool 76

RepeatButton 569

S

scene

- creating 259, 260

screen portion taps 27, 28**Scroll Rect component** 337, 338

- movement properties 338, 339

- On Value Changed event 341, 342

- properties 339-341

Sequence Seekers 27**shooting gallery style game**

- reference link 273

Slider component 355-357

- On Value Changed (Single) 357, 358

spatial interface 5**Standalone Input Module** 161**state machine** 400**static four-directional virtual D-Pad**

- creating 312-316

statistics window 491, 492**Style Sheet** 507

- using 256, 257

subscribing

- versus polling 581-583

T

text

- translating 258, 259

TextArea Controls 569**text box text**

- animating 262-267

text box windows

- creating 260, 261, 262

TextField Control 569**TextMeshPro** 281, 358, 359, 377, 378

- Control settings 379

- Input Field settings 379

- On Select (String) and On
Deselect (String) 379

- reference link 237

- used, for creating wrapped text 282-286

Text-TextMeshPro 236, 238

- Extra Settings 242, 243

- Main Settings 239-242

- Text Input properties 238, 239

- TextMeshPro Project Settings 243, 245

TexturePacker

- reference link 273

Theme Style Sheet 508**thumb zone** 28, 29**Toggle component** 350, 351

- On Value Changed (Boolean)
events 352, 353

Toggle Control 570**Toggle Group component** 354**tools, for determining performance** 491

- statistics window 491, 492

- Unity Frame Debugger 494

- Unity Profiler 492- 494

Transition property, UI Button 200

- Animation transition 204

- Color Tint transition 200, 202

- None 200

- Sprite Swap transition type 202-204

U

UI Builder

- UI, creating with 511, 512

- using, to lay out menu 538-554

- using, to make animation

- transitions 538-554

- using, to make Editor Window's UI 518-527
- using, to make style sheets 538-554
- UI Button 198, 199**
 - Button component 200
 - Navigation property 205-207
 - Transition property 200
- UI Canvas 60-63**
 - Canvas component 64
 - Canvas Renderer component 74
 - Canvas Scalar component 68
 - Graphic Raycaster component 72, 73
 - Rect Transform component 63, 64
- UI Document 507**
 - using 513
- UI Dropdown 358**
 - creating 358, 359
 - Dropdown component 362, 363
 - Dropdown Template 359-362
- UI effect components 292**
 - Outline component 293, 294
 - Position As UV1 component 294
 - Shadow component 292, 293
- UI elements**
 - accessing, in code 150
 - laying out 6
 - UI variable types 150, 151
 - UnityEngine.UI namespace 150
- UIElements namespaces 514, 515**
- UI Hierarchy 509-511**
- UI Image component properties 288, 289**
 - Image Type property 289
- UI Images 84, 85**
- UI Input Field 365**
 - creating 365
 - Input Field component 366
- UI Panel 75, 76**
- UI Scrollbars**
 - component 331, 332
 - implementing 330
- UI Scrollbars, component**
 - On Value Changed event 332-334
- UI Scroll View**
 - examples 342
 - implementing 334-337
 - making, from pre-existing menu 342-348
 - Scroll Rect component 337, 338
- UI Slider 355**
 - creating 355
 - Slider component 355, 356
- UI systems 51, 52**
 - IMGUI 52
 - selecting between 53
 - UI Toolkit 53
 - Unity UI (uGUI) 52
- UI Text 61, 84, 85**
- UI Text GameObject 232, 233**
 - Character property 233
 - Color property 235
 - Material property 235
 - Paragraph properties 233-235
 - Raycast Padding properties 236
 - Raycast Target property 236
 - reference link 243
 - Text property 233
- UI Toggle 350**
 - Toggle component 350
 - Toggle Group component 354
 - using 350
- UI Toolkit 53**
 - examples 517
 - overview 502, 503
 - using, to make Editor virtual pet 517, 518
 - using, to make menu with style sheets and animation transitions 537

UI Toolkit package

- installing 503-506

UI Toolkit system

- Panel Settings 507

- parts 506, 509

- Style Sheet 507

- Theme Style Sheet 508, 509

- UI Document 507

Unity Editor

- C# code, writing 528-536

Unity Extensible Markup

- Language (UXML) 507

Unity Frame Debugger 494**Unity Profiler 492, 494****Unity UI optimization strategies 495**

- Layout Groups use, minimizing 495, 496

- multiple Canvases and Canvas

- Hierarchies, using 495

- objects, missing 496

- Raycast computations, reducing 497

- time object pooling, enabling/

- disabling 496, 497

universal design 39**universal design principles 40**

- equitable use 40

- flexibility use 41

- low physical effort 44

- perceptible information 43

- simple and intuitive use 42

- size and space approach 45

- size and space use 45

- tolerance of error 43

user interface (UI) 31, 53

- creating, with UI Builder 511, 512

- defining 3, 4

- designing, for AR 37

- designing, for MR 36

- designing, for VR 33, 34

- making interactable, with C# 513

- particles 457, 458

- reference, getting to UI Documents

- variables 514, 515

- UIElements namespaces 514

- Visual Element events, managing 516

- Visual Element properties,

- accessing 516, 517

V**Vertical Layout Group 115, 116****virtuality continuum**

- reference link 33

virtual reality (VR) 31, 32

- used, for designing user interface (UI) 33, 34

Visual Elements 509-511**visual UI**

- consideration 34, 35

- placement 34, 35

W**World Space**

- working, considerations 477

World Space UI

- using 469-473



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

At www.packtpub.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 Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

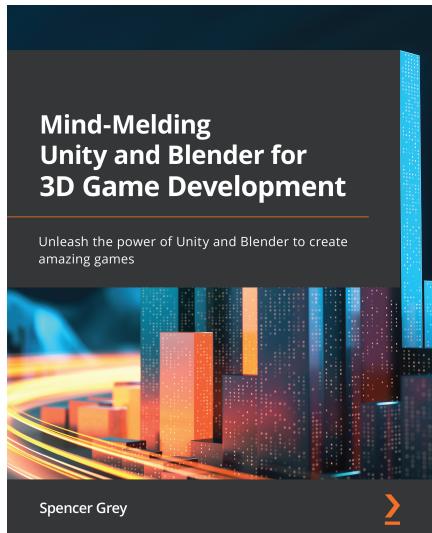


Unity 2022 Mobile Game Development

John P. Doran

ISBN: 978-1-80461-372-6

- Design responsive UIs for your mobile games
- Detect collisions, receive user input, and create player movements
- Create interesting gameplay elements using mobile device input
- Add custom icons and presentation options
- Keep players engaged by using Unity's mobile notification package
- Integrate social media into your projects
- Add augmented reality features to your game for real-world appeal
- Make your games juicy with post-processing and particle effects



Mind-Melding Unity and Blender for 3D Game Development

Spencer Grey

ISBN: 978-1-80107-155-0

- Transform your imagination into 3D scenery, props, and characters using Blender
- Get to grips with UV unwrapping and texture models in Blender
- Understand how to rig and animate models in Blender
- Animate and script models in Unity for top-down, FPS, and other types of games
- Find out how you can roundtrip custom assets from Blender to Unity and back
- Become familiar with the basics of ProBuilder, Timeline, and Cinemachine in Unity

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight 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.

Share Your Thoughts

Hi!

I am Ashley Godbold, author of *Mastering UI Development with Unity*. I really hope you enjoyed reading this book and found it useful for increasing your productivity and efficiency.

It would really help me (and other potential readers!) if you could leave a review on Amazon sharing your thoughts on this book.

Go to the link below to leave your review:

<https://packt.link/r/180323539X>

Your review will help us to understand what's worked well in this book, and what could be improved upon for future editions, so it really is appreciated.



Best wishes,

Dr. Ashley Godbold

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your e-book purchase not compatible with the device of your choice?

Don't worry!, Now with every Packt book, you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the following link:



<https://packt.link/free-ebook/9781803235394>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.