FROM CONCEPT TO PLAYABLE GAME

WITH UNITY® AND C#

New Chapters, Coding Challenges and Expanded Tutorials!

Introduction to GAME DESIGN, PROTOTYPING, and DEVELOPMENT

Third Edition

Jeremy Gibson **BOND**

Foreword by Richard Lemarchand

Praise for the Second Edition

"When teaching about game design and development, you often get asked the dreaded question: 'Where can I learn all this?' *Introduction to Game Design, Prototyping, and Development* has been my deliverance, as it provides a one-stop solution and answer. This book is quite unique in covering in-depth both game design and development: it embraces and exemplifies the idea that design, prototyping, development, and balancing combine in an iterative process. By sending the message that creating games is both complex and feasible, I believe this to be a great learning tool; and the new edition with even more detailed examples seems even better."

—**Pietro Polsinelli**, Applied Game Designer at Open Lab

"*Introduction to Game Design, Prototyping, and Development* has truly helped me in my game development journey and has opened my mind to many helpful techniques and practices. This book not only contains a full introduction to the C# language, but also includes information about playtesting, game frameworks, and the game industry itself. Jeremy is able to explain complex concepts in a way that is very informative and straightforward. I have also found the prototype tutorials to be useful and effective for developing good programming practices. I would highly recommend this book to anyone looking to learn game development from scratch, or simply brush up on their skills. I look forward to using it as a guide and reference for future projects."

—**Logan Sandberg**, Pinwheel Games & Animation

"Jeremy's approach to game design shows the importance of prototyping game rules and prepares the readers to be able to test their own ideas. Being able to create your own prototypes allows for rapid iteration and experimentation, and makes better Game Designers."

—**Juan Gril**, Executive Producer, Flowplay

"*Introduction to Game Design, Prototyping, and Development* combines the necessary philosophical and practical concepts for anyone looking to become a Game Designer. This book will take you on a journey from high-level design theories, through game development concepts and programming foundations. I regularly recommend this book to any aspiring game designers who are looking to learn new skills or strengthen their design chops. Jeremy uses his years of experience as a professor to teach you how to think with vital game design mindsets so that you can create a game with all the right

`string`: A Series of 16-bit Characters

A *string* is used to represent everything from a single character to the text of an entire book. The theoretical maximum length of a string in C# is more than 2 billion characters, but most computers will encounter memory allocation issues long before that limit is reached. To give some context, a little more than 175,000 characters are in the full version of Shakespeare's play *Hamlet*,[7] including stage directions, line breaks, and so on. This means that Hamlet could be repeated more than 12,000 times in a single string.
A string literal is surrounded by double-quote marks:

```
string theFirstLineOfHamlet = "Who's there?";
```

You can access the individual chars of a string using *bracket access*:

```
char theCharW = theFirstLineOfHamlet[0]; // W is the 0th char in the string
char theChart = theFirstLineOfHamlet[6]; // t is the 6th char in the string
```

Placing a number in brackets after the variable name retrieves the character in that position of the string (without affecting the original string). When you use bracket access, counting starts with the number 0. In the preceding example, **W** is the 0th character of the first line of *Hamlet*, and **t** is the 6th character. You will encounter bracket access much more in Chapter 23, "Collections in C#."

tip

If you see any of the following compile-time errors in your Console pane

```
error CS0029: Cannot implicitly convert type 'string' to 'char'
error CS0029: Cannot implicitly convert type 'char' to 'string'
error CS1012: Too many characters in character literal
error CS1525: Unexpected symbol '<internal>'
```

it usually means that somewhere you have accidentally used double quotes (`" "`) for a char literal or single quotes (`' '`) for a string literal. String literals always require double quotes, and char literals always require single quotes.

`class`: The Definition of a New Variable Type

A *class* defines a new type of variable that can be best thought of as a collection of both variables and functionality. All the Unity variable types and components listed in the "Important Unity Variable Types" section later in this chapter are examples of classes. Chapter 26, "Classes," covers classes in much greater detail.

7. http://shakespeare.mit.edu/hamlet/full.html — accessed August 14, 2021.

The Scope of Variables

In addition to variable type, another important concept for variables is *scope*. The scope of a variable refers to the range of code in which the variable exists and is understood. If you declare a variable in one part of your code, it might not have meaning in another part. This is a complex issue that will be covered throughout this book. If you want to learn about it progressively, just read the book in order. If you want to get a lot more information about variable scope right now, you can read the section "Variable Scope" in Appendix B, "Useful Concepts."

Naming Conventions

The code in this book follows a number of rules governing the naming of variables, functions, classes, and so on. Although none of these rules are mandatory, following them will make your code more readable not only to others who try to decipher it but also to yourself if you ever need to return to it months later and hope to understand what you wrote. Every coder follows slightly different rules—my personal rules have even changed over the years—but the rules I present here have worked well for both me and my students, and they are consistent with most C# code that I've encountered in Unity:

1. Use *camelCase* for pretty much everything (see the "CamelCase" sidebar).

CAMELCASE

CamelCase is a common way of writing variable names in programming. It allows the programmer or someone reading their code to easily parse long variable names. Here are some examples:

- `aVeryLongNameThatIsEasierToReadBecauseOfCamelCase`
- `variableNamesStartWithALowerCaseLetter`
- `ClassNamesStartWithACapitalLetter`

The key feature of camelCase is that it allows multiple words to be combined into one with a medial capital letter at the beginning of each original word. It is named *camelCase* because it looks a bit like the humps on a camel's back.

2. Variable names should start with a lowercase letter (e.g., **`someVariableName`**).
3. Function names should start with an uppercase letter (e.g., **`Start()`**, **`FixedUpdate()`**).

4. Class names should start with an uppercase letter (e.g., `GameObject`, `ScopeExample`).
5. Private variable names can start with an underscore (e.g., `_hiddenVariable`).
6. Static variable names can be all caps with snake_case (e.g., `NUM_INSTANCES`). As you can see, snake_case combines multiple words with an underscore in between them.

For your later reference, this information is repeated and expanded in the "Naming Conventions" section of Appendix B.

Important Unity Variable Types

Unity has a number of variable types that you will encounter in nearly every project. All of these variable types are actually classes and follow Unity's naming convention that all class types start with an uppercase letter. For each of the Unity variable types, you will see information about how to create a new *instance* of that class (see the later sidebar "Class Instances and Static Functions") followed by listings of important variables and functions for that data type. For most of the Unity classes listed in this section, the variables and functions are split into two groups:

- **Instance variables and functions:** These variables and functions are tied directly to a single instance of the variable type. If you look at the *Vector3* information that follows, you will see that `x`, `y`, `z`, and `magnitude` are all instance variables of Vector3, and each one is accessed by using the name of a specific Vector3 variable, a period, and then the name of the instance variable (e.g., `position.x`). Each Vector3 instance can have different values for these variables. Similarly, the `Normalize()` function acts on a single instance of Vector3 and sets the `magnitude` of that instance to 1.
- **Static class variables and functions:** *Static* variables are tied to the entire class rather than being tied to an individual instance. These are often used to store information that is the same across all instances of the class (e.g., `Color.red` is a static variable that defines `red` for the entire class Color; as a result, `red` has the same meaning and value across all instances of Color) or to act on multiple instances of the class without affecting either (e.g., `Vector3.Cross( v3a, v3b )` is used to calculate the cross product of two Vector3s and return that value as a new Vector3 without changing either `v3a` or `v3b`).

For more information on any of these Unity types, check out the Unity documentation links referenced in the footnotes.

CLASS INSTANCES AND STATIC FUNCTIONS

Just like the prefabs that you saw in Chapter 19, "Hello World: Your First Program," classes can also have *instances*. An instance of any class (also known as a *member* of the class) is a data object that is of the type defined by the class.

For example, you could define a class **`Human`**, and everyone you know would be an instance of that class. Several functions are defined for all instances of the **`Human`** class (e.g., **`Eat()`**, **`Sleep()`**, **`Breathe()`**).

Just as you differ from all other humans around you, each instance of the **`Human`** class differs from the other instances. Even if two instances have perfectly identical values, they are stored in different locations in computer memory and seen as two distinct objects. (To continue the human analogy, you could think of them as identical twins.) Class instances equality is tested by *reference*, not value. This means that if you are comparing two class instances to see whether they are the same, the thing that is compared is their *location in memory*, not their values (just as two identical twins have different names, i.e., references).

It is, of course, possible to reference the same class instance using more than one variable. Just as the person I might call "son" would also be called "grandson" by my parents, a class instance can be assigned to any number of variables yet still be the same data object, as is shown in the following code:

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

// Defining the class Human
[System.Serializable] // Allows this class to be seen in the Inspector
public class Human {
    public string name;
    public Human  partner;
}

public class Family : MonoBehaviour {
    // Declaration of public variables
    public Human husband;
    public Human wife;

    void Start() {
        // Initial state
        husband = new Human();
        husband.name = "Jeremy Gibson";
21          wife = new Human();
22          wife.name = "Melanie Schuessler";
23
24          // My wife and I get married
25          husband.partner = wife;
26          wife.partner = husband;
27
28          // We change our names
29          husband.name = "Jeremy Gibson Bond";
30          wife.name = "Melanie Schuessler Bond";
31
32          // Because wife.partner refers to the same instance as
33          //  husband, the name of wife.partner has also changed
34          print( wife.partner.name );
35          // prints "Jeremy Gibson Bond"
36      }
37 }
```

It is also possible to create *static functions* on the class **Human** that are able to act on one or more instances of the class. The following static function **Marry()** allows you to set two humans to be each other's partner with a single function as is shown in the following code.

```
37      // ~~}~~ // Comment out the closing brace on line 37.
38      static public void Marry( Human h0, Human h1 ) {
39          h0.partner = h1;
40          h1.partner = h0;
41      }
42 }
```

With this function, lines 25 and 26 from the initial code listing can now be replaced with the single line **Family.Marry(wife, husband);** . Because the **Marry()** method[8] is preceded by the **static** keyword, it can be used almost anywhere in your code. You will learn more about static variables and functions later in the book.

8. *Function* and *method* have the same basic meaning. The only difference is that *function* is the word for a standalone function whereas *method* refers to a function that is part of a class, as the **Marry()** method is part of the class **Family**.

Vector3: A Collection of Three Floats

Vector3[9] is a very common data type for working in 3D. It is used most commonly to store the three-dimensional position of objects in Unity. Follow the footnote for more detailed information about Vector3s.

```
Vector3 position = new Vector3( 0.0f, 3.0f, 4.0f ); // Sets the x, y, & z values
```

Vector3 Instance Variables and Functions

As a class, each Vector3 instance also contains a number of useful built-in values and functions:

```
print( position.x );  // 0.0, The x value of the Vector3 position from above
print( position.y );  // 3.0, The y value of the Vector3 position
print( position.z );  // 4.0, The z value of the Vector3 position
print( position.magnitude ); // 5.0, The distance of position from 0,0,0
                             //  Magnitude is another word for "length".
position.Normalize(); // Sets the magnitude of position to 1, in this case
                      //  setting the x, y, & z values to [0.0, 0.6, 0.8]
```

Vector3 Static Class Variables and Functions

In addition, several static class variables and functions are associated with the Vector3 class itself:

```
print( Vector3.zero );     // (0,0,0), Shorthand for: new Vector3( 0, 0, 0 )
print( Vector3.one );      // (1,1,1), Shorthand for: new Vector3( 1, 1, 1 )
print( Vector3.right );    // (1,0,0), Shorthand for: new Vector3( 1, 0, 0 )
print( Vector3.up );       // (0,1,0), Shorthand for: new Vector3( 0, 1, 0 )
print( Vector3.forward );  // (0,0,1), Shorthand for: new Vector3( 0, 0, 1 )
Vector3.Cross( v3a, v3b ); // Computes the cross product of the two Vector3s
Vector3.Dot( v3a, v3b );   // Computes the dot product of the two Vector3s
```

This is only a sampling of the fields and methods affiliated with Vector3. To find out more, check out the Unity documentation referenced in the footnote.[10]

Color: A Color with Transparency Information

The *Color*[11] variable type can store information about a color, including its transparency (alpha value). Colors on computers are mixtures of the three primary colors of light: red, green, and blue. These are different from the primary colors of paint you may

9. https://docs.unity3d.com/2020.3/Documentation/ScriptReference/Vector3.html. Note that references to Unity documentation are all for Unity version 2020.3 — all accessed August 13, 2021.

10. For more information on Dot Products, please see Appendix B, "Useful Concepts."

11. https://docs.unity3d.com/2020.3/Documentation/ScriptReference/Color.html — accessed October 24, 2020.