2D Game Development with Unity

Franz Lanzinger

CRC Press
Taylor & Francis Group

2D Game Development with Unity

2D Game Development with Unity

Franz Lanzinger

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

First edition published 2021
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

ISBN: 978-0-367-34911-0 (hbk)
ISBN: 978-0-367-34907-3 (pbk)
ISBN: 978-0-429-32866-4 (ebk)

Typeset in Minion
by codeMantra

Contents

Acknowledgments

A BIG THANK YOU TO THE MANY PEOPLE AND ORGANIZATIONS who made this book possible.

First and foremost, thank you Atari Coin-Op! That's where I got my start in the game industry at the ripe old age of 26 in 1982. Unfortunately, Atari Coin-Op no longer exists, but the unparalleled influence of that small group of pioneers continues to this day.

A HUGE thank you to the thousands of people who built and continue to build Unity, Blender, GIMP, Audacity, and MuseScore. This book relies heavily upon their contribution and generosity for making this valuable software available at no cost to the developer.

Thank you to the millions of players who love games and play them maybe just a little too much. Without you, none of this would exist, not the games, the game companies, nor the thousands of jobs and careers in game development.

Thank you to Rick Adams of CRC Press, who has been a real joy to work with.

If I forgot to thank you, please insert your name here. You know who you are.

Finally, thank you Susan for your love and support throughout the years.

Author

Franz Lanzinger is an independent game developer, author, and pianist. He is the owner of Lanzinger Studio located in Sunnyvale, California. His game development career spans almost forty years starting with the coin-op classic *Crystal Castles* at Atari in 1982 and continuing with *Ms. Pac-man* and *Toobin'* for the NES, published by Tengen in 1990.

Mr. Lanzinger has been an indie game developer since 1991. He worked on SNES' *Rampart*, *Championship Pool*, and *NCAA Final Four Basketball*, as well as *Gubble* for the PC, Mac, and Playstation. He is currently working on a remake of *Gubble* for PC, Mac and consoles, using Unity and Blender. In his spare time, he teaches piano and works as an accompanist for the Valley Chorale and the Serendipity Choir. Go to franzlanzinger.com for the latest news about Mr. Lanzinger.

Introduction and Overview

In this book, you'll learn to develop 2D games using Unity. *2D Game Development with Unity* combines a practical, hands-on, step-by-step approach with explanations of the theory behind it all. This book covers the major aspects of 2D game development with Unity. In addition to Unity, you'll use Blender and GIMP for graphics creation, Audacity for sound effects, MuseScore for music, and SourceTree for version control. All of this software is free to use, and much of it is open source. If you carefully work through this book, you'll learn a great deal. You'll be ready to make your own original games, whether as a solo developer, as a contributor to a small team, or as an employee at a large game company.

This book is divided into two parts. Part I explores the tools and theory behind 2D game development. You'll make a few small games, and a larger one, to get started. Then, in Part II, you'll build a 2D game with many of the features typically found in commercial games.

You'll experience the joys and occasional frustrations of game development: The awesome feeling of making your character move for the first time, and the pain of thinking it's going to work, only that it doesn't, and you have no clue as to why. It's all part of the process, and there's nothing quite like it.

In the first two chapters, you'll start with a thorough review of the foundations: math and coding. You should already know some algebra, geometry, and trigonometry. A background in calculus and more advanced university level math is not needed for this book, but it can be very helpful for advanced game development. You should have at least some coding experience, preferably in some dialect of C. If you're brand new to coding, you can still follow along with this book, but you would benefit from learning more about coding ahead of time before diving into this book.

In subsequent chapters, you'll explore the creation of art, music, and sound for games. And, of course, you'll learn how to use Unity to put all that art, music, and sound together to make games.

Part II is devoted to making a single, larger 2D game. You'll learn about coding in C#, using Unity, creating characters, controlling the characters, GUI creation, debugging, testing, and much more.

You are strongly encouraged to follow along and build the games as you're reading this book. That is the best way for you to learn. Actually, it's the only way! All code and game assets are available for download at franzlanzinger.com, so you don't really need to type in the code or draw anything, but you'll learn much more if you build everything "from

scratch" along with this book. You'll also have the opportunity to branch out and do things a bit differently from this book. That's the main advantage of doing game development yourself. You then have total control over your game. Soon you'll know enough to make that next awesome, original hit game.

Game development can be a daunting, exciting, and highly rewarding endeavor. There's a lot to learn, and some aspects may seem difficult at first. Don't let that stop you! It's incredibly fun and satisfying, so go and make some games!

I

The Basics of Game Development

First Steps

In this chapter, you'll take your first steps as a game developer. You'll install Visual Studio and Unity. You'll explore C# and some basic mathematical concepts. Best of all, you'll make your first game, a simple game in Visual Studio. Your goal is to learn how to use these game development tools. There's no better way than to dive right in and make something with them.

This book was written to let you follow along with a series of steps. By doing this, you will experience firsthand what it's like to be a game developer. There are hundreds of numbered steps in this book, requiring you to pay very close attention and to do them in order, one by one. Many of the step descriptions are followed by additional explanations, descriptions, or screen shots.

You'll need to have access to a PC or Mac to follow along with the steps in this book. Your system needs to meet the development system requirements for Unity. In January 2020, the Unity company released the 2019.3 version of Unity. That is the version used for this book. Here is a summary of the system requirements for that version of Unity:

- OS: Windows 7 and Windows 10, 64-bit versions only; macOS 10.12.6+.
- CPU: X64 architecture with SSE2 instruction set support.
- Graphics API: DX10, DX11, and DX12-capable GPUs. Mac Metal capable.

Look at *unity.com* for details. While not strictly necessary, it's highly recommended that you have at least 1920×1080 resolution for your screen. A multiple monitor setup with a fairly recent graphics card is a good idea in order to enjoy Unity even more. If you don't have multiple monitors, the use of Virtual Desktops in Windows, or Mission Control on your Mac, is a worthwhile alternative.

If your system meets the requirements listed above, it will also meet the requirements for the other software used in this book. And yes, you'll need fairly fast internet access, the faster the better. Some of your downloads will be large and time consuming, but once your software is installed, you'll be able to work on your game without internet access.

If you're like most game developers, you'll have access to several systems, old and new, laptops and desktops. Your older laptops may not be compatible with Unity, but they can likely be used for doing Visual Studio projects and for the creation of graphics, sound, and music. They may be good enough for testing your games. Testing on older systems is an important facet of game development if you intend to distribute your game for play on PCs or Macs.

This book focuses on the development of games for PCs and Macs. Unity got its name from its ability to create games once and deploy them to many targets such as desktops, consoles, and mobile devices. All games in this book will run on PCs and Macs, and most of them can be modified to run on consoles and/or mobile devices with some effort. The process of taking a game from an existing platform to another one is called *porting*. Porting is easy in Unity when compared to writing your own game engine and getting it to work on all of your target platforms. Each platform has a long list of specific requirements, especially if your goal is to produce a commercial release. The number of supported platforms for Unity stands at 25 as of 2020. You'll need to do your own research on the current details of porting to, and developing for your intended target platforms.

It's easier to learn the basics of C# inside of Visual Studio rather than Unity. You'll want to have Visual Studio installed anyhow when using Unity, so in the next section you'll install Visual Studio.

VISUAL STUDIO

Visual Studio is Microsoft's suite of development tools. It supports a plethora of languages, among them C#. In this section, you'll install the free version of Visual Studio. The Visual Studio 2019 community edition can be downloaded at *visualstudio.microsoft.com*. Go ahead and install this. Under Workloads select "Universal Windows Platform development." On a Mac, follow the specific instructions for Mac installation. This is a somewhat large install, so be sure you have enough disk space and time to download and install.

To test out whether the install worked, the first step is always to write a "Hello World!" program. This is a minimal program that only displays the text "Hello World!".

<Step 1> Run Visual Studio 2019 Community Edition.

<Step 2> Sign in or create a Microsoft Account, if necessary. This is only needed the first time you run Visual Studio.

<Step 3> Click on "Create a new project" in the Get started panel, as shown in Figure 1.1.
On a Mac, this interface looks different, but you'll also see a way to create a new project.

<Step 4> Click on "Language" and select C#.

<Step 5> Click on C# Console App (.NET Framework) or (.NET Core). Compare your screen with Figure 1.2.

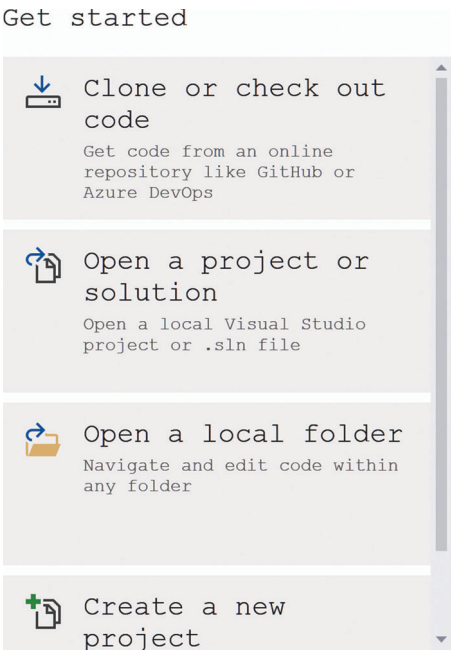

FIGURE 1.1 The Get started panel in Visual Studio 2019.

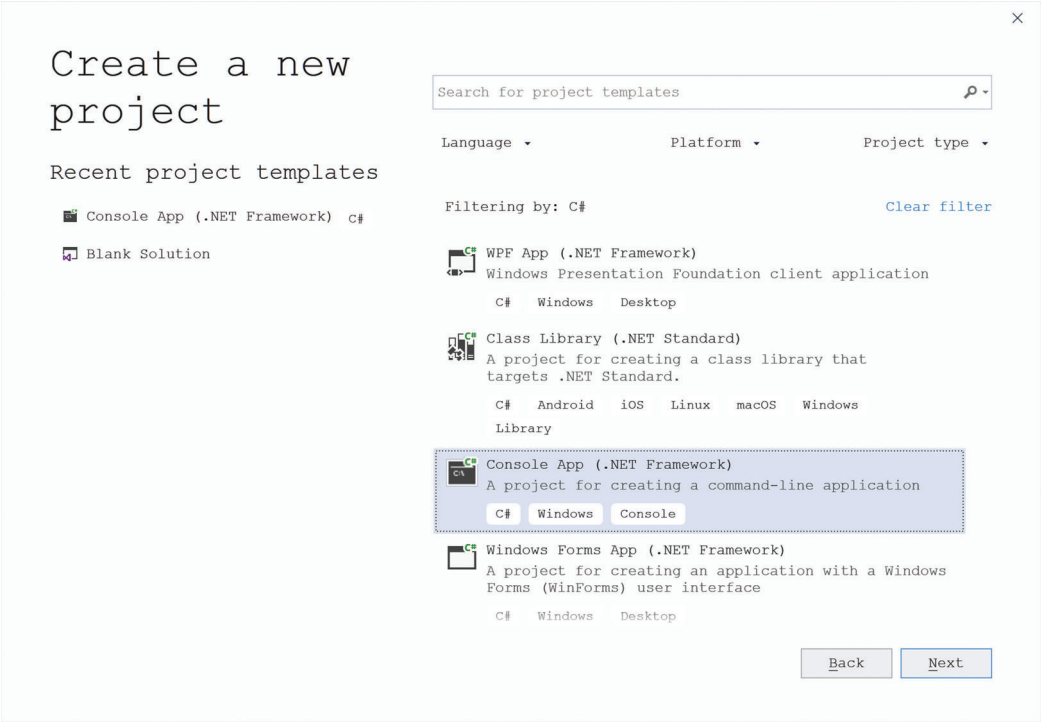

FIGURE 1.2 Creating a new project in Visual Studio 2019.

<Step 6> Click on Next.

<Step 7> Enter Project Name "Hello World".

On a Mac, use the project name "HelloWorld". On a Mac, project names may not contain spaces, nor can you have an exclamation mark.

<Step 8> Optional: Enter a project location.

This would be a good time to set up a folder for all the projects for this book. It's up to you to name and create that if you wish.

<Step 9> Click Create.

<Step 10> Add this line of code inside the Main function:

```
Console.WriteLine("Hello World!");
```

Your screen should now look similar to Figure 1.3.

On a Mac, you won't have to type in the Console.WriteLine line of code, as it's there already. Also, there's only a single "using" line.

<Step 11> Control-F5 to compile and run it.

This step automatically saves your work, compiles it, and runs it. When running the console app, a popup window appears and prints "Hello World!" in the window followed by "Press any key to continue . . .". And before you go hunting for the "Any" key on your keyboard, just press the spacebar. There is no

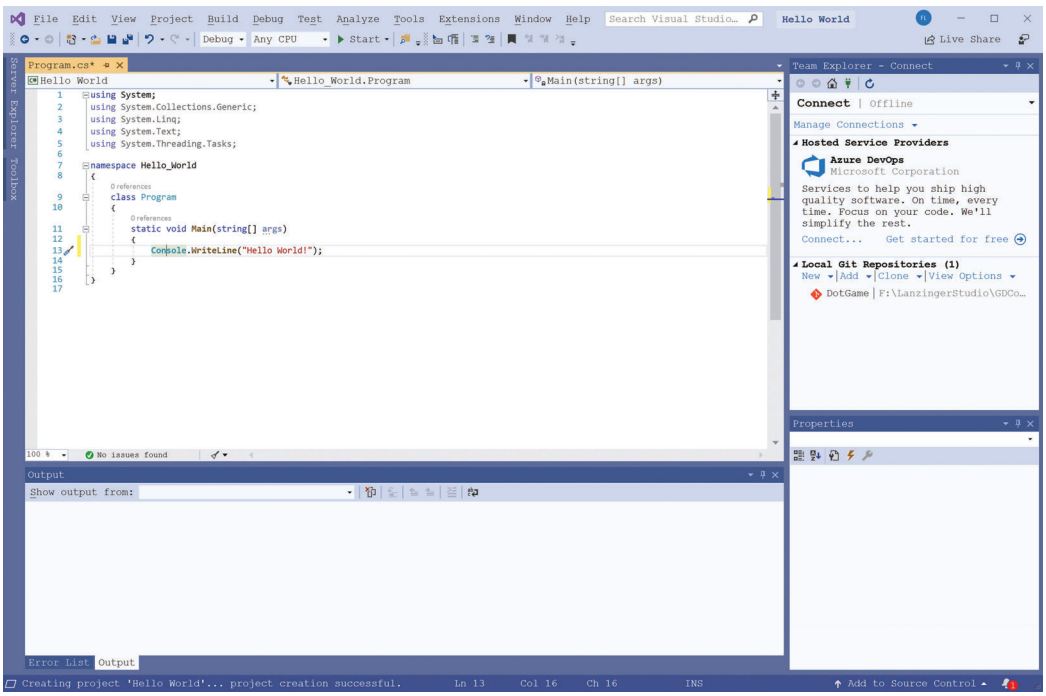

FIGURE 1.3 Hello World program in Visual Studio.

"Any" key on your keyboard. Search for "any key video" online for some very funny videos about this dilemma.

<Step 12> Exit.

You're now ready to use Visual Studio 2019. In the next few sections, you'll learn the basics of C# and make a simple game using Visual Studio.

WHAT IS C#

C# (pronounced C Sharp) is the programming language used in this book and by Unity. C# is one of the many descendants of the granddaddy of them all, C. As of 2019, C# is one of the top programming languages in terms of both job demand and popularity. As a side benefit to reading this book, you'll learn the basics of C#, and programming in general.

To get started with game development, it is helpful to first learn your programming language well enough to do the job. C# is a large language, but only a relatively small subset of C# is necessary to make games in Unity.

Basic computations often look identical in C and C#. For example, the following code runs and produces the same result in both languages:

```
int i;
int x = 0;
for (i=0; i<10; i++)
{
  x = x + i;
}
// the value of x is 45 at the end of this computation.
```

Before learning more about C#, you'll start by examining different kinds of numbers and how they are represented on computers.

NUMBERS

Numbers are the fundamental entities for Computer Game Development. Graphics, sound, and game logic are all represented by numbers in a computer game. In this section, you'll review some of the different types of numbers in mathematics and then look at how they are represented on your computer and in C#.

Here are some common types of numbers in mathematics: natural numbers, integers, rational numbers, real numbers, and complex numbers. Let's examine each of them.

Natural numbers are used for counting things. They start at one, so zero isn't considered a natural number. Mathematicians view numbers as unlimited. There is no such thing as a largest natural number. The set of natural numbers is infinite.

Integers introduce the concept of *negative number*. They are used for counting things that could possibly be zero or negative, such as a bank balance, for example.

Rational numbers are numbers of the form a/b with a and b integers, where b must be non-zero. Examples are 1/3, 12/35, –1/1000, 34/–1. *Decimal numbers* such as 10.43 are a special category of rational numbers because they can always be written as a rational number, for example 10.43 = 1043/100.

Real numbers are numbers with a possibly infinite number of decimal digits, such as pi = 3.1415926… or the square root of 2 = 1.4142… Real numbers are important for advanced mathematics, but because of their infinite nature, they cannot be directly represented on computers. Some real numbers are called *irrational* as they cannot be represented as a rational number, for example the aforementioned square root of 2.

Complex numbers are numbers of the form a+bi, where a and b are real numbers and i is the square root of –1. Complex numbers are rarely used in game development, but are commonly used in advanced mathematics.

It is important to distinguish between numbers in mathematics and numbers in computer programs. In mathematics, there is no upper limit for the size of numbers, but on a computer, the fixed amount of available memory requires a limit on the size of numbers. The next section shows the implications of this in C#.

INTS, FLOATS, AND DOUBLES

In this section, you'll explore three types of numbers in C#. Here's an example:

```
int i = 17;
float x = 17.0f;
double y = 17.0;
```

What we have here is the number 17 represented in three different ways. The first is as an int, the second as a float, and the third as a double. You would call all of these "seventeen," but on your computer, they are represented in three very different ways. The int version is a 32-bit integer, stored in four consecutive bytes as `0x00000011`. That x stands for hexadecimal. The float version also uses up 32 bits, but it's a different bit pattern: `0x41880000`. The double version uses 64 bits stored in eight consecutive bytes as follows: `0x4031000000000000`. You don't need to understand the details of how this works, just that floats and doubles are stored in this way. Game developers tend to use 32-bit integers and floats, mainly because they are adequate for most games as well as faster and more space-efficient than the alternatives.

Notice the letter `f` at the end of the float 17. That is the way you tell C# that the number is a float rather than a double. When coding in Unity, this distinction becomes important as you will see later on in this book.

It's time to experiment with these numbers by writing some code using ints, floats, and doubles.

<Step 1> In Visual Studio, create a new C# console project and call it NumberTest.

<Step 2> Insert the following code into Main:

```
int i = 17;
float x = 17.0f;
double y = 17.0;
Console.WriteLine("i={0},x={1},y={2}", i, x, y);
```

If you're on a Mac, you can leave the Hello World! line at the beginning.

<Step 3a> For Mac users only: Visual Studio – Preferences – Key Bindings – Scheme. Select Visual Studio (Windows). Click on OK.

This will make your keyboard shortcuts the same as for Windows. For example, to save a file is <Ctrl> S rather than <command> S. In this way, the keyboard shortcuts mentioned in this book work both on Windows and on Macs. You may use the Mac Key Bindings instead if you prefer those, but then you won't match the book's instructions in a few places. Be aware that this setting applies to Visual Studio only, so on other applications such as Unity or Audacity you'll be using <command> instead of <Ctrl>.

<Step 3b> Run NumberTest using <Ctrl>-F5. The output from this should be

```
i=17,x=17,y=17
Press any key to continue . . .
```

This really simple exercise shows how to declare variables, assign values to them, and then to access those values in a `Console.WriteLine` statement. All three variables show 17 as their value. Next, you're going to do some computations with those numbers.

<Step 4> Insert the following code at the end of Main after the existing `Console.WriteLine` statement:

```
i = i + 1;
x = x * 2;
y = y / 3.0;
Console.WriteLine("i={0},x={1},y={2}", i, x, y);
```

<Step 5> Again, run the code using <Ctrl>-F5. The output from this should be

```
i=17,x=17,y=17
i=18,x=34,y=5.66666666666667
Press any key to continue . . .
```

The variable `i` is one larger, `x` got doubled, and `y` divided by 3. `y` isn't really 17/3 but rather a close approximation. Because `y` is declared as a `double`, it has about 15 significant digits.

<Step 6> Change the declaration of `y` to `float` and run the program again.

You probably got two errors when you did that. That's because you also need to add an `f` at the end of the `17.0` in the declaration, and the `3.0` in the division. You are also seeing the location of the errors underlined in red. To see the error messages, on a Mac, click on the red error message on top. On a PC, look at the Error List in the panel at the bottom. You may need to expand the Error list panel by dragging the boundary up.

Visual Studio can be helpful for avoiding C# coding errors. It won't catch all coding errors right away, but if you see a red squiggly underline in the code editor, chances are high that you goofed. It's useful to know that hovering the mouse over a squiggly red line will pop up a short explanation.

<Step 7> Fix the errors by adding f's after the constants and try again.

The value of `y` is now `5.666667`. The number of printed digits for `y` is now 7 instead of 15 because you have only 32 bits instead of 64 bits available to store the number. You'll now move on and test out some additional ways to do computations in C#.

<Step 8> Add the following code to Main:

```
double z;
i = 17 % 12;   // remainder function
x = (float)i + 3.0f;   // casting an integer to a float
z = Math.Sqrt(2.0);   // square root of 2
y = 1.0f / (float)Math.Sin(Math.PI * 0.25); // sqrt(2)
Console.WriteLine("i={0},x={1},y={2},z={3}", i, x, y, z);
```

Then run it and compare your results with this:

```
i=5,x=8,y=1.414214,z=1.4142135623731
```

Try to understand what's going on here. The variable `i` is computed as the remainder of dividing 17 by 12, thus 5. The expression `(float)i` converts the integer i to a float. This is called *casting*. The variable z, a double, is used to store the result of the built-in `Math.Sqrt` function. The Math functions use doubles, so if your code primarily uses floats, you'll often need to cast the results of Math functions to floats as indicated in the computation for y. That strange expression to compute y also results in the square root of 2. Oh, and the letters after the double slashes are comments and are ignored by C#.

<Step 9> Save your project.

You have just experienced one of the basic principles of good programming practice. Writing small bits of test code to try out features of a programming language is the best way to learn to use these features. You can freely experiment and even try to break things. When you're comfortable with your test code and fully understand it, you'll be ready to use what you learned in your actual project.

You're now ready to tackle the next topic, random numbers.

RANDOM NUMBERS

In this section, you'll learn about random numbers in game design. Random numbers have been used in computer games since the very beginning. They are essential for any game with random elements such as shuffling a virtual deck of cards, slot machines, dice, and procedurally generated data. They are also great for fooling your players into thinking that you created sophisticated AI, when in fact your creatures just got lucky and acted smart.

The first thing to realize is this: Random numbers on computers aren't really random. They are generated using a deterministic algorithm which generates a sequence of numbers that appear to be random. The numbers appear random because these sequences pass certain statistical tests, but under the hood, they aren't random at all. It is possible to build

hardware that generates truly random numbers, but this is actually a bad thing for software development. You want to be able to recreate your "random" number sequence for testing purposes.

This brings us to the concept of a *seed*. Computer generated pseudo-random number sequences start with a seed, which is just another word for the first number in the random sequence. The random number algorithm computes each number in the sequence using only the previous number as input. If you want to recreate the same sequence, you simply start with the same seed. If you want a different random sequence compared to the last time you executed the program, you start with a different seed. Commonly used techniques to make these random numbers seem more random are to base the seed on the system clock or to increment the seed from game to game. Feel free to search the internet for additional discussions about randomness and random number generators.

In the following project, you will explore how to create and use random numbers in your C# based games.

<Step 1> In Visual Studio, create a new C# console project with the name "RandomTest".

<Step 2> Enter the following code to Main and run it multiple times:

```
Random rnd = new System.Random();
int randomnumber = rnd.Next(1,10);
Console.WriteLine("randomnumber = {0}", randomnnumber);
```

It's highly likely that you will see different numbers on subsequent program executions.

This code needs some explanation. The first line creates a new random number generator and you called it "rnd". The rnd.Next call returns a random number between 1 and 10. The result will be greater than or equal to 1, but strictly less than 10.

<Step 3> Go and insert a seed number into the System.Random() call like this:

```
Random rnd = new System.Random(42516);
```

Now, when you run the code, you'll get the same output every time because the seed is the same.

In the next step, you'll generate random doubles.

<Step 4> Remove the seed, then insert the following code:

```
double randomdouble = rnd.NextDouble();
Console.WriteLine("randomdouble = {0}", randomdouble);
```

You should see random doubles greater than or equal to 0.0 and less than 1.0.

NUMBER GAME

This is a book about game development, so it's about time for you to make your first game. This is a simple number guessing game. There's not much to the design, but don't let that fool you. It's not going to be all that easy.

The core of the design is this: The game asks the player to guess an integer between 1 and 10 inclusive. The "inclusive" means that 1 and 10 are valid guesses. If the player guesses correctly, he wins. If not, the game gives a hint about the number. If the player guesses too low, the game prints "higher," or else "lower." Then, the player gets to try again.

This game is simple enough that you won't need to use Unity. A small Visual Studio Console app written in C# is perfectly adequate.

<Step 1> Create a Visual Studio console app named "NumberGuess".

<Step 2> Add code which prints a message that asks the player to guess a number, then exits. Run your code.

Here comes the most important lesson in this book. Are you ready? If you learn nothing else, learn this: Write your code one small piece at a time, and immediately test each piece before you go on to create the next one. This isn't a game at all yet, just a console app that prints some text and exits. It's basically Hello World with different text. The book doesn't tell you right away how to code this. Here's your chance to show that you remember how to do this. Or, if you don't remember how to output text in C#, you can look it up from a previous code listing in this book. Your code should look similar to this:

```
static void Main(string[] args)
{
    Console.WriteLine("Guess the number from 1 to 10: ");
}
```

Next, you'll generate the number that the player needs to guess.

<Step 3> Put in code that generates a random integer from 1 to 10. Do this before asking for the guess.

Here is the code for this step:

```
static void Main(string[] args)
{
    Random rnd = new Random();
    int number = rnd.Next(1, 10);
    Console.WriteLine("Guess the number from 1 to 10: ");
}
```

Well, this code runs, but it appears to execute the same way as the previous version. To help test it you'll temporarily put in a WriteLine statement so you can see the value of number.

<Step 4> Insert the following statement immediately after the `rnd.Next` statement:

```
Console.WriteLine("number = {0}",number);
```

You can now run the code and check that the number being generated is indeed between 1 and 10. There is one problem though. How do you know that all the possible numbers 1, 2, 3, …, 8, 9, 10 are being generated with equal frequency? Here is some code that checks this:

```
for (int i = 1; i <= 10; i++)
{
    int counter = 0;
    for (int j = 1; j < 1000; j++)
    {
        number = rnd.Next(1, 10);
        if (number == i) counter++;
    }
    Console.WriteLine("Frequency of {0} is {1} out of 1000",
                      i, counter);
}
```

When you run this code, you'll see a problem. The frequency for the numbers from 1 to 9 is about 100, but 10 has a frequency of 0. The problem lies with the `rnd.Next(1,10)` statement. It generates random numbers between 1 and 10, but not 10 itself. To fix this, you need to change the 10 to an 11. Now when you run the code everything looks good and you may remove the test code. Because you hate to throw this code away, you do the following. Instead of deleting it, comment it out by surrounding it with /* and */.

You're finally ready to move on and do the next step.

<Step 5> Insert the following code:

```
int guess;
bool valid;
valid = Int32.TryParse(Console.ReadLine(), out guess);
if (valid) if (guess == number)
{
   Console.WriteLine("You Win");
   return;
};
Console.WriteLine("You Lose");
```

This code lets the player enter one string, and if it matches the random number the player wins, otherwise he loses. Then the game exits. This code introduces several new C# concepts.

The variable `valid` is declared to be `bool`. A `bool` (pronounced Boolean after George Boole) has just two possible values, `true` or `false`. The next line reads a string from the console, converts it to a 32-bit integer, and stores the result in the variable guess. If the players enter an invalid input the `TryParse` call fails and returns false. Otherwise the code compares the guess with the number and if they are equal the player wins.

As you can see, it took quite a bit of code to deal with something as simple as getting an integer from the player and testing it, but that's the nature of programming. In coding, things are often quite a bit more difficult that you might at first expect. This is why it's so very difficult to predict the time it'll take to program something, even if it's very simple.

To make matters worse, the game isn't even done yet. There's still the matter of looping back to the beginning to give the player additional chances at guessing the number. You also need to add code to explain to the player what went wrong when the input isn't a valid number.

<Step 6> Replace the code after the declaration of `valid` with this:

```
while (true)
{
    valid = Int32.TryParse(Console.ReadLine(), out guess);
    if (valid) if (guess == number)
       { Console.WriteLine("You Win"); return; };
    if (!valid)
    {
        Console.WriteLine("Invalid input, try again");
        continue;
    }
    if (guess<number)
        Console.WriteLine("too low, please try again");
    else
        Console.WriteLine("too high, please try again");
}
```

Congratulations! You reached your first major milestone, the first playable version of your first game! You're almost finished with the game, and it's playable. It's not really fun, but that doesn't matter right now. Your goal was to learn how to create a simple, playable game.

The next step will fix the most glaring problem: the code tells you what the number is that you're trying to guess, but that surely makes the game way too easy.

<Step 7> Remove the output of `number`. Then play the game and try to win.

Now you're getting somewhere. It's no longer totally obvious how to win. Rather than continue the development of this game, it's time to wrap things up by putting in a quick title.

<Step 8> Add the following line at the beginning of Main:

```
Console.WriteLine("Number Guess");
```

You're going to shelve this game now. It's just a small exercise on your way to making games in Unity.

<Step 9> Save and exit Visual Studio.

IMPORTANT: NOTES FOR MAC USERS

If you are following along with this book on a Mac, here are some important notes. Macs can be wonderful development machines, so by all means, continue to use your Mac if that is your preferred computer. You can skip this section if you're only using Windows.

All of the software tools used in this book are available both for Windows 10 and for Mac OS. There can be slight differences between Mac and Windows versions, but for the most part they are close to the same. The devil lies in the details. The keyboard shortcuts are different sometimes, and the screenshots do not match exactly, or in some rare instances not at all.

From here on in, this book will usually assume that you're using Windows. You can still follow along on your Mac, but you will need to adjust the keyboard shortcuts, and you'll

have to live with the slightly different screen shots. Occasionally, the Menu structures are different, but the same set of choices is always available. It helps to use the same software version numbers as the corresponding Windows version number recommended in the book. Also, be sure to go to www.franzlanzinger.com for the latest compatibility notes and additional help for Mac users.

INSTALLING UNITY

In this section, you'll install Unity and take it for a quick spin. Go to *www.unity.com* and install the Personal, Plus, or Pro edition of version 2019.3.0f6. You'll need to check the financial eligibility conditions to see which version you are eligible to use. This book is compatible with all three editions. This book was produced and tested with the Personal Edition, version 2019.3.0f6. Check the author website www.franzlanzinger.com for the latest compatibility information if you are interested in following along with this book using a different version of Unity.

This book was originally developed on Windows. The screen captures were done on the author's 4K monitor running Windows 10. On a Mac, your screen might look a little different from the corresponding Windows screen captures. Also, make sure to read or reread the previous section for Mac users. Note that the author tested on a 2015 iMac using Unity Version 2018.2.8f1, which is similar but not exactly the same as Windows Unity 2019.3.0f6. See the author website for more details about this.

Unity is a real-time development platform for making games and similar applications for a wide variety of targets. With Unity, you can develop for PCs, Macs, game consoles, VR, and mobile devices. The name "Unity" implies that you can develop your game once and then deploy it to many platforms. It is without a doubt the world's most popular game engine with half of all released games developed using Unity, according to a rough estimate by Unity's CEO in 2018. Only a few very large game development studios take the plunge and write their own game engines for their games.

Now that you've installed Unity your logical next goal will be to create a "Hello World" project. This will be quite a bit different from the Hello World project for Visual Studio. No coding will be necessary. You'll simply add a GUI object with the text "Hello World." Follow the next few steps to do this:

<Step 1> Run the Unity Version that you just installed, click on New. Use the name HelloUnity. Choose the 2D Template, select Location, Create Project. Wait about a minute or two.

<Step 2> GameObject – UI – Text. Set Pos X and Pos Y to 0 in Inspector Panel.

<Step 3> Play the game by clicking on the Play arrow, middle top of the window.

You should see a blue screen with "New Text" in the middle. This "game" is just a static screen.

<Step 4> Stop playing the game by clicking the Play arrow again, and then change the text from "New Text" to "Hello World!" in Inspector panel.

<Step 5> Play and then stop.

<Step 6> File – Save. File – Exit. (On a Mac it's Unity – Quit.)

Special instructions for Mac users may not be there in the remainder of the book. Please read the previous section for Mac users if you haven't done so already.

In the next chapter, you'll learn more about how to code in the Unity environment using C#.

CHAPTER 2

Programming C# in Unity

This chapter covers the basics of programming C# in Unity. In each section, you'll create a small Unity project. Those projects explore some C# programming features as well as how to use them in Unity. This book focuses on those portions of C# necessary for coding games in Unity. C# as a whole is a much larger language. You'll be just fine learning a smaller subset and sticking with it as you work through the rest of this book. In the more distant future, when you become more experienced as a game developer, you may wish to learn more advanced C# techniques.

THE DEFAULT C# SCRIPT IN UNITY

<Step 1> Create a 2D project in Unity and use "DataTypes" as the name.

You can use the Unity Hub to do this. In the hub you select which version of Unity you'll be using. It is recommended that you use 2019.3.0f6 so that you're using the same version as the one used during the creation of this book. Using a more recent version will probably work, but there may be minor differences. Creating a new Unity project may take some time, possibly a minute or two depending on your system.

<Step 2> Edit – Preferences… – External Tools – External Script Editor. (Unity – Preferences on Mac.)

This step may be unnecessary for you. If necessary, select Visual Studio 2019 (Community). As mentioned earlier, this book assumes that you're using Visual Studio as your code editor.

<Step 3> Layout – Revert Factory Settings…

This dropdown menu is at the top right corner of the Unity Editor window. Your screen should look like Figure 2.1.

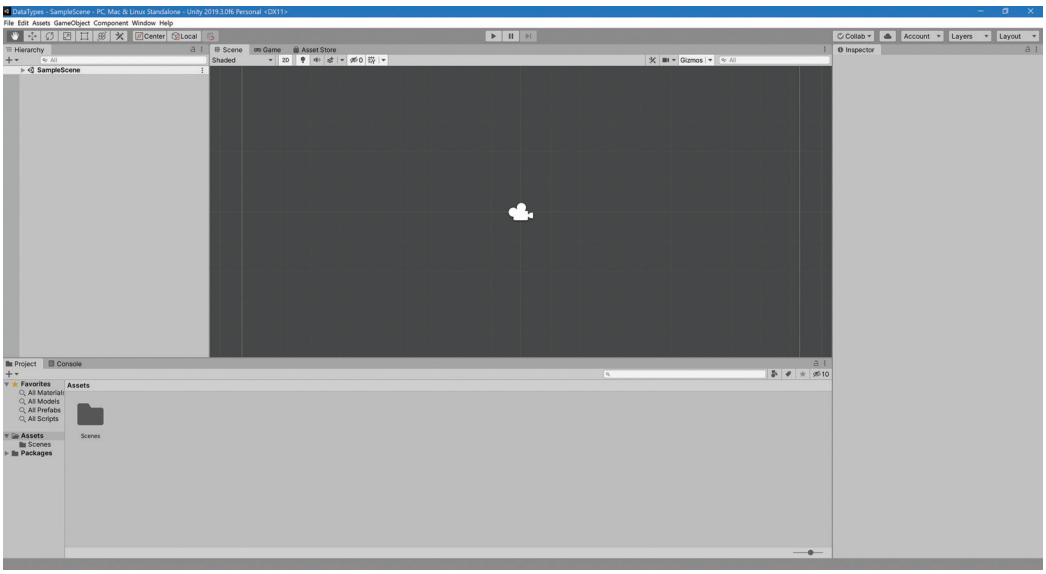

FIGURE 2.1 Unity Screen Default Layout.

<Step 4a> In the Project panel, click on the + icon below the Project tab. Then click on C# Script. Type DataTypesTest and <Enter> (<Return> on Mac) to name it.

<Step 4b> Expand the SampleScene in the Hierarchy by clicking on the small triangle to the left of the bold text.

<Step 4c> Drag the newly created script from the Assets panel on top of the Main Camera in the Hierarchy panel.

<Step 5a> Left-click on DataTypesTest in the Assets panel to select it.

Your screen should look like Figure 2.2.

In the Inspector panel on the right, in the Assembly Information section, you see the current contents of the script. Your next step allows you to edit this script in Visual Studio.

<Step 5b> Double-click on DataTypesTest in the Assets panel.

You should see Visual Studio 2019 open as a separate window. If you're doing this for the first time, this might take a minute or so. If you have a multi-monitor setup, this would be a good time to move the Visual Studio window to another monitor. In Windows, the Visual Studio window should look similar to Figure 2.3.

The Mac display is somewhat different, but it works similarly.

For the rest of this book, code won't be displayed as a graphic figure, but as text:

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

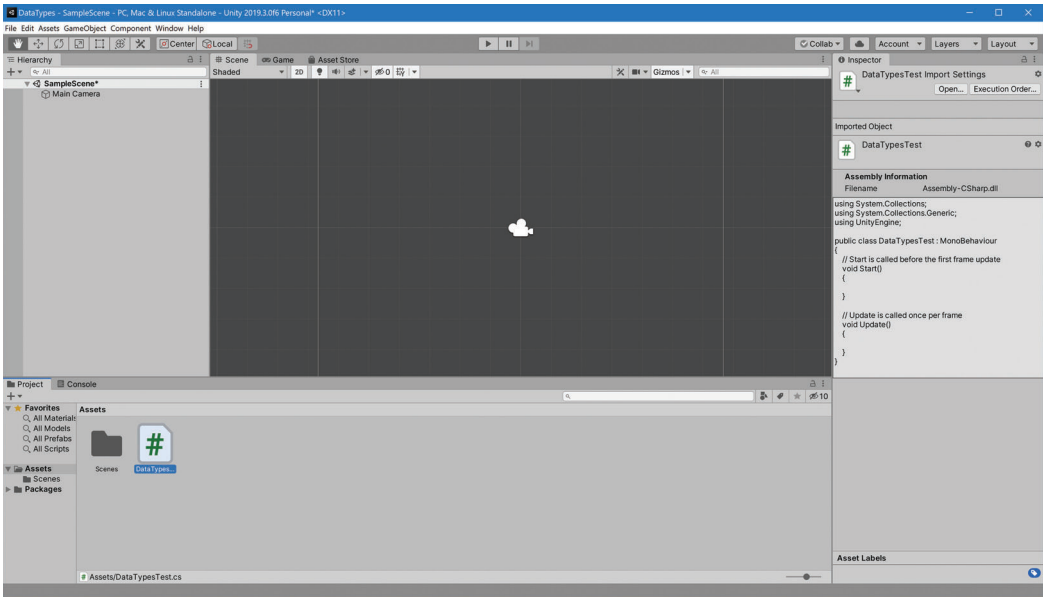

FIGURE 2.2 Creating a script in Unity.

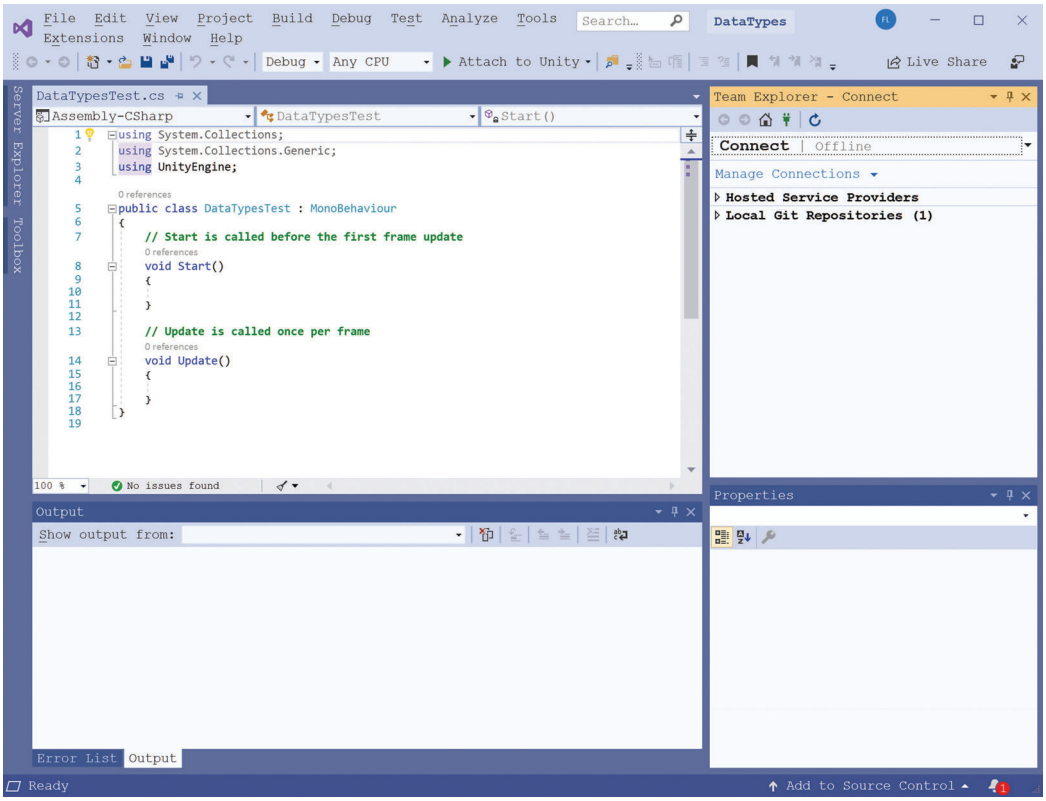

FIGURE 2.3 Visual Studio showing the DataTypesTest script.

```
public class DataTypesTest : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

The colors of the text may or may not match your screen. Using different colors for different types of code entities is called `Color-Coding`. Color-coding the C# code in Visual Studio is a useful automatic feature to help programmers make sense of it all and to help prevent bugs. Before you move on, you'll examine this script line by line.

At the beginning are three lines that start with `using`. These lines are examples of *using directives*. These directives give you access to objects in `System.Collections`, `System.Collections.Generic`, and `UnityEngine`. You can safely ignore these directives for now, but you do need to have them at the top of all of your Unity C# files.

Before you continue your exploration of this file, you need to review your knowledge of animations and frames. Unity is designed to create animations on the various supported target devices. Animations are a sequence of *frames*. Each frame is a graphic image designed to be displayed quickly and in sequence. Movies usually run at 24 frames per second, television broadcasts typically at 25 or 30 frames per second. Video game frame rates vary. They look good at 60 frames per second, though 30 or a bit less is considered acceptable depending on the type of game. Unity renders frames at a target framerate. The default target framerate depends on the device. You can control this in code if you wish.

It's time to return to examining the default file. After the three "using" directives, you see the class definition `DataTypesTest` with two methods: `Start` and `Update`. The Unity engine renders some number of frames each second. The `Start` method is called some time before the first frame is rendered. The `Update` method is automatically called once per frame. These methods contain no statements, so they don't do anything yet. They are there to speed up your editing later on when you insert code into them. If you know you'll never need them, it's safe to delete them.

A note about the terminology for *methods* vs. *functions*: In this book, the words "method" and "function" are used interchangeably. The word "function" sometimes designates a method that returns a value. For example, in mathematics, the square root function takes a nonnegative number as input and returns the square root of that number. The official term in C# is "method" but it's OK to call methods "functions" whether they return a value or not. When a method doesn't return a value, it is declared with the keyword `void`.

NUMERIC DATA TYPES

In this section, you'll explore some of the numeric data types in C#. Examples of numeric data types are bytes, 32-bit integers, and floating point numbers. A good overview of *all* of

the data types for C# can be found at *docs.microsoft.com*. The C# section of that website is a valuable reference resource straight from the company that created C#. That documentation is intended for experienced software engineers, so you may want to stay away if you're a beginner.

You'll start by looking at the eight integer data types: `sbyte`, `short`, `int`, `long`, and their unsigned cousins: `byte`, `ushort`, `uint`, and `ulong`. The code in the following steps tests these data types. You will use the Console window to look at the output of your code. You open it by selecting Window – General – Console in the Unity window.

<Step 6> In the Visual Studio window, insert the following code in `Start`, <Ctrl> S to save, open the Console window if necessary, and click play.

```
int i = 17;
Debug.Log($"i={i}");
i = 2+2;
Debug.Log($"i={i}");
```

In the Console window you should see the following output:

```
i=17
i=4
```

You might see additional text in the Console window, including a time stamp, for example, or additional information about each line.

<Step 7> Stop playing the game by clicking on the play arrow again.

It can sometimes be difficult to know whether or not a game is playing. When in doubt, look at the play arrow at the top of the Unity window. The play arrow has a dark background when the game is playing, and a light gray background when it's not.

The Unity Console window may start out as a detached window or as a panel next to the Project panel. In the following steps, you'll try out some of the features of the Console window (or panel).

<Step 8> Clear the Console window by clicking on the "Clear" tab of the Console window.

<Step 9> Deselect Clear on Play in the Console window if necessary.

<Step 10> Run and stop the game several times.

You should get additional output in the Console every time you run the game.

<Step 11> Select Clear on Play and then repeat Step 10.

This time the console clears every time you run the game, and then the Start method outputs the same text. The result is that the console appears to do nothing.

<Step 12> Insert the following lines:

```
string Timenow = System.DateTime.Now.ToString();
Debug.Log(Timenow);
```

When you run the game multiple times, you'll see the output change slightly where it displays the seconds in the Timenow string.

You're ready to get back to exploring numeric data types.

<Step 13> Replace the `Start` method with the code below, and then test it.

```
void Start()
{
    sbyte b_int;
    short s_int;
    int i_int;
    long l_int;

    b_int = 17;
    s_int = 17;
    i_int = 17;
    l_int = 17;

    Debug.Log($"b_int={b_int}");
    Debug.Log($"s_int={s_int}");
    Debug.Log($"i_int={i_int}");
    Debug.Log($"l_int={l_int}");
}
```

There's not much to this code yet. Your next goal is to break it in various ways. This is the best way to experience how your development environment deals with errors. You'll also gain a better understanding of these data types along the way, which is, after all, your real goal.

<Step 14> Replace the first 17 with 1000.

You'll immediately get an error in Visual Studio: `Constant value '1000' cannot be converted to a 'byte'`. That's because the range of valid numbers for unsigned bytes is `0 to 255`. Below is a table which shows the valid number ranges for the eight integral types in C#:

Data Type	Bytes	Lower Limit	Upper Limit
Byte	1	0	255
Sbyte	1	−128	127
Ushort	2	0	65,535
Short	2	−32,768	32,767
Uint	4	0	4,293,967,295
Int	4	−2,127,483,648	2,147,483,647
Ulong	8	0	18,446,744,073,709,551,615
Long	8	−9,223,372,036,854,775,808	9,223,372,036,854,775,807

It's helpful to understand this table. This will allow you to make an informed decision when selecting which of the integral data types to use in your projects.

<Step 15> Replace `1000` with `128`, then with `-128`. Test this.

`128` is too high for sbyte, `-128` is OK. Notice that you get the error directly in the Visual Studio Editor before even running the game. You can test the other data types in a similar manner if you wish.

<Step 16> Insert the following lines and test them:

```
i_int = 2000000000;
i_int = i_int * 2;
```

That big number is 2 billion, i.e. 2 followed by nine zeros. You might expect 4000000000 as the result of that computation, but instead you get –294967296, which is clearly very wrong. This is an example of *overflow*. 32 bits are just not quite enough storage to represent 4 billion. The next step shows how to fix this.

<Step 17> Replace those last two lines with this:

```
l_int = 2000000000;
l_int = l_int * 2;
```

You now should get the expected result of `4000000000` because the variable l_int is declared as a long. C# does have a feature that enables overflow checking. With that feature enabled, you will get an error when an overflow occurs. This feature is generally not used by programmers because it's a little bit slower, but it can be useful when searching for bugs. Search for "overflow detection C#" in the C# language specification for more details.

As a general rule, using the int data type is a reasonable choice for representing integers. It's unusual to have to count anything that reaches more than 2 billion. The int data type is probably faster than long and it uses half the storage. The short and byte data types are rarely used except when storing large arrays where the storage savings can be significant. The unsigned integral data types are often used when storing bit patterns.

Next, you're now going to look at floating point numbers. C# has two data types for this, float and double. Here is the data type table for floating point numbers in C#:

Data Type	Number of Bytes	Approximate Range	Precision
Float	4	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	6–9 digits
Double	8	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15–17 digits

<Step 18> Insert the following code into the Start method, and then test it:

```
float x;
double y;

x = 2;
y = 3;

Debug.Log($"x = {x}");
Debug.Log($"y = {y}");
```

There's not much going on here. You're displaying a float and a double.

<Step 19> Set x and y to different values as follows, then test.

```
x = 2.1f;
y = 3.1;
```

The x value now needs an f. In C#, when specifying a float like this it's necessary to append the letter f at the end, as you will see in the following step.

<Step 20> Remove the f from 2.1f and see what happens.

You immediately get an error in Visual Studio. You're likely going to see this error in the future because it's easy to forget to type that f when entering code. Unity uses floats for most of its built-in data structures, but the default floating point data type in C# is double.

The next steps do some simple computations with integers and floating point numbers. It's good for you to understand this code before going on to writing actual games.

<Step 21> Insert and test:

```
// Conversion from int to float
Debug.Log($"i_int={i_int}");
x = i_int;
Debug.Log($"x = {x}");

// Conversion from float to int
x = 3.14159f;
i_int = (int)x;
Debug.Log($"i_int = {i_int}");
```

When converting ints to floats and vice versa, you can just do an assignment statement for int to float, but you need to do a *cast* when converting from float to int. A cast looks like this: (data type) and it will explicitly convert the following expression to the data type within the parentheses. The next step shows more casting in action:

<Step 22> Insert and test:

```
x = (float)(3.3 * 5.7);
Debug.Log($"x = {x}");

i_int = (int)(3.3 * 5.7);
Debug.Log($"i_int = {i_int}");

x = (float)3 / (float)(2 + 5);
Debug.Log($"x = {x}");
```

You should get 18.81 for x, 18 for i_int, and then 0.4285714 for x. The first computation takes the double 3.3, multiplies it by the double 5.7, and then converts it to a float. The second computation casts the 18.81 to an int, which has the effect of dropping the fractional part of 18.81. The last computation is simply the floating point version of 3/7.

MATH OPERATORS

In C#, *operators* are sequences of one or more special characters such as + - * / & =. Operators can be combined with variables and literals to form expressions. In this section, you'll explore the basic math operators in C#: addition, subtraction, multiplication, division, modulus, increment, and decrement. A solid knowledge of these operators is essential when developing in C#.

<Step 1> Create a Unity project with the name `Operators`.

<Step 2> Create a script with name `OperatorTest` and drag it onto the `Main Camera` just as you did before in the previous project. Then insert the following code into the `Start` method:

```
int i, j, k, answer;
i = 2;
j = 3;
k = 4;
answer = i + j;
Debug.Log($"answer = {answer}");
```

The first thing to notice is that you're getting a warning in Visual Studio. The warning states that the variable `k` is assigned but never used. This warning can be ignored for now because you intend to use the variable `k` in the very next step. It's a good habit to watch out for warnings and to fix them quickly. It is considered poor programming practice to ignore warnings for any significant length of time.

Your output from Step 2 should be "answer=5". It doesn't get much easier than that. You are now ready to do some more experimentation with the C# arithmetic operators.

<Step 3> Try out the following code:

```
int i, j, k, answer;
i = 2;
j = 3;
k = 4;
answer = i + j + k;
Debug.Log($"answer = {answer}");
answer = i + j*k;
Debug.Log($"answer = {answer}");
answer = (i + j)*k;
Debug.Log($"answer = {answer}");
answer = i + j + k*1000;
Debug.Log($"answer = {answer}");
answer = (i + j + k)*1000;
Debug.Log($"answer = {answer}");
answer = k/3;
Debug.Log($"answer = {answer}");
answer = k%3;
Debug.Log($"answer = {answer}");
answer = (i + j + k)%7;
Debug.Log($"answer = {answer}");
```

You should get the following answers: 9, 14, 20, 4005, 9000, 1, 1, 2. This code tests the four basic arithmetic operators and the modulus operator. Addition, multiplication, and subtraction are straightforward, but division and modulus aren't quite that easy to understand. The result of an integer division is always an integer with any remainder dropped. The remainder can be obtained by using the modulus operator `%`. The last answer in the above code example is the remainder of dividing 9 by 7, which is 2. The modulus operator is quite common and useful, so learning it right now is worthwhile.

The use of parentheses is critical when coding complicated expressions. If you know the *order of operation* of the C# operators you can sometimes avoid parentheses, for example, in the expression `i+j*k`. In that expression, the multiplication is done first, then the addition. That's because multiplication has a higher order of operation than addition. A good rule of thumb is to not rely on your memory of the order of operations, but rather to put in the parentheses. This rule is often broken when it comes to multiplication vs. addition because we are all used to that from algebra. Countless bugs are caused by faulty knowledge of operator precedence, so it's best to put in those parentheses when there's any doubt. Unnecessary parentheses can become necessary when you or someone else modifies your code, so avoid relying on the order of operation rules if there's a good chance that the code will be modified later on.

In the next step you'll test out increment and decrement operators.

<Step 4> Try out the following code:

```
answer = 10;
answer++;
Debug.Log($"answer = {answer}");
answer--;
Debug.Log($"answer = {answer}");
++answer;
Debug.Log($"answer = {answer}");
--answer;
Debug.Log($"answer = {answer}");
```

The answers should be 11, 10, 11, and 10. This code uses the increment and decrement operators both in *prefix* and *postfix* mode. In prefix mode, the operator appears before the expression that it affects, and in postfix mode, it appears afterward. There is another difference between prefix and postfix: The postfix version returns the expression and then operates. The prefix version operates and then returns the expression. The following step tests this.

<Step 5> Try out the following code:

```
answer = 10;
Debug.Log($"answer = {answer++}");
Debug.Log($"answer = {answer}");
answer = 20;
Debug.Log($"answer = {++answer}");
Debug.Log($"answer = {answer}");
```

The answers should be 10, 11, 21, and 21. Notice that in the postfix code the variable `answer` is printed and then incremented. The second output statement prints the incremented variable. In the prefix code, the variable `answer` is incremented first, then printed, thus you get 21 right away.

The next step is really easy, and it's only in here for completeness. The increment and decrement operators are examples of *unary* operators. Two additional unary operators are the `plus` and the `minus` operator.

<Step 6> Try the following code:

```
answer = 10;
Debug.Log($"answer = {+answer}");
Debug.Log($"answer = {-answer}");
```

You will get 10 and –10 as the answers. The strange thing about the plus operator is that it doesn't do anything. It's included in C# for consistency and completeness. It complements the `minus` operator, which is used quite frequently. Both the plus and minus operators are prefix only.

So far, you've only used the `int` data type for testing out these operators. Amazingly, they also work for all of other numeric types, though there are some subtle differences.

<Step 7> Try this:

```
Debug.Log("Floating Test");
float x, y;
double z;
x = 10.1f;
z = 50.123451234512345123;
y = x + 10;
y = x + (float)z;
Debug.Log($"x y z = {x} {y} {z}");
```

The output is **`x y z = 10.1 60.22345 50.1234512345123`**. What is going on here? The assignment for x needs the f at the end of the literal **`10.1f`**. If you forget the f, you get an error. Oddly enough the similar statement **`x = 10`**; does not cause an error because C# can implicitly convert integer literals to floats. That assignment for z has too many digits but C# quietly truncates the extra ones, no error, no warning. This also happens for floats by the way. The first addition statement implicitly converts the 10 to a float and then performs the addition. The second addition statement needs to have a cast for z to convert it to a float. If you remove the cast, there you get an error. The output for y only has seven digits as that is the limit for floats.

The next step tests out compound assignment statements. For a binary operator op, a compound assignment statement looks like this: x op= y, which is the same as x=x op y.

<Step 8> Insert the following code and test it:

```
int i = 2;
i += 7;
Debug.Log($"i = {i}");
```

The value of `i` will be `9`. Compound assignment is a very commonly used feature of C#. It makes your code more readable, and in some cases more efficient, so use it!

BITWISE OPERATORS

In this section, you'll explore the bitwise operators including AND, OR, XOR, Complement, and the Shifts. These operators correspond to very fast hardware instructions on your

device. These operators should be in every game programmer's arsenal. In this section, you will continue to use the Operators project in Unity.

<Step 1> Create a script with name `BitOperatorTest` and drag it onto the `Main Camera` just as you did in the previous section. `Main Camera` now has two active scripts.

<Step 2> Select the `Main Camera` object and disable the `OperatorTest` script in the Inspector. You do this by clicking on the check box next to the name.

<Step 3> Use Visual Studio to insert the following code into the `Start` method of `BitOperatorTest`.

```
int a, b, c;
a = 32;
b = a >> 3;
c = a << 3;
Debug.Log($"a,b,c = {a} {b} {c}");
```

This code tests out the two shift operators: right shift >> and left shift <<. The right shift operator shifts the bit pattern of the variable `a` to the right by three positions, which has the effect of dividing it by 8. Left shift moves the bits to the left, effectively multiplying the variable `a` by 8. The output of this code should be

```
a,b,c = 32,4,256
```

The next step tests out the logical bitwise operators: & | ~ !.

<Step 4> Insert the following code:

```
uint d, e, f;

a = 0x04700_8999;
b = 0x0ffff_0000;

c = a & b;  // AND
d = a | b;  // OR
e = a ^ b;  // XOR
f = ~a;     // Complement


Debug.Log($"a = {a:x}");
Debug.Log($"b = {b:x}");
Debug.Log($"c = {c:x}");
Debug.Log($"d = {d:x}");
Debug.Log($"e = {e:x}");
Debug.Log($"f = {f:x}");
```

This code tests the four bitwise operators AND, OR, XOR, and Complement. Your output should look like this:

```
a,b,c = 32 4 256
a = 47008999
b = ffff0000
c = 47000000
d = ffff8999
e = b8ff8999
f = b8ff7666
```