

O'REILLY®

Fourth  
Edition

Head First

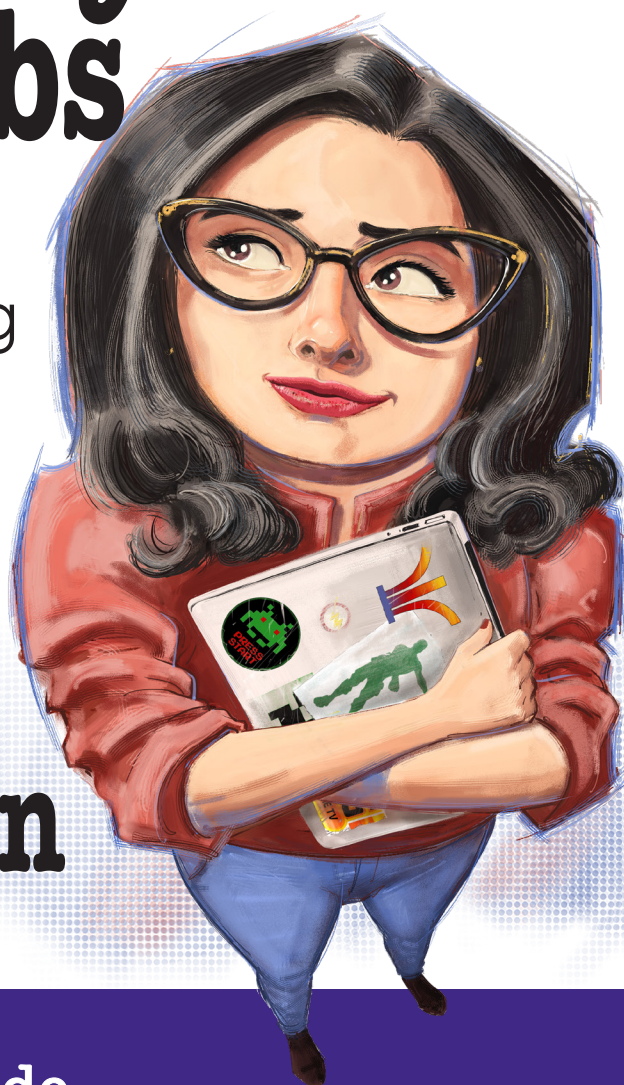
# C# Unity Labs

A Learner's Guide to  
Real-World Programming  
with C# and .NET Core

Andrew Stellman  
& Jennifer Greene

**RD** **Rider Edition**

by JetBrains



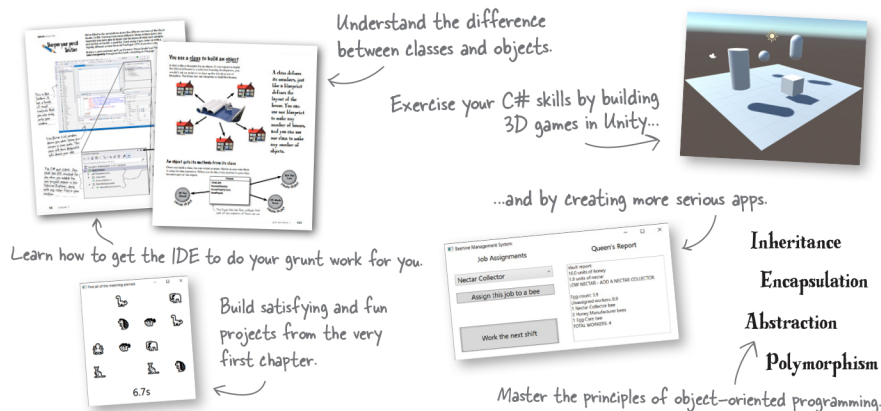
A Brain-Friendly Guide

# Head First

# C#

## What will you learn from this book?

Dive into C# and create apps, user interfaces, games, and more using this fun and highly visual introduction to C#, .NET Core, and Visual Studio. With this completely updated guide, which covers C# 8.0 and Visual Studio 2019, beginning programmers like you will build a fully functional game in the opening chapter. Then you'll learn how to use classes and object-oriented programming, create 3D games in Unity, and query data with LINQ. And you'll do it all by solving puzzles, doing hands-on exercises, and building real-world applications. By the time you're done, you'll be a solid C# programmer—and you'll have a great time along the way!



## What's so special about this book?

Based on the latest research in cognitive science and learning theory, *Head First C#* uses a visually rich format to engage your mind rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multisensory learning experience is designed for the way your brain really works.

"Thank you so much!  
Your books have  
helped me to launch  
my career."

—Ryan White  
Game Developer

"Andrew and Jennifer  
have written a  
concise, authoritative,  
and most of all, fun  
introduction to C#  
development."

—Jon Galloway  
Senior Program Manager on the  
.NET Community Team  
at Microsoft

"If you want to learn  
C# in depth and have  
fun doing it, this is THE  
book for you."

—Andy Parker  
Fledgling C# programmer

.NET

US \$64.99

CAN \$85.99

ISBN: 978-1-491-97670-8



O'REILLY®

# Head First C#

Fourth Edition



WOULDN'T IT BE DREAMY IF  
THERE WAS A C# BOOK THAT WAS  
MORE FUN THAN MEMORIZING  
A DICTIONARY? IT'S PROBABLY  
NOTHING BUT A FANTASY...

Andrew Stellman  
Jennifer Greene

## Welcome to the Head First C# Unity Labs – Rider edition!

Unity is an amazing cross-platform tool for game development and simulation that uses C#. When you combine Unity with C# development, you get a really fun way to learn and experiment with C#, which is perfect for people learning C# or who want to brush up on their skills.

That's why we included Unity Lab projects throughout Head First C#, our popular book on learning C# published by O'Reilly. Our Unity Labs give you a fast and easy way to get up to speed with Unity, whether you're new to C# and looking to practice your skills or an experienced C# developer curious about Unity. You can download all of our Unity Labs for free from the Head First C# GitHub page:

<https://github.com/head-first-csharp/fourth-edition>

This PDF contains the Rider edition of the first four Unity Labs. JetBrains Rider is a fast and powerful cross-platform .NET IDE that features seamless built-in integration with Unity. You can learn more about Rider on the JetBrains website: <https://www.jetbrains.com/rider/>

# Head First C#

## Fourth Edition

by Andrew Stellman and Jennifer Greene

Copyright © 2021 Jennifer Greene, Andrew Stellman. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Series Creators:**

Kathy Sierra, Bert Bates

**Series Advisors:**

Eric Freeman, Elisabeth Robson

**Cover Designer:**

Ellie Volckhausen

**Editors:**

Nicole Taché, Amanda Quinn

**Proofreader:**

Rachel Head

**Indexer:**

Potomac Indexing, LLC

**Illustrator:**

Jose Marzan

**Page Viewers:**

Greta the miniature bull terrier and Samosa the Pomeranian

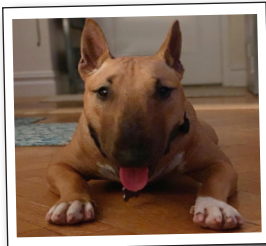
**Printing History:**

November 2007: First Edition

May 2010: Second Edition

August 2013: Third Edition

December 2020: Fourth Edition



The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First C#*, and related trade dress are trademarks of O'Reilly Media, Inc.

Microsoft, Windows, Visual Studio, MSDN, the .NET logo, Visual Basic, and Visual C# are registered trademarks of Microsoft Corporation. JetBrains, Rider, and the Rider logo are registered trademarks of JetBrains.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

No bees, space aliens, or comic book heroes were harmed in the making of this book.

ISBN: 978-1-491-97670-8

[LSI]

[2020-12-18]



# Unity Lab #1

## Explore C# with Unity

Welcome to your first **Head First C# Unity Lab**. Writing code is a skill, and like any other skill, getting better at it takes **practice and experimentation**. Unity will be a really valuable tool for that.

Unity is a cross-platform game development tool that you can use to make professional-quality games, simulations, and more. It's also a fun and satisfying way to get **practice with the C# tools and ideas** you'll learn throughout this book. We designed these short, targeted labs to **reinforce** the concepts and techniques you just learned to help you hone your C# skills.

These labs are optional, but valuable practice—**even if you aren't planning on using C# to build games**.

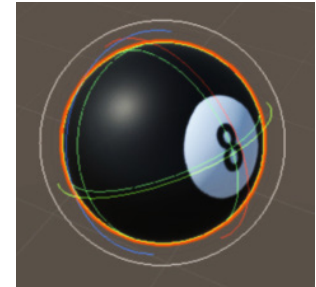
In this first lab, you'll get up and running with Unity. You'll get oriented with the Unity editor, and you'll start creating and manipulating 3D shapes.

# Unity is a powerful tool for game design

Welcome to the world of Unity, a complete system for designing professional-quality games—both two-dimensional (2D) and three-dimensional (3D)—as well as simulations, tools, and projects. Unity includes many powerful things, including..

## A cross-platform game engine

A **game engine** displays the graphics, keeps track of the 2D or 3D characters, detects when they hit each other, makes them act like real-world physical objects, and much, much more. Unity will do all of these things for the 3D games you build throughout this book.

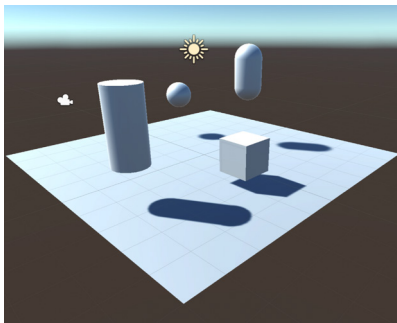


## A powerful 2D and 3D scene editor

You'll be spending a lot of time in the Unity editor. It lets you edit levels full of 2D or 3D objects, with tools that you can use to design complete worlds for your games. Unity games use C# to define their behavior, and the Unity editor integrates with your C# IDE to give you a seamless game development environment.



While these Unity Labs will concentrate on C# development in Unity, if you're a visual artist or designer, the Unity editor has many artist-friendly tools designed just for you. Check them out here: <https://unity3d.com/unity/features/editor/art-and-design>.



## An ecosystem for game creation

Beyond being an enormously powerful tool for creating games, Unity also features an ecosystem to help you build and learn. The Learn Unity page (<https://unity.com/learn>) has valuable self-guided learning resources, and the Unity forums (<https://forum.unity.com>) help you connect with other game designers and ask questions. The Unity Asset Store (<https://assetstore.unity.com>) provides free and paid assets like characters, shapes, and effects that you can use in your Unity projects.

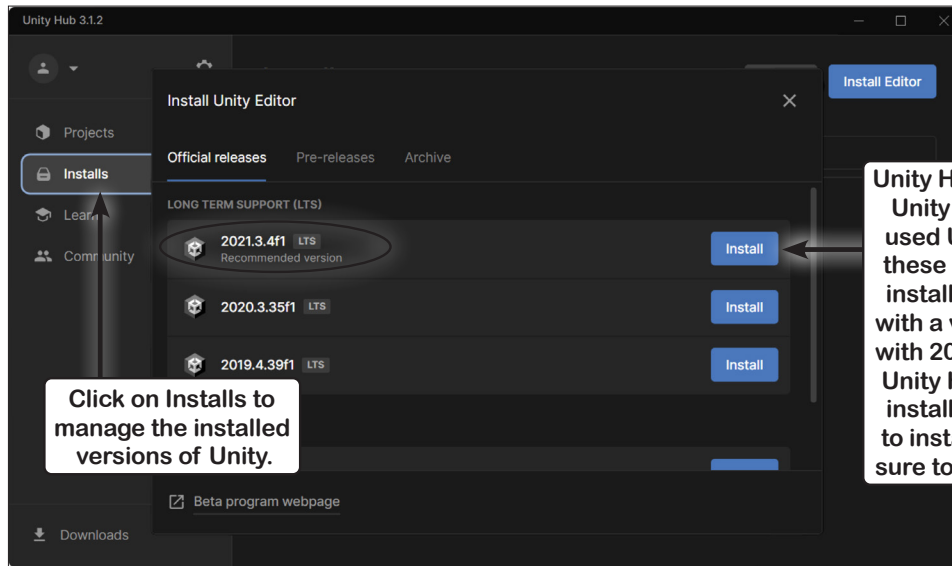
## Our Unity Labs will focus on using Unity as a tool to explore C#, and practicing with the C# tools and ideas that you've learned throughout the book.

The *Head First C#* Unity Labs are laser-focused on a **developer-centric learning path**. The goal of these labs is to help you ramp up on Unity quickly, with the same focus on brain-friendly just-in-time learning you'll see throughout *Head First C#* to **give you lots of targeted, effective practice with C# ideas and techniques**.

## Download Unity Hub

**Unity Hub** is an application that helps you manage your Unity projects and your Unity installations, and it's the starting point for creating your new Unity project. Start by downloading Unity Hub from <https://store.unity.com/download>—then install it and run it.

All of the screenshots in this book were taken with the free Personal Edition of Unity. You'll need to enter your unity.com username and password into Unity Hub to activate your license.



Unity Hub lets you install multiple versions of Unity on the same computer, so you should install the same version that we used to build these labs. **Click Official Releases** and install the latest version that starts with **Unity 2021.3**—that's the same version we used to take the screenshots in these labs. Once it's installed, make sure that it's set as the preferred version.

The Unity installer may prompt you to install Visual Studio—but since this is the Rider edition of the Unity Lab, we'll be using Rider instead. (If you're curious, you can still have multiple installations of Visual Studio on the same computer as Rider, and if you already have one version of Visual Studio installed there's no need to make the Unity installer add another one.)

You can learn more about installing Unity Hub on Windows, macOS, and Linux here: <https://docs.unity3d.com/Manual/GettingStartedInstallingUnity.html>

Unity Hub lets you have many Unity installs on the same computer. So even if there's a newer version of Unity available, you can use Unity Hub to install the version we used in the Unity Labs.



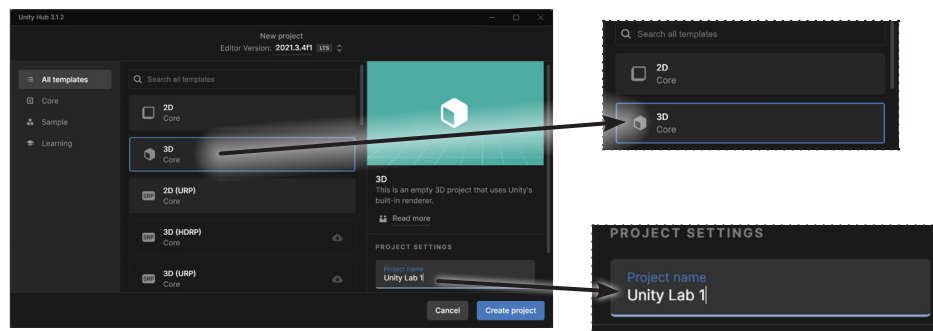
**Watch it!**

### Unity Hub may look a little different.

*The screenshots in this book were taken with Unity 2021.3 (Personal Edition) and Unity Hub 3.1.2. You can use Unity Hub to install many different versions of Unity on the same computer, but you can only install the latest version of Unity Hub. The Unity development team is constantly improving Unity Hub and the Unity editor, so it's possible that what you see won't quite match what's shown on this page. We update these Unity Labs for newer printings of **Head First C#**. We'll add PDFs of updated labs to our GitHub page: <https://github.com/head-first-csharp/fourth-edition>.*

## Use Unity Hub to create a new project

Click the **New project** button on the Project page in Unity Hub to create a new Unity project. Name it **Unity Lab 1**, make sure the 3D template is selected, and check that you're creating it in a sensible location (usually the Unity Projects folder underneath your home directory).



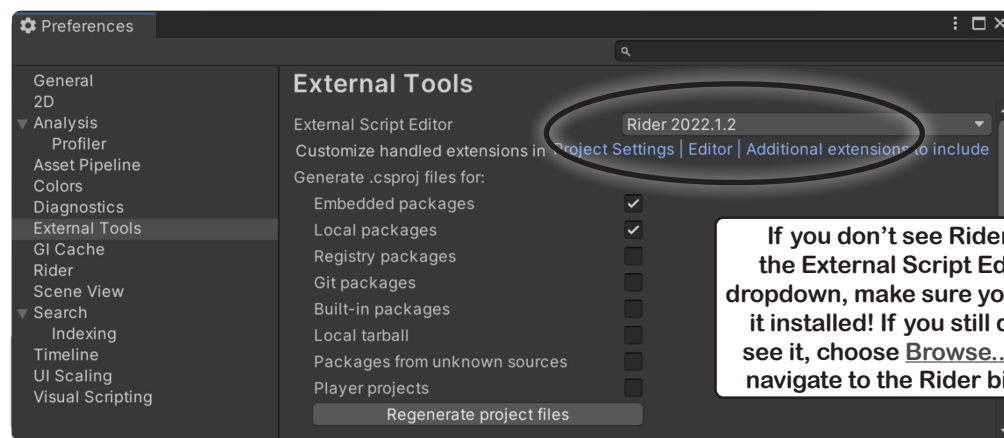
You can use Rider to debug the code in your Unity games. Just choose Rider as the external script editor in Unity's preferences.

Click Create Project to create the new folder with the Unity project. When you create a new project, Unity generates a lot of files (just like your C# IDE does when it creates new projects for you). It could take Unity a minute or two to create all of the files for your new project.

## Make Rider your Unity script editor

The Unity editor works hand-in-hand with the Rider IDE to make it really easy to edit and debug the code for your games. So the first thing we'll do is make sure that Unity is hooked up to Rider. **Choose Preferences from the Edit menu** (or from the Unity menu on a Mac) to open the Unity Preferences window. Click on External Tools on the left, and **choose Rider** from the External Script Editor window.

*In some older versions of Unity, you may see an **Editor Attaching** checkbox—if so, make sure that it's checked (that will let you debug your Unity code in the IDE).*



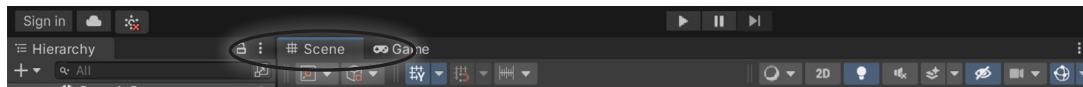
OK! You're all ready to get started building your first Unity project.



## Take control of the Unity layout

The Unity editor is like an IDE for all of the parts of your Unity project that aren't C#. You'll use it to work with scenes, edit 3D shapes, create materials, and so much more. The windows and panels in the Unity editor can be rearranged in many different layouts.

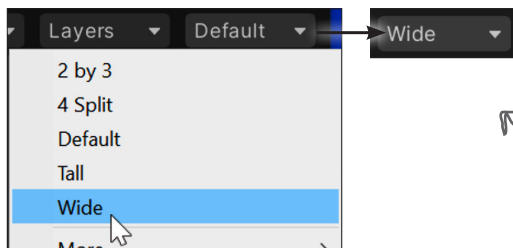
Find the Scene tab near the top of the window. Click on the tab and drag it to detach the window:



Try docking it inside or next to other panels, then drag it to the middle of the editor to make it a floating window.

## Choose the Wide layout to match our screenshots

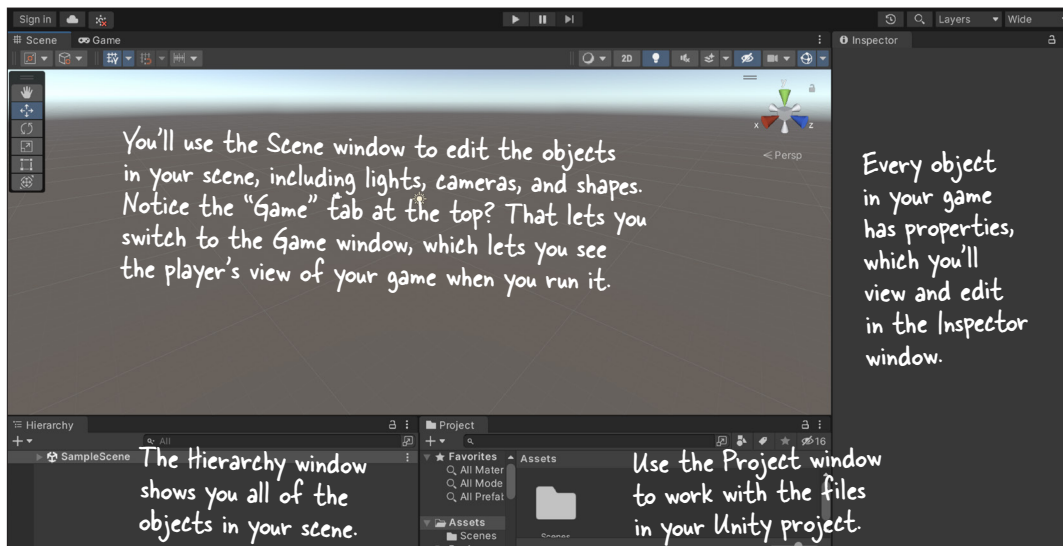
We've chosen the Wide layout because it works well for the screenshots in these labs. Find the Layout dropdown and choose Wide so your Unity editor looks like ours.



Once you change the layout with the Layout dropdown on the right side of the toolbar, the dropdown may change its label to match the layout that you selected.

The Scene view is your main interactive view of the world that you're creating. You use it to position 3D shapes, cameras, lights, and all of the other objects in your game.

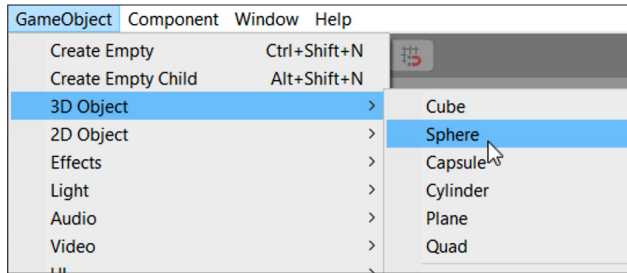
Here's what your Unity editor should look like in the Wide layout:



### Your scene is a 3D environment

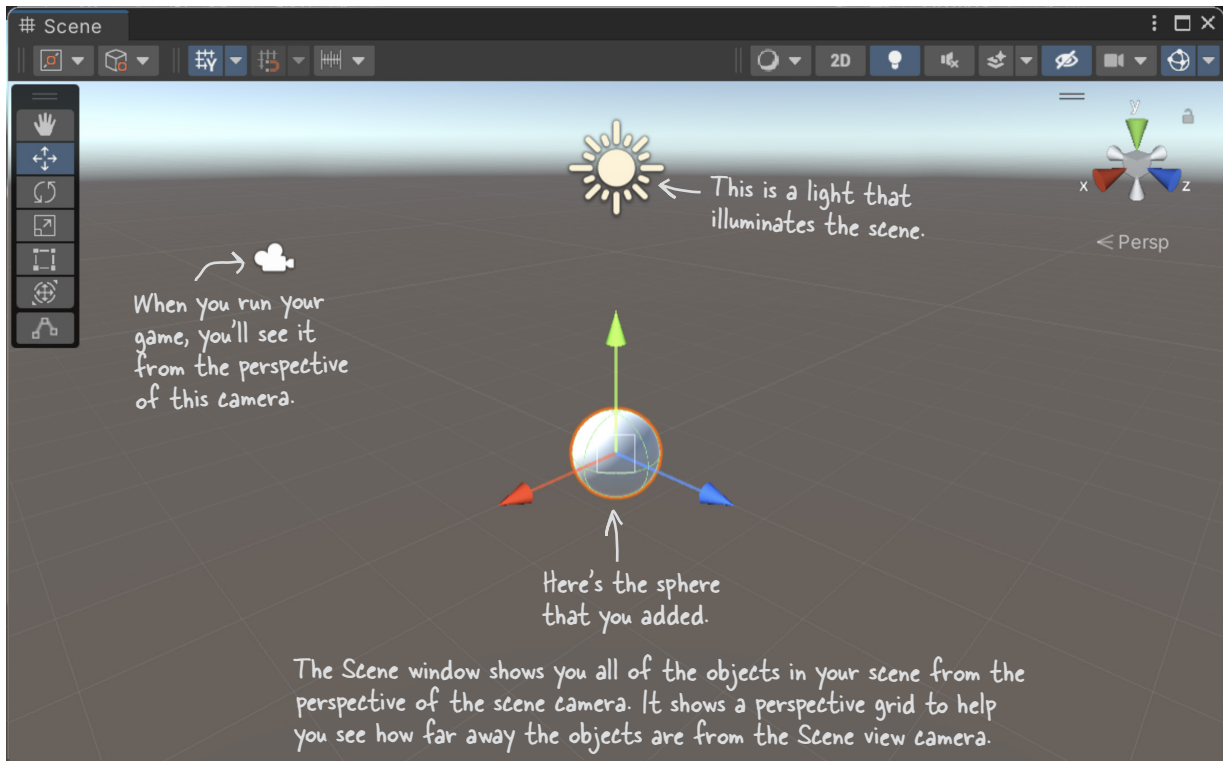
As soon as you start the editor, you're editing a **scene**. You can think of scenes as levels in your Unity games. Every game in Unity is made up of one or more scenes. Each scene contains a separate 3D environment, with its own set of lights, shapes, and other 3D objects. When you created your project, Unity added a scene called *SampleScene*, and stored it in a file called *SampleScene.unity*.

Add a sphere to your scene by choosing **GameObject >> 3D Object >> Sphere** from the menu:



These are called Unity's "primitive objects." We'll be using them a lot throughout these Unity Labs.

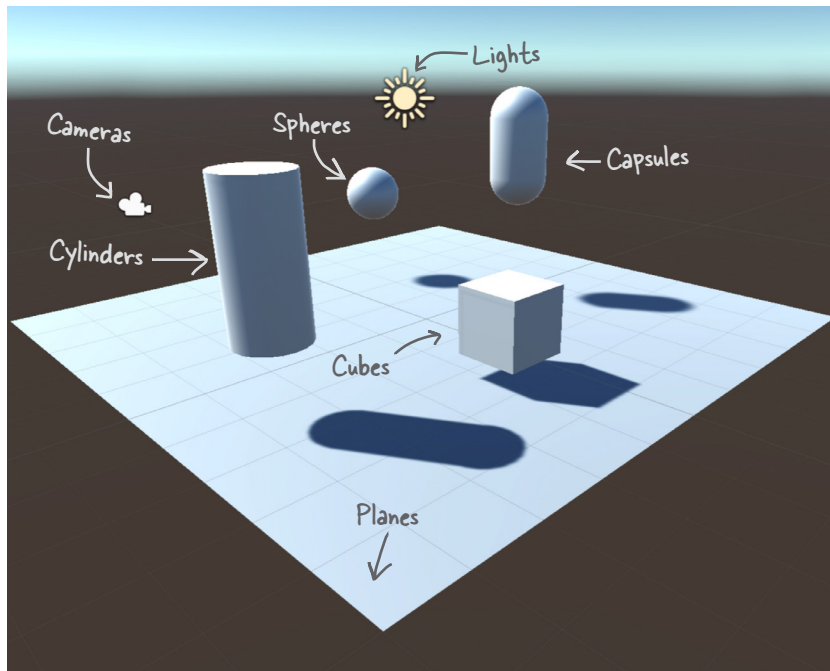
A sphere will appear in your Scene window. Everything you see in the Scene window is shown from the perspective of the **Scene view camera**, which "looks" at the scene and captures what it sees.



### Unity games are made with GameObjects

When you added a sphere to your scene, you created a new **GameObject**. The **GameObject** is a fundamental concept in Unity. Every item, shape, character, light, camera, and special effect in your Unity game is a **GameObject**. Any scenery, characters, and props that you use in a game are represented by **GameObjects**.

In these Unity Labs, you'll build games out different kinds of **GameObjects**, including:



Each **GameObject** contains a number of **components** that provide its shape, set its position, and give it all of its behavior. For example:

- ★ *Transform components* determine the position and rotation of the **GameObject**.
- ★ *Material components* change the way the **GameObject** is **rendered**—or how it's drawn by Unity—by changing the color, reflection, smoothness, and more.
- ★ *Script components* use C# scripts to determine the **GameObject**'s behavior.

ren-der, verb.

to represent or depict artistically.

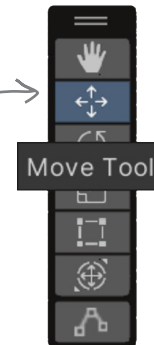
*Michelangelo **rendered** his favorite model with more detail than he used in any of his other drawings.*

GameObjects are the fundamental objects in Unity, and components are the basic building blocks of their behavior. The Inspector window shows you details about each **GameObject** in your scene and its components.

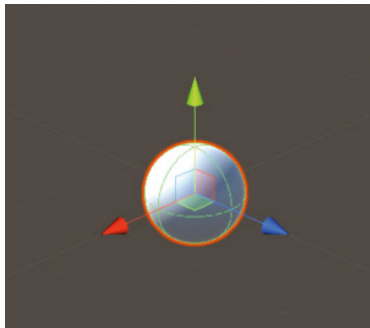
## Use the Move Gizmo to move your GameObjects

The toolbar at the top of the Unity editor lets you choose Transform tools. If the Move tool isn't selected, press its button to select it.

The buttons on in the floating toolbar let you choose Transform Tools like the Move tool, which displays the Move Gizmo as arrows and a cube on top of the GameObject that's currently selected.

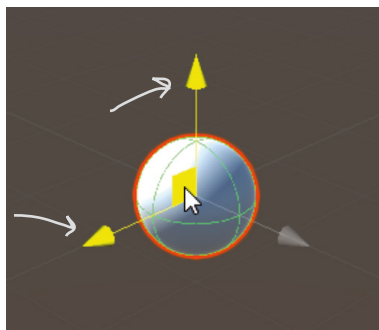


The Move tool lets you use the **Move Gizmo** to move GameObjects around the 3D space. You should see red, green, and blue arrows and a cube appear in the middle of the window. This is the Move Gizmo, which you can use to move the selected object around the scene.



Move your mouse cursor over the cube at the center of the Move Gizmo—notice how each of the faces of the cube lights up as you move your mouse cursor over it? Click on the upper-left face and drag the sphere around. You're moving the sphere in the X-Y plane.

When you click on the upper-left face of the cube in the middle of the Move Gizmo, its X and Y arrows light up and you can drag your sphere around the X-Y plane in your scene.



The Move Gizmo lets you move GameObjects along any axis or plane of the 3D space in your scene.

**Move your sphere around the scene** to get a feel for how the Move Gizmo works. Click and drag each of the three arrows to drag it along each plane individually. Try clicking on each of the faces of the cube in the Scene Gizmo to drag it around all three planes. Notice how the sphere gets smaller as it moves farther away from you—or really, the scene camera—and larger as it gets closer.

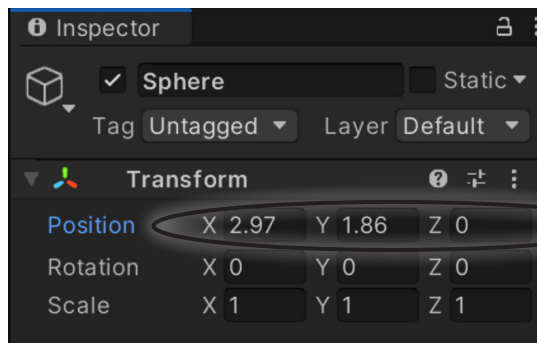


## The Inspector shows your GameObject's components

As you move your sphere around the 3D space, watch the **Inspector window**, which is on the right side of the Unity editor if you're using the Wide layout. Look through the Inspector window—you'll see that your sphere has four components labeled Transform, Sphere (Mesh Filter), Mesh Renderer, and Sphere Collider.

Every GameObject has a set of components that provide the basic building blocks of its behavior, and every GameObject has a **Transform component** that drives its location, rotation, and scale.

You can see the Transform component in action as you use the Move Gizmo to drag the sphere around the X-Y plane. Watch the X and Y numbers in the Position row of the Transform component change as the sphere moves.



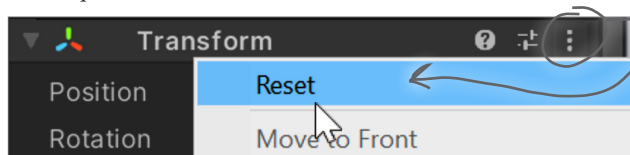
If you accidentally deselect a GameObject, just click on it again. If it's not visible in the scene, you can select it in the **Hierarchy window**, which shows all of the GameObjects in the scene. When you reset the layout to Wide, the Hierarchy window is in the lower-left corner of the Unity editor.

Did you notice the grid in your 3D space? As you're dragging the sphere around, hold down the Control key. That causes the GameObject that you're moving to snap to the grid. You'll see the numbers in the Transform component move by whole numbers instead of small decimal increments.

Try clicking on each of the other two faces of the Move Gizmo cube and dragging to move the sphere in the X-Z and Y-Z planes. Then click on the red, green, and blue arrows and drag the sphere along just the X, Y, or Z axis. You'll see the X, Y, and Z values in the Transform component change as you move the sphere.

Now **hold down Shift** to turn the cube in the middle of the Gizmo into a square. Click and drag on that square to move the sphere in the plane that's parallel to the Scene view camera.

Once you're done experimenting with the Move Gizmo, use the sphere's Transform component context menu to reset the component to its default values. Click the **context menu button** (⋮) at the top of the Transform panel and choose Reset from the menu.



Use the context menu to reset a component. You can either click the three dots or right-click anywhere in the top line of the Transform panel in the Inspector window to bring up the context menu.

The position will reset back to [0, 0, 0].

You can learn more about the tools and how to use them to position GameObjects in the Unity Manual. Click Help >> Unity Manual and search for the "Position GameObjects" page.

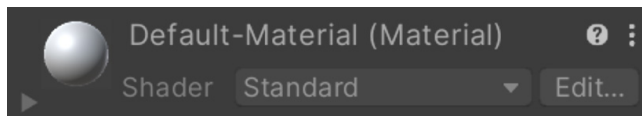
**Save your scene often! Use File >> Save or Ctrl+S / ⌘S to save the scene right now.**

## Add a material to your Sphere GameObject

Unity uses **materials** to provide color, patterns, textures, and other visual effects. Your sphere looks pretty boring right now because it just has the default material, which causes the 3D object to be rendered in a plain, off-white color. Let's make it look like a billiard ball.

### ① Select the sphere.

When the sphere is selected, you can see its material as a component in the Inspector window:



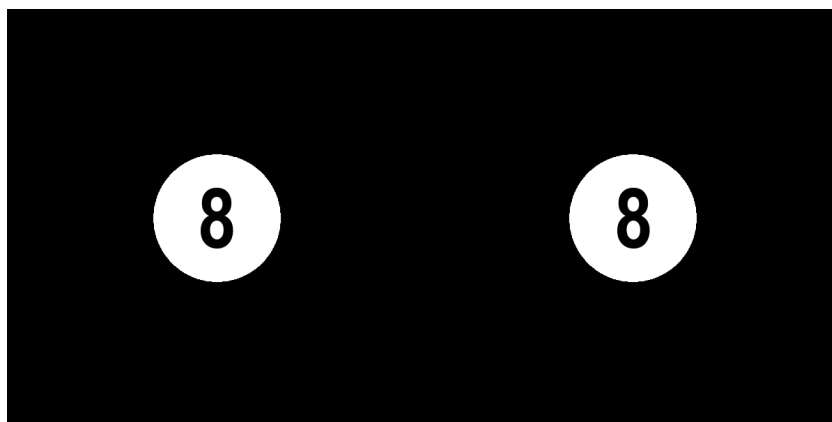
We'll make your sphere more interesting by adding a **texture**—that's just a simple image file that's wrapped around a 3D shape, almost like you printed the picture on a rubber sheet and stretched it around your object.

### ② Go to our Billiard Ball Textures page on GitHub.

Go to <https://github.com/head-first-csharp/fourth-edition> and click on the *Billiard Ball Textures* link to browse a folder of texture files for a complete set of billiard balls.

### ③ Download the texture for the 8 ball.

Click on the file *8 Ball Texture.png* to view the texture for an 8 ball. It's an ordinary 1200 × 600 PNG image file that you can open in your favorite image viewer.



← We designed this image file so that it looks like an 8 ball when Unity “wraps” it around a sphere.

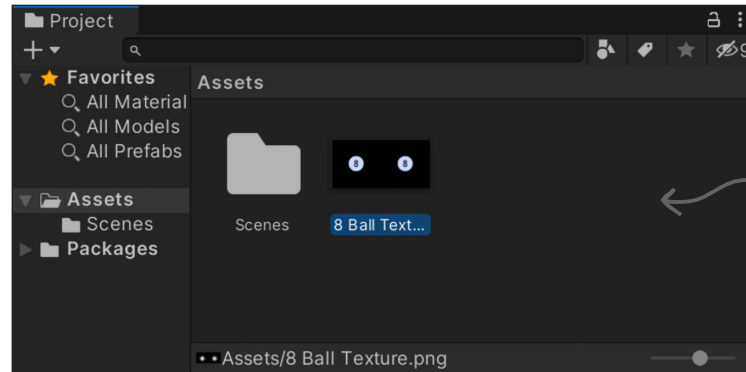
Download the file into a folder on your computer.

*(You might need to right-click on the Download button to save the file, or click Download to open it and then save it, depending on your browser.)*

④

### Import the 8 Ball Texture image into your Unity project.

Right-click on the Assets folder in the Project window, choose **Import New Asset...** and import the texture file. You should now see it when you click on the Assets folder in the Project window.

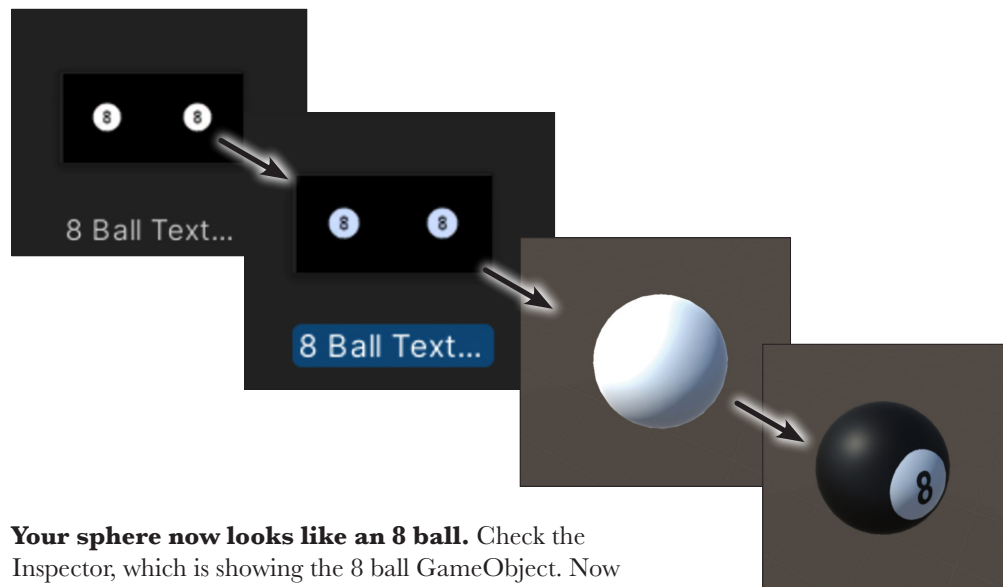


You right-clicked inside the Assets folder in the Project window to import the new asset, so Unity imported the texture into that folder.

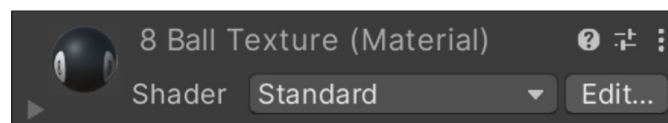
⑤

### Add the texture to your sphere.

Now you just need to take that texture and “wrap” it around your sphere. Click on 8 Ball Texture in the Project window to select it. Once it's selected, **drag it onto your sphere**.



**Your sphere now looks like an 8 ball.** Check the Inspector, which is showing the 8 ball GameObject. Now it has a new material component:





I'M LEARNING C# FOR MY JOB,  
NOT TO WRITE VIDEO GAMES. WHY  
SHOULD I CARE ABOUT UNITY?

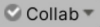
### Unity is a great way to really “get” C#.

Programming is a skill, and the more practice you get writing C# code, the better your coding skills will get. That's why we designed the Unity Labs throughout this book to specifically **help you practice your C# skills** and reinforce the C# tools and concepts that you learn in each chapter. As you write more C# code, you'll get better at it, and that's a really effective way to become a great C# developer. Neuroscience tells us that we learn more effectively when we experiment, so we designed these Unity Labs with lots of options for experimentation, and suggestions for how you can get creative and keep going with each lab.

But Unity gives us an even more important opportunity to help get important C# concepts and techniques into your brain. When you're learning a new programming language, it's really helpful to see how that language works with lots of different platforms and technologies. That's why we included both console apps and WPF apps in the main chapter material, and in some cases even have you build the same project using both technologies. Adding Unity to the mix gives you a third perspective, which can really accelerate your understanding of C#.

The **GitHub for Unity extension** (<https://unity.github.com>) lets you save your Unity projects in GitHub. Here's how:

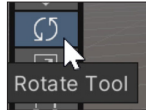
- **To install GitHub for Unity:** Go to <https://assetstore.unity.com> and add GitHub for Unity to your assets. Go back to Unity, **choose Package Manager** from the Window menu, select “GitHub for Unity” from “My Assets,” and import it. You'll need to import GitHub into each new Unity project.
- **To push your changes to a GitHub repo:** Choose GitHub from the Window menu. Each Unity project is stored in a separate repository in your GitHub account, so **click the Initialize button** to initialize a new *local* repo (you'll be prompted to log into GitHub), then **click the Publish button** to create a new repo in your GitHub account for your project. Any time you want to push your changes to GitHub, **go to the Changes tab** in the GitHub window, **click All**, enter a **commit summary** (any text will do), and **click Commit** at the bottom of the GitHub window. Then click **Push (1)** at the top of the GitHub window to push your changes back to GitHub.

You can also back up and share your Unity projects with **Unity Collaborate**, which lets you publish your projects to their cloud storage. Your Unity Personal account comes with 1 GB of cloud storage for free, which is enough for all of the Unity Lab projects in this book. Unity will even keep track of your project history (which doesn't count against your storage limit). To publish your project, click the **Collab** () **button** on the toolbar, then click Publish. Use the same button to publish any updates. To see your published projects, log into <https://unity3d.com> and use the account icon to view your account, then click the Projects link from your account overview page to see your projects.

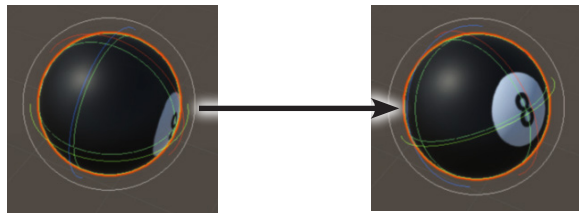


### Rotate your sphere

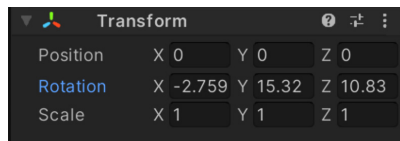
Click the **Rotate tool** in the toolbar. You can use the Q, W, E, R, T, and Y keys to quickly switch between the Transform tools—press E and W to toggle between the Rotate tool and Move tool.



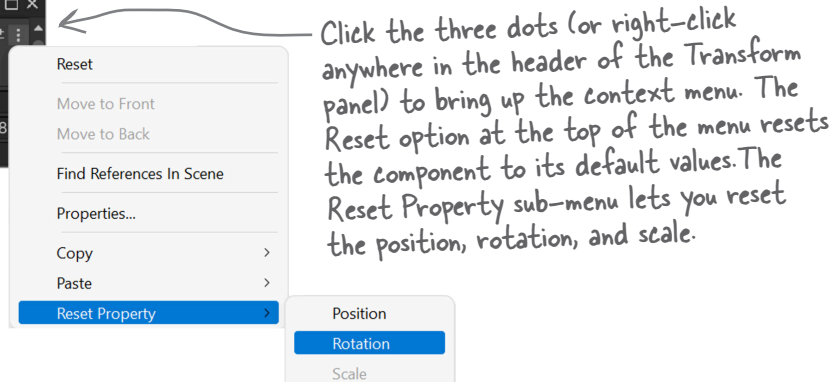
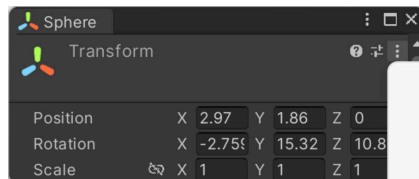
- 1 **Click on the sphere.** Unity will display a wireframe sphere Rotate Gizmo with red, blue, and green circles. Click the red circle and drag it to rotate the sphere around the X axis.



- 2 **Click and drag the green and blue circles to rotate around the Y and Z axes.** The outer white circle rotates the sphere along the axis coming out of the Scene view camera. Watch the Rotation numbers change in the Inspector window.



- 3 **Open the context menu of the Transform panel in the Inspector window.** Click Reset, just like you did before. It will reset everything in the Transform component back to default values—in this case, it will change your sphere's rotation back to [0, 0, 0].



**It's easy to reset your windows and scene camera.**

If you change your Scene view so you can't see your sphere anymore, or if you drag your windows out of position, just use the layout dropdown in the upper-right corner to **reset the Unity editor to the Wide layout**. It will reset the window layout and move the Scene view camera back to its default position.

**Use File >> Save or Ctrl+S / ⌘S to save the scene right now. Save early, save often!**

### Move the Scene view camera with the View tool and Scene Gizmo

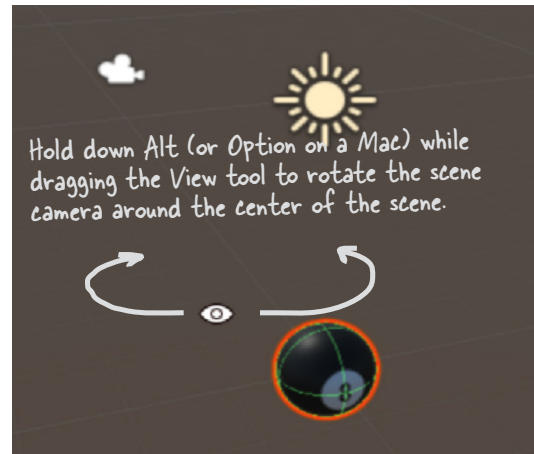
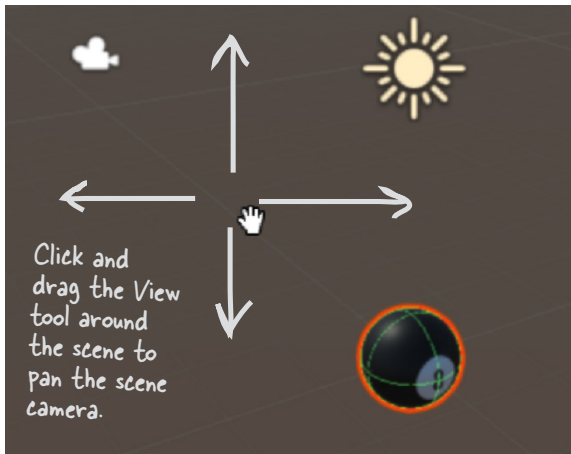
Use the mouse scroll wheel or scroll feature on your trackpad to zoom in and out, and toggle between the Move and Rotate Gizmos. Notice that the sphere changes size, but the Gizmos don't. The Scene window in the editor shows you the view from a virtual **camera**, and the scroll feature zooms that camera in and out.

Press **Q** to select the **View Tool** or choose it from the toolbar. Your cursor will change to a hand.



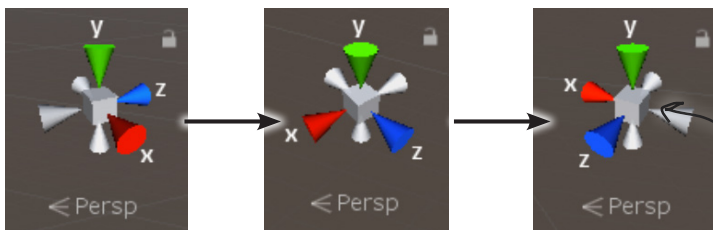
Hold down **ALT** (or **Option** on a Mac) while dragging and the View tool turns into an eye and rotates the view around the center of the window

The View tool—sometimes called the “Hand tool”—pans around the scene by changing the position and rotation of the scene camera. When the View tool is selected, you can click anywhere in the scene to pan.



When the View tool is selected, you can **pan** the scene camera by **clicking and dragging**, and you can **rotate** it by holding **down ALT (or Option) and dragging**. Use the **mouse scroll wheel** to zoom. Holding down the **right mouse button** lets you **fly through the scene** using the W-A-S-D keys.

When you rotate the scene camera, keep an eye on the **Scene Gizmo** in the upper-right corner of the Scene window. The Scene Gizmo always displays the camera's orientation—check it out as you use the View tool to move the Scene view camera. Click on the X, Y, and Z cones to snap the camera to an axis.



The Unity Manual has great tips on navigating scenes: <https://docs.unity3d.com/Manual/SceneViewNavigation.html>.

### *there are no* Dumb Questions

**Q:** I'm still not clear on exactly what a component is. What does it do, and how is it different from a GameObject?

**A:** A GameObject doesn't actually do much on its own. All a GameObject really does is serve as a *container* for components. When you used the GameObject menu to add a Sphere to your scene, Unity created a new GameObject and added all of the components that make up a sphere, including a Transform component to give it position, rotation, and scale, a default Material to give it its plain white color, and a few other components to give it its shape, and help your game figure out when it bumps into other objects. These components are what make it a sphere.

**Q:** So does that mean I can just add any component to a GameObject and it gets that behavior?

**A:** Yes, exactly. When Unity created your scene, it added two GameObjects, one called Main Camera and another called Directional Light. If you click on Main Camera in the Hierarchy window, you'll see that it has three components: a Transform, a Camera, and an Audio Listener. If you think about it, that's all a camera actually needs to do: be somewhere, and pick up visuals and audio. The Directional Light GameObject just has two components: a Transform and a Light, which casts light on other GameObjects in the scene.

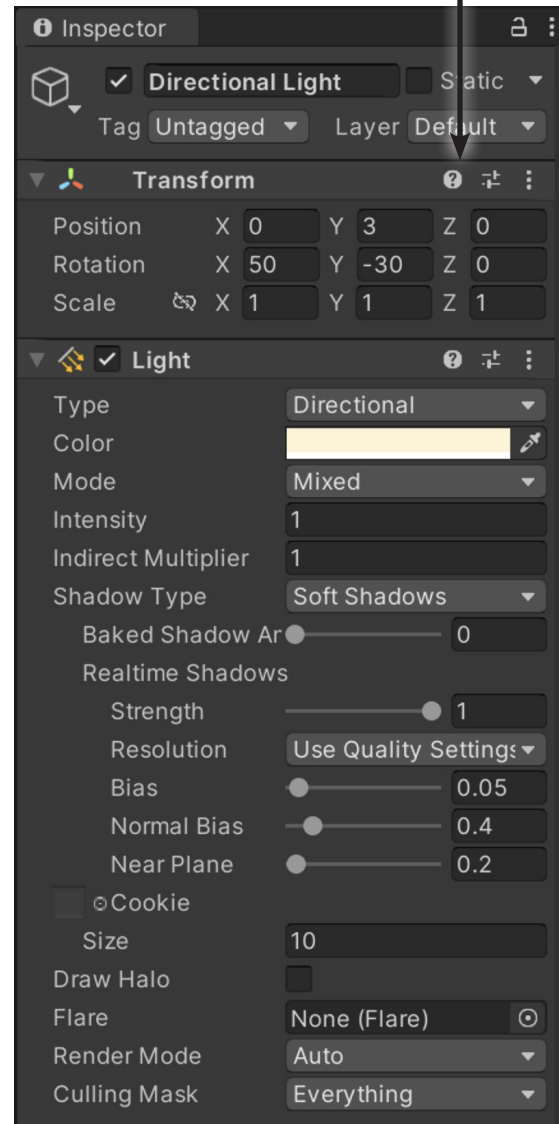
**Q:** If I add a Light component to any GameObject, does it become a light?

**A:** Yes! A light is just a GameObject with a Light component. If you click on the Add Component button at the bottom of the Inspector and add a Light component to your ball, it will start emitting light. If you add another GameObject to the scene, it will reflect that light.

**Q:** It sounds like you're being careful with the way you talk about light. Is there a reason you talk about emitting and reflecting light? Why don't you just say that it glows?

**A:** Because there's a difference between a GameObject that emits light and one that glows. If you add a Light component to your ball, it will start emitting light—but it won't look any different, because the Light only affects other GameObjects in the scene that reflect its light. If you want your GameObject to glow, you'll need to change its material or use another component that affects how it's rendered.

You can click on the Help icon for any component to bring up the Unity Manual page for it.



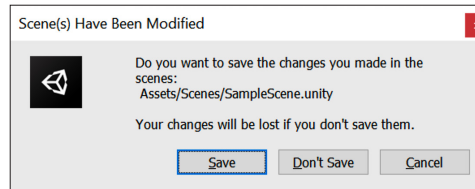
↑  
When you click on the Directional Light GameObject in the Hierarchy window, the Inspector shows you its components. It just has two: a Transform component that provides its position and rotation and a Light component that actually casts the light.

## Get creative!

We built these Unity Labs to give you a **platform to experiment on your own with C#** because that's the single most effective way for you to become a great C# developer. At the end of every Unity Lab, we'll include a few suggestions for things that you can try on your own. Take some time to experiment with everything you just learned before moving on to the next chapter:

- ★ Add a few more spheres to your scene. Try using some of the other billiard ball maps. You can download them all from the same location where you downloaded *8 Ball Texture.png* from.
- ★ Try adding other shapes by choosing Cube, Cylinder, or Capsule from the GameObject >> 3D Object menu.
- ★ Experiment with using different images as textures. See what happens to photos of people or scenery when you use them to create textures and add them to different shapes.
- ★ Can you create an interesting 3D scene out of shapes, textures, and lights?

When you're ready to move on to the next chapter, make sure you save your project, because you'll come back to it in the next lab.. Unity will prompt you to save when you quit.



The more C# code you write, the better you'll get at it. That's the most effective way for you to become a great C# developer. We designed these Unity Labs to give you a platform for practice and experimentation.

## BULLET POINTS

- The **Scene view** is your main interactive view of the world that you're creating.
- The **Move Gizmo** lets you move objects around your scene. The **Scale Gizmo** lets you modify your GameObjects' scale.
- The **Scene Gizmo** always displays the camera's orientation.
- Unity uses **materials** to provide color, patterns, textures, and other visual effects.
- Some materials use **textures**, or image files wrapped around shapes.
- Your game's scenery, characters, props, cameras, and lights are all built from **GameObjects**.
- GameObjects are the fundamental objects in Unity, and **components** are the basic building blocks of their behavior.
- Every GameObject has a **Transform component** that provides its position, rotation, and scale.
- The **Project window** gives you a folder-based view of your project's assets, including C# scripts and textures.
- The **Hierarchy window** shows all of the GameObjects in the scene.
- **GitHub for Unity** (<https://unity.github.com>) makes it easy to save your Unity projects in GitHub.
- **Unity Collaborate** also lets you back up projects to free cloud storage that comes with a Unity Personal account.



# Unity Lab #2

## Write C# Code for Unity

Unity isn't *just* a powerful, cross-platform engine and editor for building 2D and 3D games and simulations. It's also a **great way to get practice writing C# code**.

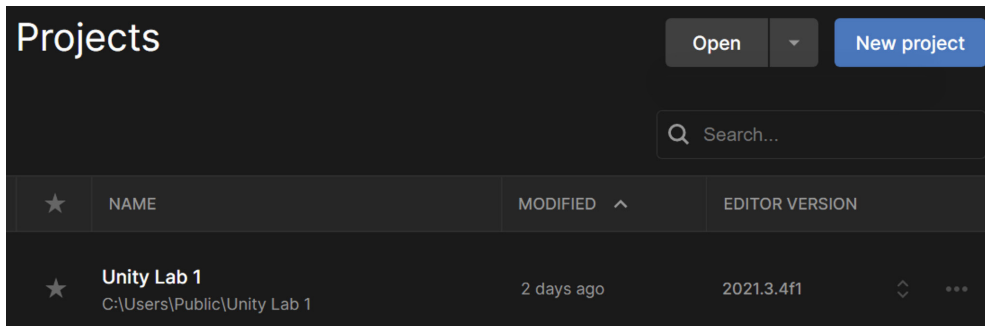
In the last Unity Lab, you learned how to navigate around Unity and your 3D space, and started to create and explore GameObjects. Now it's time to write some code to take control of your GameObjects. The whole goal of that lab was to get you oriented in the Unity editor (and give you an easy way to remind yourself of how to navigate around it if you need it).

In this Unity Lab, you'll start writing code to control your GameObjects. You'll write C# code to explore concepts you'll use in the rest of the Unity Labs, starting with adding a method that rotates the 8 Ball GameObject that you created in the last Unity Lab. You'll also start using the C# debugger with Unity to sleuth out problems in your games.

## C# scripts add behavior to your GameObjects

Now that you can add a GameObject to your scene, you need a way to make it, well, do stuff. That's where your C# skills come in. Unity uses **C# scripts** to define the behavior of everything in the game.

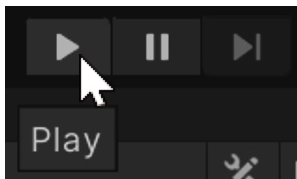
This Unity Lab will introduce tools that you'll use to work with C# and Unity. You're going to build a simple "game" that's really just a little bit of visual eye candy: you'll make your 8 ball fly around the scene. Start by going to Unity Hub and **opening the same project** that you created in the first Unity Lab.



This Unity Lab picks up where the first one left off, so go to Unity Hub and open the project you created in the last lab.

Here's what you'll do in this Unity Lab:

- 1 **Attach a C# script to your GameObject.** You'll add a Script component to your Sphere GameObject. When you add it, Unity will create a class for you. You'll modify that class so that it drives the 8 ball sphere's behavior.
- 2 **Use Rider to edit the script.** Remember how you set the Unity editor's preferences to make Rider the script editor? That means you can just double-click on the script in the Unity editor and it will open up in Rider.
- 3 **Play your game in Unity.** There's a Play button at the top of the screen. When you press it, it starts executing all of the scripts attached to the GameObjects in your scene. You'll use that button to run the script that you added to the sphere.



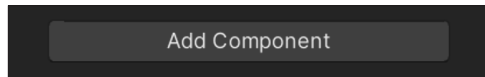
← The Play button does not save your game! So make sure you save early and save often. A lot of people get in the habit of saving the scene every time they run the game.

- 4 **Use Unity and Rider together to debug your script.** You've already seen how valuable the Rider debugger is when you're trying to track down problems in your C# code. Unity and Rider work together seamlessly so you can add breakpoints, use the Locals window, and work with the other familiar tools in the Rider debugger while your game is running.

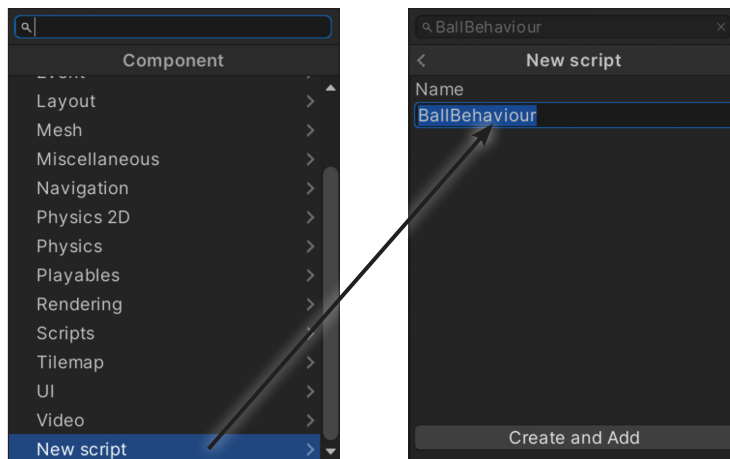
### Add a C# script to your GameObject

Unity is more than an amazing platform for building 2D and 3D games. Many people use it for artistic work, data visualization, augmented reality, and more. It's especially valuable to you, as a C# learner, because you can write code to control everything that you see in a Unity game. That makes Unity *a great tool for learning and exploring C#*.

Let's start using C# and Unity right now. Make sure the Sphere GameObject is selected, then **click the Add Component button** at the bottom of the Inspector window.



When you click it, Unity pops up a window with all of the different kinds of components that you can add—and there are *a lot* of them. **Choose “New script”** to add a new C# script to your Sphere GameObject. You'll be prompted for a name. **Name your script BallBehaviour.**



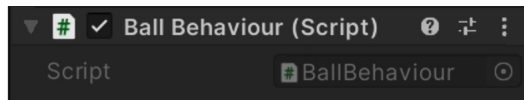
**Watch it!**

**Unity code uses British spelling.**

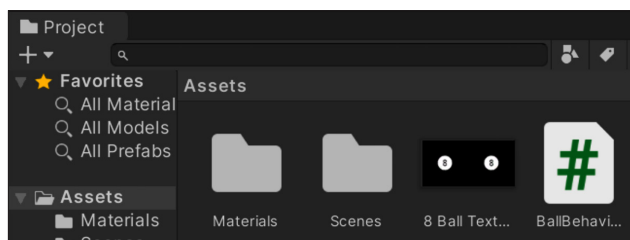
*If you're American (like us), or if you're used to the US*

*spelling of the word **behavior**, you'll need to be careful when you work with Unity scripts because the class names often feature the British spelling **behaviour**.*

Click the “Create and Add” button to add the script. You'll see a component called *Ball Behaviour (Script)* appear in the Inspector window.



You'll also see the C# script in the Project window.



The Project window gives you a folder-based view of your project. Your Unity project is made up of files: media files, data files, C# scripts, textures, and more. Unity calls these files **assets**. The Project window was displaying a folder called **Assets** when you right-clicked inside it to import your texture, so Unity added it to that folder.

Did you notice a folder called *Materials* appeared in the Project window as soon as you dragged the 8 ball texture onto your sphere?

### Write C# code to rotate your sphere

In the first lab, you told Unity to use Rider as its external script editor. So go ahead and **double-click on your new C# script**. When you do, **Unity will open your script in Rider**. Your C# script contains a class called BallBehaviour with two empty methods called Start and Update:

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

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

    }

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

    }
}
```

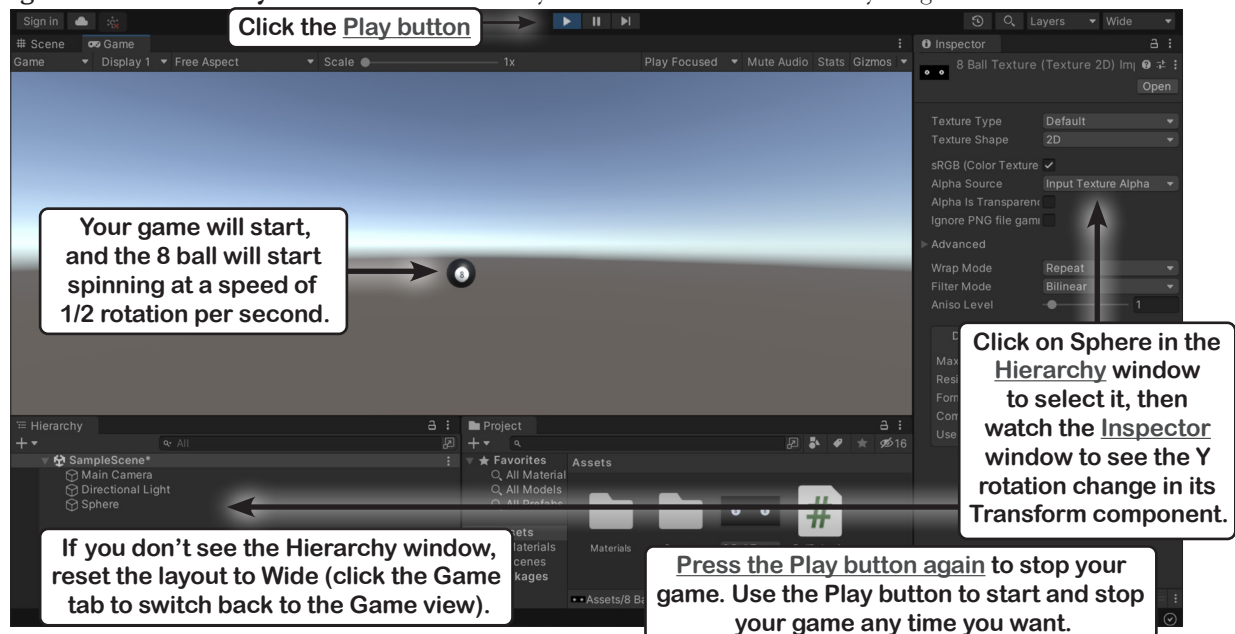
You opened your C# script in Rider by clicking on it in the Hierarchy window, which shows you a list of every **GameObject** in the current scene. When Unity created your project, it added a scene called SampleScene with a camera and a light. You added a sphere to it, so your Hierarchy window will show all of those things.

If Unity didn't launch Rider and open your C# script in it, go back to the beginning of Unity Lab 1 and make sure you followed the steps to set the External Tools preferences.

Here's a line of code that will rotate your sphere. **Add it to your Update method**:

**transform.Rotate(Vector3.up, 180 \* Time.deltaTime);**

Now **go back to the Unity editor** and click the Play button in the toolbar:



### Your Code Up Close



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

You learned about namespaces in Chapter 2. When Unity created the file with the C# script, it added `using` lines at the top so it can use code in the `UnityEngine` namespace and other commonly used namespaces.

```
public class BallBehaviour : MonoBehaviour
```

```
{
```

```
    // Start is called before the first frame update
```

```
    void Start()
```

```
    {
```

```
    }
```

A frame is a fundamental concept of animation. Unity draws one still frame, then draws the next one very quickly, and your eye interprets changes in these frames as movement. Unity calls the `Update` method for every `GameObject` before each frame so it can move, rotate, or make any other changes that it needs to make. A faster computer will run at a higher frame rate—or number of frames per second (FPS)—than a slower one.

```
    // Update is called once per frame
```

```
    void Update()
```

```
    {
```

```
        transform.Rotate(Vector3.up, 180 * Time.deltaTime);
```

```
    }
```

```
}
```

The `transform.Rotate` method causes a `GameObject` to rotate. The first parameter is the axis to rotate around. In this case, your code used `Vector3.up`, which tells it to rotate around the Y axis. The second parameter is the number of degrees to rotate.

Different computers will run your game at different frame rates. If it's running at 30 FPS, we want one rotation every 60 frames. If it's running at 120 FPS, it should rotate once every 240 frames. Your game's frame rate may even change if it needs to run more or less complex code.

That's where the `Time.deltaTime` value comes in handy. Every time the Unity engine calls a `GameObject`'s `Update` method—once per frame—it sets `Time.deltaTime` to the fraction of a second since the last frame. Since we want our ball to do a full rotation every two seconds, or 180 degrees per second, all we need to do is multiply it by `Time.deltaTime` to make sure that it rotates exactly as much as it needs to for that frame.

Inside your `Update` method, multiplying any value by `Time.deltaTime` turns it into that value per second.

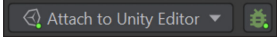
`Time.deltaTime` is static—and like we saw in Chapter 3, you don't need an instance of the `Time` class to use it.

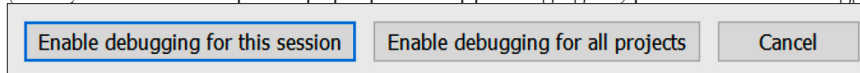
### Add a breakpoint and debug your game

Let's debug your Unity game. First **stop your game** if it's still running (by pressing the Play button again). Then switch over to Rider, and **add a breakpoint** on the line that you added to the Update method.

```
14 void Update()
15 {
16     transform.Rotate(Vector3.up, angle:180 * Time.deltaTime);
17 }
```

Now find the button at the top of Rider that starts the debugger:

- ★ First you'll need to **attach** Rider to Unity. Find this section of the toolbar:  and click the debug button, or choose *Run >> Debug 'Attach to Unity Editor'* from the menu
- ★ Switch over to Unity. You'll see a dialog window—press the “Enable debugging for this session” button (or if you want to keep that pop-up from appearing again, press “Enable debugging for all projects”).



Rider is now **attached** to Unity, which means it can debug your game. You can tell that the debugger is attached to Unity by checking the bug icon in the lower right corner of the Unity editor:



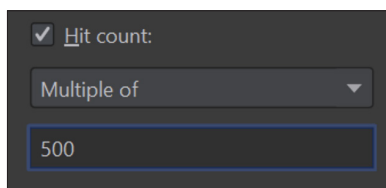
Now **press the Play button in Unity** to start your game. Alternately, you can use the **Run button in Rider** (or choose Run from the menu)—Rider will tell Unity to start playing. Since Rider is attached to Unity, it **breaks immediately** on the breakpoint that you added, just like with any other breakpoint you've set.

### Use a hit count to skip frames

← Congratulations, you're now debugging a game!

Sometimes it's useful to let your game run for a while before your breakpoint stops it. For example, you might want your game to spawn and move its enemies before your breakpoint hits. Let's tell your breakpoint to break every 500 frames. You can do that by adding a **Hit Count condition** to your breakpoint:

- ★ Right-click on the breakpoint dot (●) at the left side of the line and choose **More** from the pop-up menu. This will expand the window to give you more options for editing your breakpoint.
- ★ Check the *Hit Count* box, choose *Multiple of* from the dropdown, and enter 500 in the box:



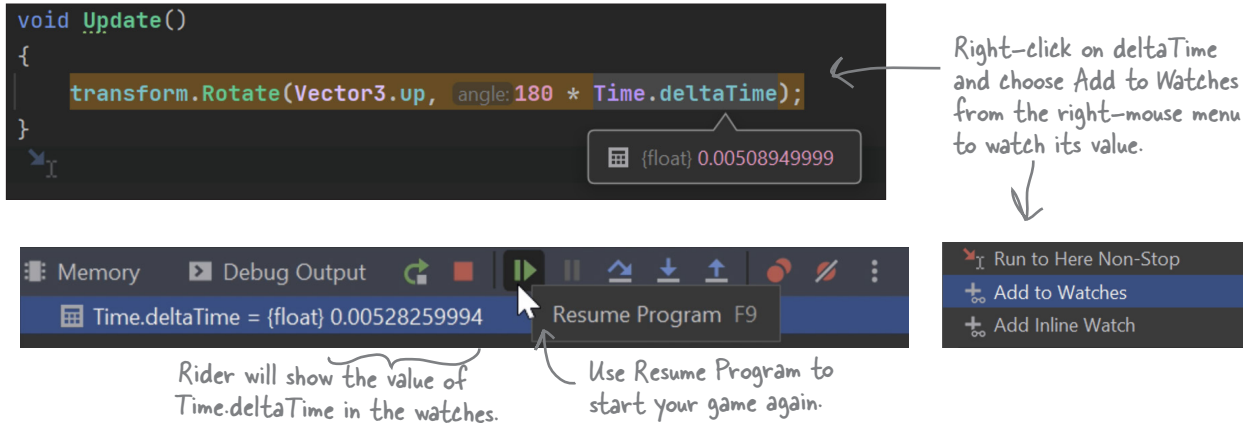
Now the breakpoint will only pause the game every 500 times the Update method is run—or every 500 frames. So if your game is running at 60 FPS, that means when you press Continue the game will run for a little over 8 seconds before it breaks again. **Press Continue, then switch back to Unity** and watch the ball spin until the breakpoint breaks.



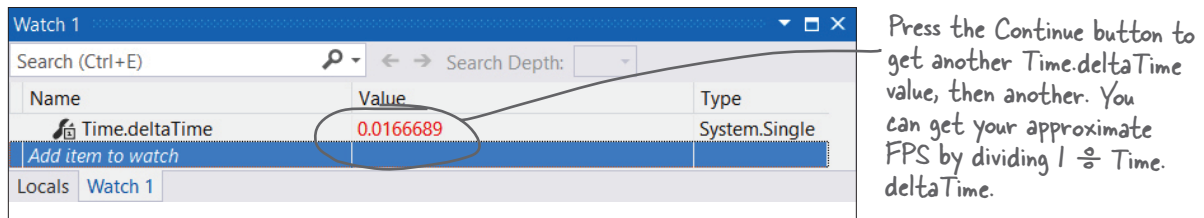
### Use the debugger to understand Time.deltaTime

You're going to be using `Time.deltaTime` in many of the Unity Labs projects. Let's take advantage of your breakpoint and use the debugger to really understand what's going on with this value.

While your game is paused on the breakpoint in Rider, **hover over `Time.deltaTime`** to see the fraction of a second that elapsed since the previous frame (you'll need to put your mouse cursor over `deltaTime`). Then **add a watch for `Time.deltaTime`** by right-clicking on `Time.deltaTime` and choosing Add Watch from the right-mouse menu.



**Continue debugging** by clicking Resume Program (just like any other C# program) to resume your game. The ball will start rotating again, and after another 500 frames the breakpoint will trigger again. You can keep running the game for 500 frames at a time. Keep your eye on the Watch window each time it breaks.



**Stop debugging** by clicking the stop button next to Resume Program to stop your program. Then **start debugging again**. Since your game is still running, the breakpoint will continue to work when you reattach Rider to Unity. Once you're done debugging, **right-click on your breakpoint and uncheck Enabled** so Rider will still keep track of it but not break when it's hit. **Stop debugging** one more time to detach from Unity. Go back to Unity and **stop your game**—and save it, because the Play button doesn't automatically save the game.

The Play button in Unity starts and stops your game. Rider will stay attached to Unity even when the game is stopped.

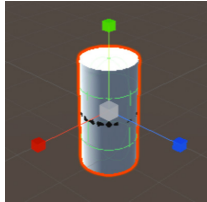


Debug your game again and hover over "`Vector3.up`" to inspect its value—you'll have to put your mouse cursor over `up`. It has a value of `(0.0, 1.0, 0.0)`. What do you think that means?

### Add a cylinder to show where the Y axis is

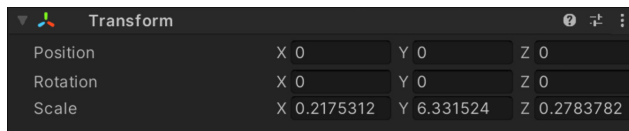
Your sphere is rotating around the Y axis at the very center of the scene. Let's add a very tall and very skinny cylinder to make it visible. **Create a new cylinder** by choosing *3D Object >> Cylinder* from the GameObject menu. Make sure it's selected in the Hierarchy window, then look at the Inspector window and check that Unity created it at position (0, 0, 0)—if not, use the context menu (ⓘ) to reset it.

Let's make the cylinder tall and skinny. Choose the Scale tool from the Tools panel: either click on it (⏏) or press the R key. You should see the Scale Gizmo appear on your cylinder:



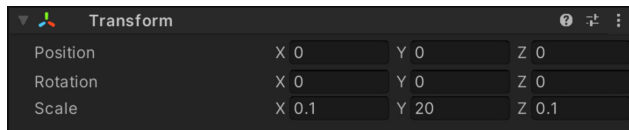
The Scale Gizmo looks a lot like the Move Gizmo, except that it has cubes instead of cones at the end of each axis. Your new cylinder is sitting on top of the sphere—you might see just a little of the sphere showing through the middle of the cylinder. When you make the cylinder narrower by changing its scale along the X and Z axes, the sphere will get uncovered.

Click and drag the green cube up to elongate your cylinder along the Y axis. Then click on the red cube and drag it toward the cylinder to make it very narrow along the X axis, and do the same with the blue cube to make it very narrow along the Z axis. Watch the Transform panel in the Inspector as you change the cylinder's scale—the Y scale will get larger, and the X and Z values will get much smaller.



**Click on the X label in the Scale row in the Transform panel and drag up and down.** Make sure you click the actual X label to the left of the input box with the number. When you click the label it turns blue, and a blue box appears around the X value. As you drag your mouse up and down, the number in the box goes up and down, and the Scene view updates the scale in as you change it. Look closely as you drag—the scale can be positive and negative.

Now **select the number inside the X box and type .1**—the cylinder gets very skinny. Press Tab and type 20, then press Tab again and type .1, and press Enter.



Now your sphere has a very long cylinder going through it that shows the Y axis where Y = 0.



### Add fields to your class for the rotation angle and speed

In Chapter 3 you learned how C# classes can have **fields** that store values methods can use. Let's modify your code to use fields. Add these four lines just under the class declaration, **immediately after the first curly brace {**:

```
public class BallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;
```

These are just like the fields that you added to the projects in Chapters 3 and 4. They're variables that keep track of their values—each time Update is called it reuses the same field over and over again.

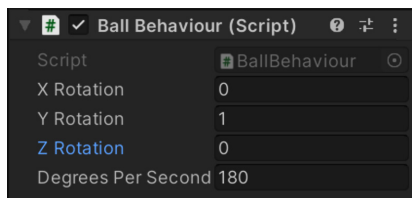
The XRotation, YRotation, and ZRotation fields each contain a value between 0 and 1, which you'll combine to create a **vector** that determines the direction that the ball will rotate:

```
new Vector3(XRotation, YRotation, ZRotation)
```

The DegreesPerSecond field contains the number of degrees to rotate per second, which you'll multiply by Time.deltaTime just like before. **Modify your Update method to use the fields.** This new code creates a Vector3 variable called **axis** and passes it to the transform.Rotate method:

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
}
```

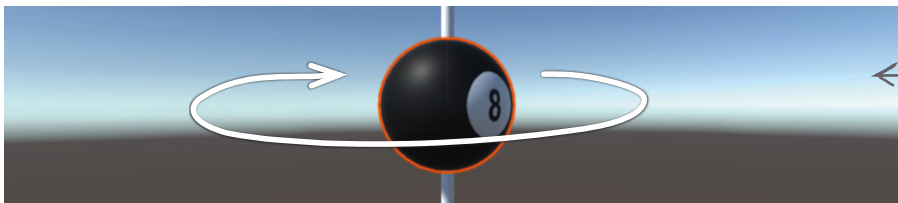
Select the Sphere in the Hierarchy window. Your fields now show up in the Script component. When the Script component renders fields, it adds spaces between the capital letters to make them easier to read.



When you add public fields to a class in your Unity script, the Script component displays input boxes that let you modify those fields. If you modify them while the game is not running, the updated values will get saved with your scene. You can also modify them while the game is running, but they'll revert when you stop the game.

Run your game again. **While it's running**, select the Sphere in the Hierarchy window and change the degrees per second to 360 or 90—the ball starts to spin at twice or half the speed. Stop your game—and the field will reset to 180.

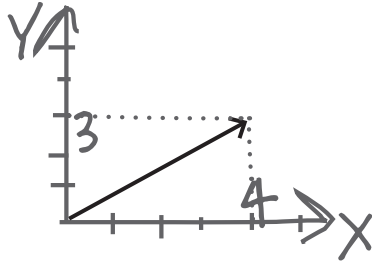
While the game is stopped, use the Unity editor to change the X Rotation field to 1 and the Y Rotation field to 0. Start your game—the ball will rotate away from you. Click the X Rotation label and drag it up and down to change the value while the game is running. As soon as the number turns negative, the ball starts rotating toward you. Make it positive again and it starts rotating away from you.



When you use the Unity editor to set the Y Rotation field to 1 and then start your game, the ball rotates clockwise around the Y axis.

## Use Debug.DrawRay to explore how 3D vectors work

A **vector** is a value with a **length** (or magnitude) and a **direction**. If you ever learned about vectors in a math class, you probably saw lots of diagrams like this one of a 2D vector:



Here's a diagram of a two-dimensional vector. You can represent it with two numbers: its value on the X axis (4) and its value on the Y axis (3), which you'd typically write as (4, 3).

That's not hard to understand...on an intellectual level. But even those of us who took a math class that covered vectors don't always have an **intuitive** grasp of how vectors work, especially in 3D. Here's another area where we can use C# and Unity as a tool for learning and exploration.

### Use Unity to visualize vectors in 3D

You're going to add code to your game to help you really "get" how 3D vectors work. Start by having a closer look at the first line of your Update method:

```
Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
```

What does this line tell us about the vector?

- ★ **It has a type: Vector3.** Every variable declaration starts with a type. Instead of using string, int, or bool, you're declaring it with the type Vector3. This is a type that Unity uses for 3D vectors.
- ★ **It has a variable name: axis.**
- ★ **It uses the new keyword to create a Vector3.** It uses the XRotation, YRotation, and ZRotation fields to create a vector with those values.

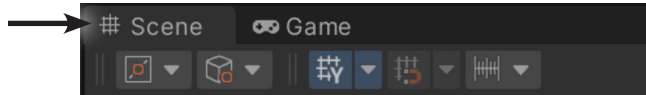
So what does that 3D vector look like? There's no need to guess—we can use one of Unity's useful debugging tools to draw the vector for us. **Add this line of code to the end of your Update method:**

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.Rotate(axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow);
}
```

The Debug.DrawRay method is a special method that Unity provides to help you debug your games. It draws a **ray**—which is a vector that goes from one point to another—and takes parameters for its start point, end point, and color. There's one catch: **the ray only appears in the Scene view**. The methods in Unity's Debug class are designed so that they don't interfere with your game. They typically only affect how your game interacts with the Unity editor.

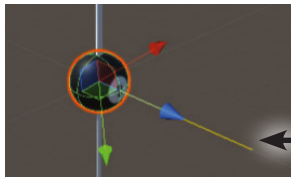
### Run the game to see the ray in the Scene view

Now run your game again. You won't see anything different in the Game view because `Debug.DrawRay` is a tool for debugging that doesn't affect gameplay at all. Use the Scene tab to **switch to the Scene view**. You may also need to **reset the Wide layout** by choosing Wide from the Layout dropdown.



Now you're back in the familiar Scene view. Do these things to get a real sense of how 3D vectors work:

- ★ Use the Inspector to **modify the BallBehaviour script's fields**. Set the X Rotation to 0, Y Rotation to 0, and **Z Rotation to 3**. You should now see a yellow ray coming directly out of the Z axis and the ball rotating around it (remember, the ray only shows up in the Scene view).



The vector (0, 0, 3) extends 3 units along the Z axis. Look closely at the grid in the Unity editor—the vector is exactly 3 units long. Try clicking and dragging the Z Rotation label in the Script component in the Inspector. The ray will get larger or smaller as you drag. When the Z value in the vector is negative, the ball rotates in the other direction.

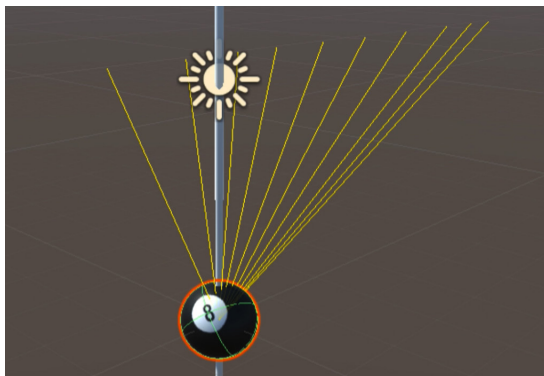
- ★ Set the Z Rotation back to 3. Experiment with dragging the X Rotation and Y Rotation values to see what they do to the ray. Make sure to reset the Transform component each time you change them.
- ★ Use the Hand tool and the Scene Gizmo to get a better view. Click the X cone on the Scene Gizmo to set it to the view from the right. Keep clicking the cones on the Scene Gizmo until you see the view from the front. It's easy to get lost—you can **reset the Wide layout to get back to a familiar view**.

### Add a duration to the ray so it leaves a trail

You can add a fourth argument to your `Debug.DrawRay` method call that specifies the number of seconds the ray should stay on the screen. Add **.5f** to make each ray stay on screen for half a second:

```
Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
```

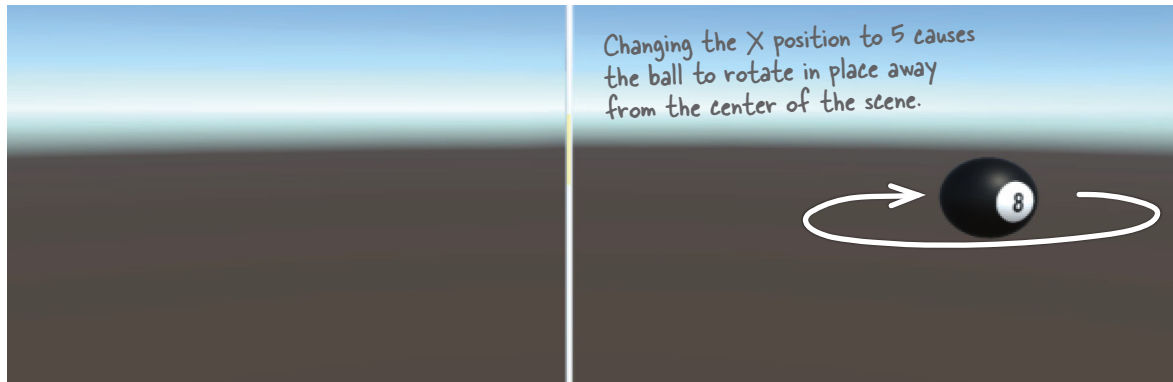
Now run the game again and switch to the Scene view. Now when you drag the numbers up and down, you'll see a trail of rays left behind. This looks really interesting, but more importantly, it's a great tool to visualize 3D vectors.



Making your ray leave a trail is a good way to help you develop an intuitive sense of how 3D vectors work.

### Rotate your ball around a point in the scene

Your code calls the `transform.Rotate` method to rotate your ball around its center, which changes its X, Y, and Z rotation values. **Select Sphere in the Hierarchy window and change its X position to 5** in the Transform component. Then **use the context menu (⋮) in the BallBehaviour Script component** to reset its fields. Run the game again—now the ball will be at position (5, 0, 0) and rotating around its own Y axis.



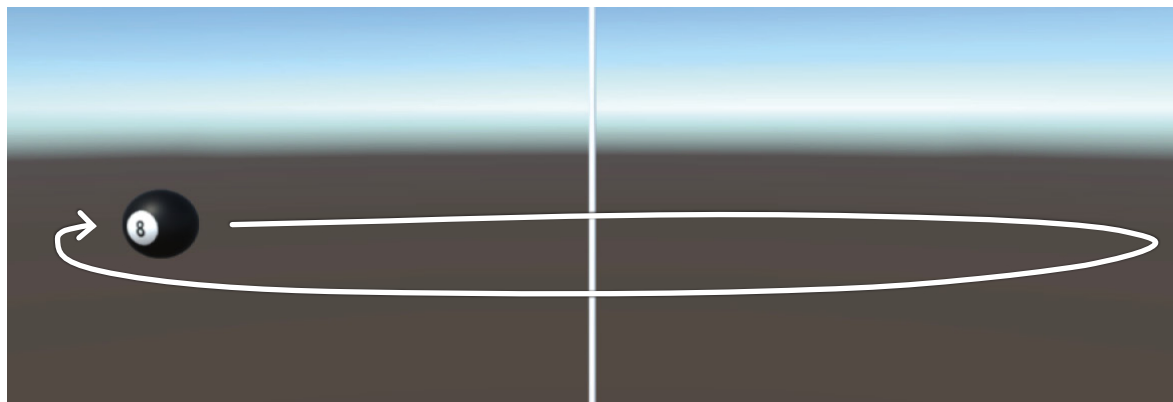
Let's modify the Update method to use a different kind of rotation. Now we'll make the ball rotate around the center point of the scene, coordinate (0, 0, 0), using the **transform.RotateAround method**, which rotates a GameObject around a point in the scene. (This is *different* from the `transform.Rotate` method you used earlier, which rotates a GameObject around its center.) Its first parameter is the point to rotate around. We'll use **Vector3.zero** for that parameter, which is a shortcut for writing `new Vector3(0, 0, 0)`.

Here's the new Update method:

```
void Update()
{
    Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
    transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
    Debug.DrawRay(Vector3.zero, axis, Color.yellow, .5f);
}
```

This new Update method rotates the ball around the point (0, 0, 0) in the scene.

Now run your code. This time it rotates the ball in a big circle around the center point:



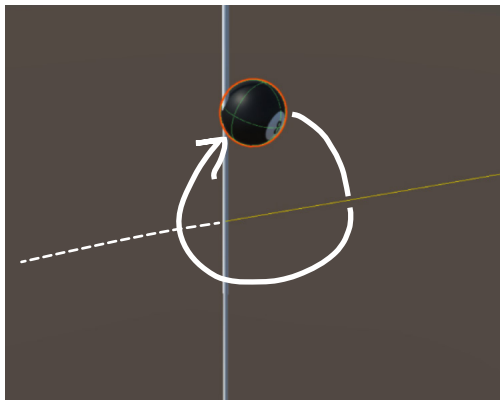


### Use Unity to take a closer look at rotation and vectors

You're going to be working with 3D objects and scenes in the rest of the Unity Labs throughout the book. Even those of us who spend a lot of time playing 3D video games don't have a perfect feel for how vectors and 3D objects work, and how to move and rotate in a 3D space. Luckily, Unity is a great tool to **explore how 3D objects work**. Let's start experimenting right now.

While your code is running, try changing parameters to experiment with the rotation:

- ★ **Switch back to the Scene view** so you can see the yellow ray that `Debug.DrawRay` renders in your `BallBehaviour.Update` method.
- ★ Use the Hierarchy window to **select the Sphere**. You should see its components in the Inspector window.
- ★ Change the **X Rotation, Y Rotation, and Z Rotation values** in the Script component to **10** so you see the vector rendered as a long ray. Use the Hand tool (Q) to rotate the Scene view until you can clearly see the ray.
- ★ Use the Transform component's context menu (☰) to **reset the Transform component**. Since the center of the sphere is now at the zero point in the scene, (0, 0, 0), it will rotate around its own center.
- ★ Then **change the X position in** the Transform component to **2**. The ball should now be rotating around the vector. You'll see the ball cast a shadow on the Y axis cylinder as it flies by.



While the game is running, set the X, Y, and Z Rotation fields in the `BallBehaviour` Script component to 10, reset the sphere's Transform component, and change its X position to 2—as soon as you do, it starts rotating around the ray.

Try **repeating the last three steps** for different values of X, Y, and Z rotation, resetting the Transform component each time so you start from a fixed point. Then try clicking the rotation field labels and dragging them up and down—see if you can get a feel for how the rotation works.

Unity is a great tool to explore how 3D objects work by modifying properties on your GameObjects in real time.

## Get creative!

This is your chance to **experiment on your own with C# and Unity**.

You've seen the basics of how you combine C# and Unity GameObjects. Take some time and play around with the different Unity tools and methods that you've learned about in the first two Unity Labs. Here are some ideas:

- ★ Add cubes, cylinders, or capsules to your scene. Attach new scripts to them—make sure you give each script a unique name!—and make them rotate in different ways.
- ★ Try putting your rotating GameObjects in different positions around the scene. See if you can make interesting visual patterns out of multiple rotating GameObjects.
- ★ Try adding a light to the scene. What happens when you use `transform.RotateAround` to rotate the new light around various axes?
- ★ Here's a quick coding challenge: try using `+=` to add a value to one of the fields in your `BallBehaviour` script. Make sure you multiply that value by `Time.deltaTime`. Try adding an `if` statement that resets the field to 0 if it gets too large.

Before you run the code, try to figure out what it will do. Does it act the way you expected it to act? Trying to predict how the code you added will act is a great technique for getting better at C#.

Take the time to experiment with the tools and techniques you just learned. This is a great way to take advantage of Unity and Rider as tools for exploration and learning.

## BULLET POINTS

- The **Scene Gizmo** always displays the camera's orientation.
- You can **attach a C# script** to any GameObject. The script's `Update` method will be called once per frame.
- The **`transform.Rotate` method** causes a GameObject to rotate a number of degrees around an axis.
- Inside your `Update` method, multiplying any value by **`Time.deltaTime`** turns it into that value per second.
- You can **attach** the Rider debugger to Unity to debug your game while it's running. It will stay attached to Unity even when your game is not running.
- Adding a **Hit Count condition** to a breakpoint to makes it break after the statement has executed a certain number of times.
- A **field** is a variable that lives inside of a class outside of its methods, and it retains its value between method calls.
- Adding public fields to the class in your Unity script makes the Script component show **input boxes that let you modify those fields**. It adds spaces between capital letters in the field names to make them easier to read.
- You can create 3D vectors using **new `Vector3`**. (You learned about the new keyword in Chapter 3.)
- The **`Debug.DrawRay` method** draws a vector in the Scene view (but not the Game view). You can use vectors as a debugging tool, but also as a learning tool.
- The **`transform.RotateAround` method** rotates a GameObject around a point in the scene.

# Unity Lab #3

## GameObject Instances

C# is an object-oriented language, and since these Head First C# Unity Labs are all **about getting practice writing C# code**, it makes sense that these labs will focus on creating objects.

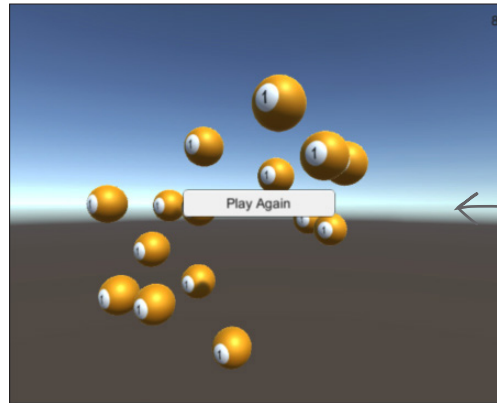
You've been creating objects in C# since you learned about the **new** keyword in Chapter 3. In this Unity Lab you'll **create instances of a Unity GameObject** and use them in a complete, working game. This is a great jumping-off point for writing Unity games in C#.

The goal of the next two Unity Labs is to **create a simple game** using the familiar billiard ball from the last lab. In this lab, you'll build on what you learned about C# objects and instances to start building the game. You'll use a **prefab**—Unity's tool for creating instances of GameObjects—to create lots of instances of a GameObject, and you'll use scripts to make your GameObjects fly around your game's 3D space.

## Let's build a game in Unity!

Unity is all about building games. So in the next two Unity Labs, you'll use what you've learned about C# to build a simple game. Here's the game that you're going to create:

When you start the game, the scene slowly fills up with billiard balls. The player needs to keep clicking them to make them disappear. Once there are 15 balls in the scene, the game is over.

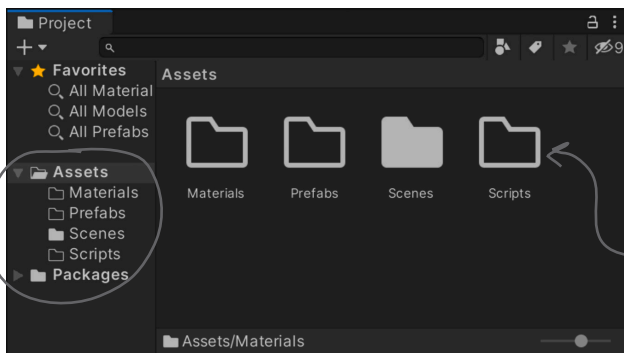


The game displays a score in the upper-right corner. Players score a point for each ball they click on.

When the game is over, a Play Again button lets the player start a new game.

So let's get started. The first thing you'll do is get your Unity project set up. This time we'll keep the files a little more organized, so you'll create separate folders for your materials and scripts—and one more folder for prefabs (which you'll learn about later in the lab):

1. Before you begin, close any Unity project that you have open. Also close Rider—you'll let Unity open it for you.
2. **Create a new Unity project** using the 3D template, just like you did for the previous Unity Labs. Give it a name to help you remember which labs it goes with ("Unity Labs 3 and 4").
3. Choose the Wide layout so your screen matches the screenshots.
4. Create a folder for your materials underneath the Assets folder. **Right-click on the Assets folder** in the Project window and choose Create >> Folder. Name it **Materials**.
5. Create another folder under Assets named **Scripts**.
6. Create one more folder under Assets named **Prefabs**.



Make sure you create the Materials, Scripts, and Prefabs folders underneath Assets.

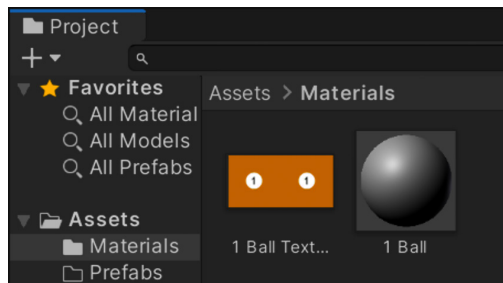
The Project window shows folders as outlines when they're empty.

### Create a new material inside the Materials folder

Double-click on your new Materials folder to open it. You'll create a new material here.

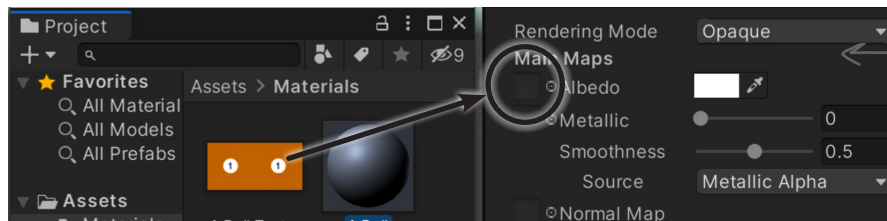
Go to <https://github.com/head-first-csharp/fourth-edition> and click on the Billiard Ball Textures link (just like you did in the first Unity lab) and download the texture file **1 Ball Texture.png** into a folder on your computer, then drag it into your Materials folder—just like you did with the downloaded file in the first Unity Lab, except this time drag it into the Materials folder you just created instead of the parent Assets folder.

Now you can create the new material. Right-click on the Materials folder in the Project window and **choose Create >> Material**. Name your new material **1 Ball**. You should see it appear in the Materials folder in the Project window.



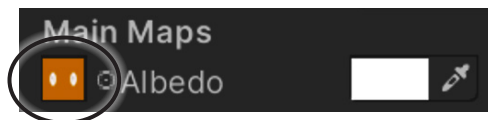
In the previous Unity Labs we used a texture, or a bitmap image file that Unity can wrap around GameObjects. When you dragged the texture onto a sphere, Unity automatically created a material, which is what Unity uses to keep track of information about how a GameObject should be rendered that can have a reference to a texture. This time you're creating the material manually. Just like last time, you may need to click the Download button on the GitHub page to download the texture PNG file.

Make sure the 1 Ball material is selected in the Materials window, so it shows up in the Inspector. Click on the *1 Ball Texture* file and **drag it into the box to the left of the Albedo label**.

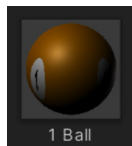


Select the 1 Ball material in the Project window so you can see its properties, then drag the texture map onto the box to the left of the Albedo label.

You should now see a tiny little picture of the 1 Ball texture in the box to the left of Albedo in the Inspector.



Now your material looks like a billiard ball when wrapped around a sphere.



### GameObjects reflect light from their surfaces.

### Behind the Scenes



When you see an object in a Unity game with a color or texture map, you're seeing the surface of a GameObject reflecting light from the scene, and the **albedo** controls the color of that surface. Albedo is a term from physics (specifically astronomy) that means the color that's reflected by an object. You can learn more about albedo from the Unity Manual. Choose "Unity Manual" from the Help menu to open the manual in a browser and search for "albedo"—there's a manual page that explains albedo color and transparency.

### Spawn a billiard ball at a random point in the scene

Create a new Sphere GameObject with a script called OneBallBehaviour:

- ★ Choose 3D Object >> Sphere from the GameObject menu to **create a sphere**.
- ★ Drag your new **1 Ball material** onto it to make it look like a billiard ball.
- ★ Next, **right-click on the Scripts folder** that you created in the Project window and **create a new C# script** named OneBallBehaviour.
- ★ **Drag the script onto the Sphere** in the Hierarchy window. Select the sphere and make sure a Script component called “One Ball Behaviour” shows up in the Inspector window.

Double-click on your new script to edit it in Rider. **Add exactly the same code** that you used in BallBehaviour in the first Unity Lab, then **comment out the Debug.DrawRay line** in the Update method.

Your OneBallBehaviour script should now look like this:

```
public class OneBallBehaviour : MonoBehaviour
{
    public float XRotation = 0;
    public float YRotation = 1;
    public float ZRotation = 0;
    public float DegreesPerSecond = 180;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        Vector3 axis = new Vector3(XRotation, YRotation, ZRotation);
        transform.RotateAround(Vector3.zero, axis, DegreesPerSecond * Time.deltaTime);
        // Debug.DrawRay(Vector3.zero, axis, Color.yellow);
    }
}
```

We won't include the using lines in the script code, but assume they're there.

When you add a Start method to a GameObject, Unity calls that method every time a new instance of that object is added to the scene. If the Start method is in a script attached to a GameObject that appears in the Hierarchy window, that method will get called as soon as the game starts.

Unity often instantiates a GameObject some time before it's added to the scene. It only calls the Start method when the GameObject is actually added to the scene.

You won't need this line, so comment it out.

Now modify the Start method to move the sphere to a random position when it's created. You'll do this by setting **transform.position**, which changes the position of the GameObject in the scene. Here's the code to position your ball at a random point—**add it to the Start** method of your OneBallBehaviour script:

```
// Start is called before the first frame update
void Start()
{
    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);
}
```

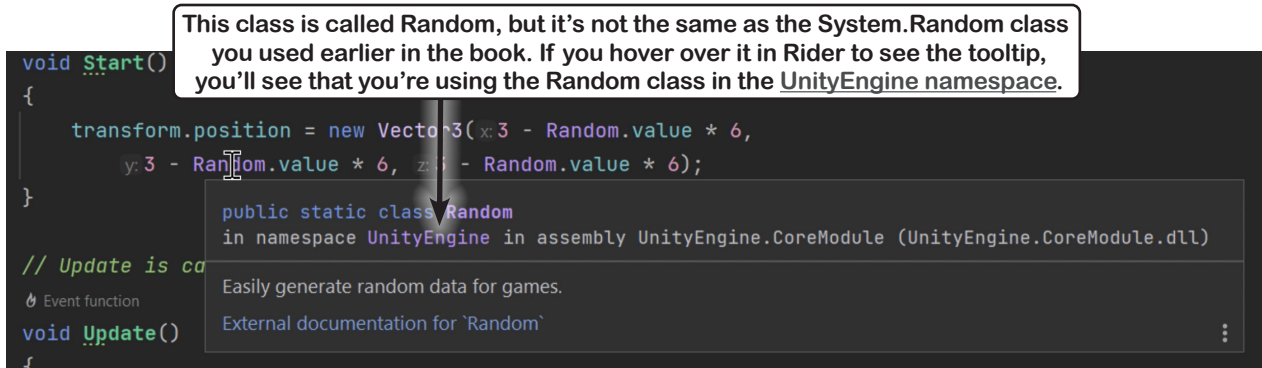
Remember, the Play button does not save your game! Make sure you save early and save often.

**Use the Play button in Unity to run your game.** A ball should now be circling the Y axis at a random point. Stop and start the game a few times. The ball should spawn at a different point in the scene each time.



### Use the debugger to understand Random.value

You've used the Random class in the .NET System namespace a few times already. You used it to scatter the animals in the animal matching game in Chapter 1 and to pick random cards in Chapter 3. This Random class is different—try hovering over the Random keyword in Rider.



You can see from the code that this new Random class is different from the one you used before. Earlier you called Random.Next to get a random value, and that value was a whole number. This new code uses **Random.value**, but that's not a method—it's actually a property.

Use the Rider debugger to see the kinds of values that this new Random class gives you. Click the debug button to attach Rider to Unity ( ). Then **add a breakpoint** to the line you added to the Start method.

Unity may prompt you to enable debugging, just like in the last Unity Lab.

Now go back to Unity and **start your game**. It should break as soon as you press the Play button. Hover your cursor over Random.value—make sure it's over **value**. Rider will show you its value in a tooltip:



Keep Rider attached to Unity and restart your game a few times. You'll get a new random number between 0 and 1 each time you restart it.

Keep Rider attached to Unity, then go back to the Unity editor and **stop your game** (in the Unity editor, not in Rider). Start your game again. Do it a few more times. You'll get a different random value each time. That's how UnityEngine.Random works: it gives you a new random value between 0 and 1 each time you access its value property.

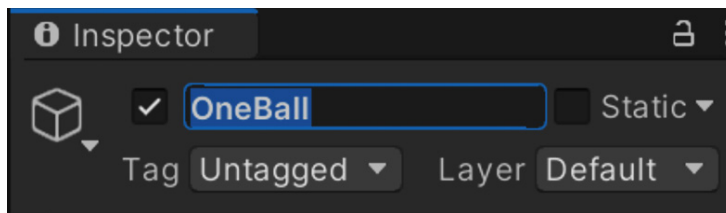
Press the Resume Program button ( ) to resume your game. It should keep running—the breakpoint was only in the Start method, which is just called once for each GameObject instance, so it won't break again. Then go back to Unity and stop the game.

**You can't edit scripts in Rider while it's attached to Unity, so click the square Stop Debugging button to detach the Rider debugger from Unity.**

### Turn your GameObject into a prefab

In Unity, a **prefab** is a GameObject that you can instantiate in your scene. Over the past few chapters you've been working with object instances, and creating objects by instantiating classes. Unity lets you take advantage of objects and instances, so you can build games that reuse the same GameObjects over and over again. Let's turn your 1 ball GameObject into a prefab.

GameObjects have names.. Change the name of your GameObject to *OneBall*. Start by **selecting your sphere**, by clicking on it in the Hierarchy window or in the scene. Then use the Inspector window to **change its name to OneBall**.



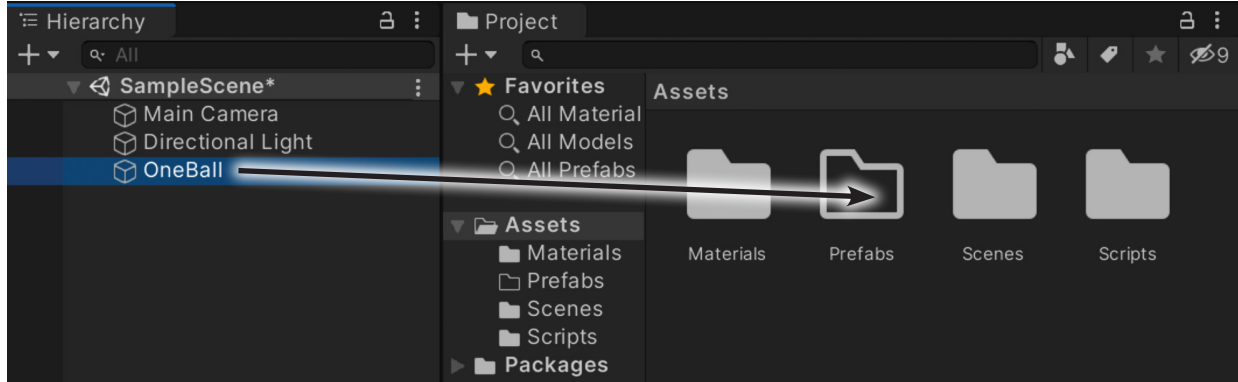
**Watch it!**

**Rider won't let you edit code while it's attached to Unity.**

*If you try to edit your code but find that Rider won't let you make any changes, that means Rider is probably still attached to Unity! Press the square Stop Debugging button to detach it.*

← You can also rename a GameObject by right-clicking on it in the Hierarchy window and choosing Rename.

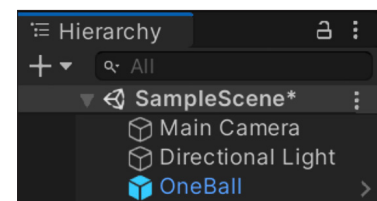
Now you can turn your GameObject into a prefab. **Drag OneBall from the Hierarchy window into the Prefabs folder.**



OneBall should now appear in your Prefabs folder. Notice that **OneBall is now blue in the Hierarchy window**. This indicates that it's now a prefab—Unity turned it blue to tell you that an instance of a prefab is in your hierarchy. That's fine for some games, but for this game we want all of the instances of the balls to be created by scripts.

Right-click on OneBall in the Hierarchy window **and delete the OneBall GameObject from the scene**. You should now only see it in the Project window, and not in the Hierarchy window or the scene.

**Have you been saving your scene as you go? Save early, save often!**



↑ When a GameObject is blue in the Hierarchy window, Unity is telling you it's a prefab instance.

### Create a script to control the game

The game needs a way to add balls to the scene (and eventually keep track of the score, and whether or not the game is over).

Right-click on the Scripts folder in the Project window and **create a new script called GameController**. Your new script will use two methods available in any GameObject script:

★ **The Instantiate method creates a new instance of a GameObject.**

When you're instantiating GameObjects in Unity, you don't typically use the **new** keyword like you saw in Chapter 2. Instead, you'll use the Instantiate method, which you'll call from the AddABall method.

★ **The InvokeRepeating method calls another method in the script over and over again.** In this case, it will wait one and a half seconds, then call the AddABall method once a second for the rest of the game.

Here's the source code for it:

```
public class GameController : MonoBehaviour
{
    public GameObject OneBallPrefab;

    void Start()
    {
        InvokeRepeating("AddABall", 1.5F, 1);
    }

    void AddABall()
    {
        Instantiate(OneBallPrefab);
    }
}
```

What's the type of the second argument that you're passing to InvokeRepeating?

Unity's InvokeRepeating method calls another method over and over again. Its first parameter is a string with the name of the method to call ("invoke" just means calling a method).

This is a method called AddABall. All it does is create a new instance of a prefab.

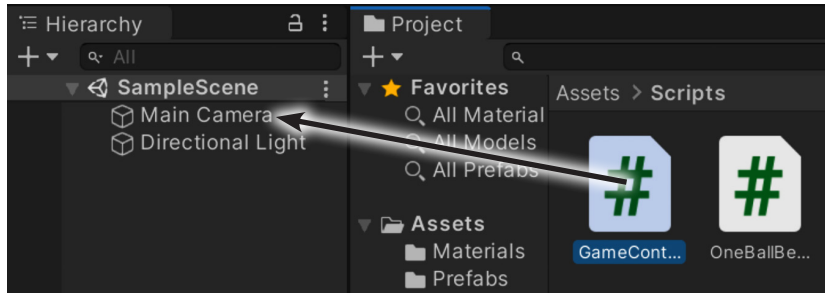
You're passing the OneBallPrefab field as a parameter to the Instantiate method, which Unity will use to create an instance of your prefab.



Unity will only run scripts that are attached to GameObjects in a scene. The GameController script will create instances of our OneBall prefab, but we need to attach it to something. Luckily, we already know that a camera is just a GameObject with a Camera component (and also an AudioListener). The Main Camera will always be available in the scene. So...what do you think you'll do with your new GameController script?

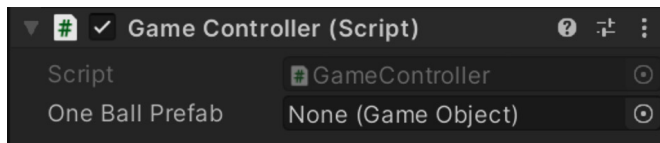
### Attach the script to the Main Camera

Your new GameController script needs to be attached to a GameObject to run. Luckily, the Main Camera is just another GameObject—it happens to be one with a Camera component and an AudioListener component—so let's attach your new script to it. **Drag your GameController** script out of the Scripts folder in the Project window and **onto the Main Camera** in the Hierarchy window.



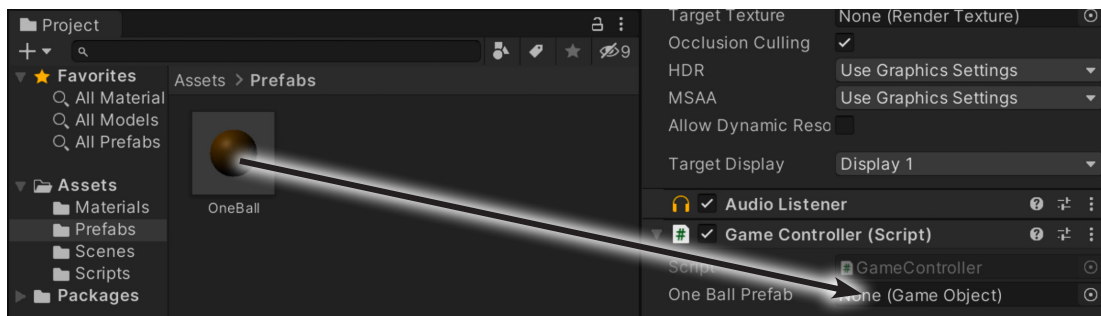
You learned all about public versus private fields in Chapter 5. When a script class has a public field, the Unity editor shows that field in the Script component in the Inspector. It adds spaces between uppercase letters to make its name easier to read.

Look in the Inspector—you'll see a component for the script, exactly like you would for any other GameObject. The script has a **public field called OneBallPrefab**, so Unity displays it in the Script component.

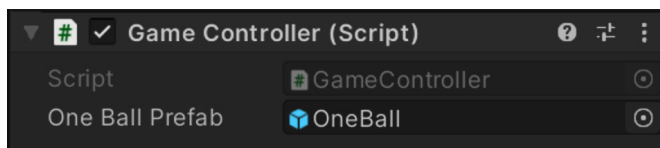


Here's the OneBallPrefab field in your GameController class. Unity added spaces before uppercase letters to make it easier to read (just like we saw in the last lab).

The OneBallPrefab field still says None, so we need to set it. **Drag OneBall out of the Prefabs folder and onto the box next to the One Ball Prefab label.**



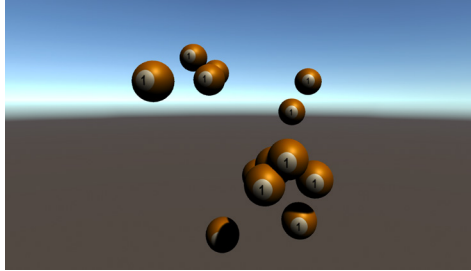
Now the GameController's OneBallPrefab field contains a **reference** to the OneBall prefab:



Go back to the code and **look closely the AddABall method**. It calls the Instantiate method, passing it the OneBallPrefab field as an argument. You just set that field so that it contains your prefab. So every time GameController calls its AddABall method, it will **create a new instance of the OneBall prefab**.

### Press Play to run your code

Your game is all ready to run. The GameController script attached to the Main Camera will wait 1.5 seconds, then instantiate a OneBall prefab every second. Each instantiated OneBall's Start method will move it to a random position in the scene, and its Update method will rotate it around the Y axis every 2 seconds using OneBallBehaviour fields (just like in the last Lab). Watch as the play area slowly fills up with rotating balls:

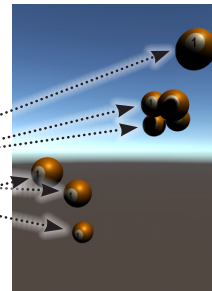
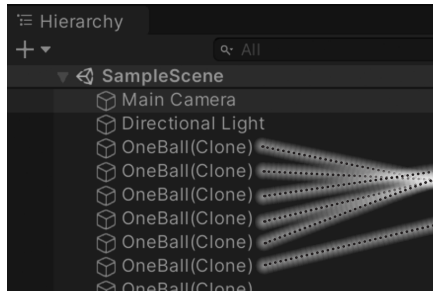


Unity calls every  
GameObject's Update  
method before each frame.  
That's called the update loop.

When you  
instantiate  
GameObjects  
in your code,  
they show  
up in the  
Hierarchy  
window when  
you run your  
game.

### Watch the live instances in the Hierarchy window

Each of the balls flying around the scene is an instance of the OneBall prefab. Each of the instances has its own instance of the OneBallBehaviour class. You can use the Hierarchy window to track all of the OneBall instances—as each one is created, a “OneBall(Clone)” entry is added to the Hierarchy.



We've included some  
coding exercises in the  
Unity Labs. They're  
just like the exercises  
in the rest of the  
book—and remember,  
it's not cheating to  
peek at the solution.

**Click on any of the OneBall(Clone) items** to view it in the Inspector. You'll see its Transform values change as it rotates, just like in the last lab.



### Exercise

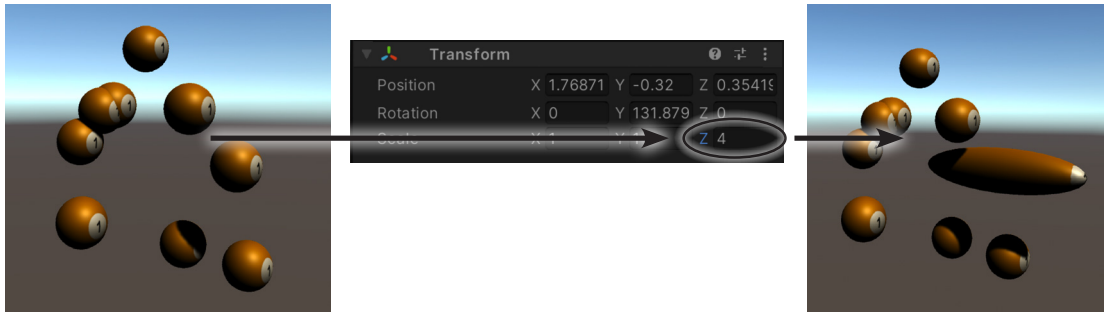
Figure out how to add a BallNumber field to your OneBallBehaviour script, so that when you click on a OneBall instance in the Hierarchy and check its One Ball Behaviour (Script) component, under the X Rotation, Y Rotation, Z Rotation, and Degrees Per Second labels it has a Ball Number field:

Ball Number 11

The first instance of OneBall should have its Ball Number field set to 1. The second instance should have it set to 2, the third 3, etc. *Here's a hint: you'll need a way to keep track of the count that's **shared by all of the OneBall instances**. You'll modify the Start method to increment it, then use it to set the BallNumber field.*

### Use the Inspector to work with GameObject instances

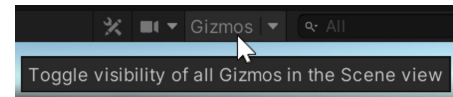
Run your game. Once a few balls have been instantiated, click the Pause button—the Unity editor will jump back to the Scene view. Click one of the OneBall instances in the Hierarchy window to select it. The Unity editor will outline it in the Scene window to show you which object you selected. Go to the Transform component in the Inspector window and **set its Z scale value to 4** to make the ball stretch.



Start your simulation again—now you can track which ball you’re modifying. Try changing its DegreesPerSecond, XRotation, YRotation, and ZRotation fields like you did in the last lab.

While the game is running, switch between the Game and Scene views. You can use the Gizmos in the Scene view **while the game is running**, even for GameObject instances that were created using the Instantiate method (rather than added to the Hierarchy window).

Try clicking the Gizmos button at the top of the toolbar to toggle them on and off. You can turn on the Gizmos in the Game view, and you can turn them off in the Scene view.



#### Exercise Solution

You can add a BallNumber field to the OneBallBehaviour script by keeping track of the total number of balls added so far in a static field (which we called BallCount). Each time a new ball is instantiated, Unity calls its Start method, so you can increment the static BallCount field and assign its value to that instance's BallNumber field.

```
static int BallCount = 0;
public int BallNumber;

void Start()
{
    transform.position = new Vector3(3 - Random.value * 6,
        3 - Random.value * 6, 3 - Random.value * 6);

    BallCount++;
    BallNumber = BallCount;
}
```

← All of the OneBall instances share a single static BallCount field, so the first instance's Start method increments it to 1, the second instance increments BallCount to 2, the third increments it to 3, etc.

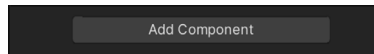


### Use physics to keep balls from overlapping

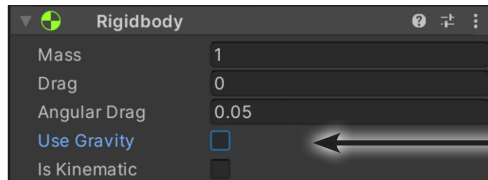
Did you notice that occasionally some of the balls overlap each other?

Unity has a powerful **physics engine** that you can use to make your GameObjects behave like they're real, solid bodies—and one thing that solid shapes don't do is overlap each other. To prevent that overlap, you just need to tell Unity that your OneBall prefab is a solid object.

Stop your game, then **click on the OneBall prefab in the Project window** to select it. Then go to the Inspector and scroll all the way down to the bottom to the Add Component button:



Click the button to pop up the Component window. **Choose Physics** to view the physics components, then **select Rigidbody** to add the component.

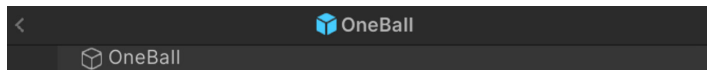


Make sure that you uncheck Use Gravity. Otherwise the balls will all react to gravity and start falling, and since there's nothing for them to hit, they'll keep falling forever.

Run your game again—now you won't see balls overlap. Occasionally one ball will get created on top of another one. When that happens, the new ball will knock the old one out of the way.

**Let's run a little physics experiment** to prove that the balls really are rigid now.

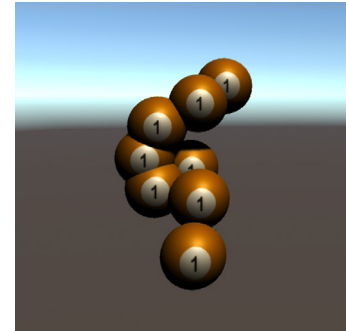
Start your game, then pause it as soon as there are more than two balls created. Go to the Hierarchy window. If it looks like this:



then you're editing the prefab—click the back caret (◀) in the top-right corner of the Hierarchy window to get back to the scene (you may need to expand SampleScene again).

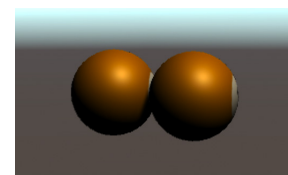
- ★ Hold down the Shift key, click the first OneBall instance in the Hierarchy window, and then click the second one so the first two OneBall instances are selected.
- ★ You'll see dashes (—) in the Position boxes in the Transform panel. **Set the Position to (0, 0, 0)** to set both OneBall instances' positions at the same time.
- ★ Use Shift-click to select any other instances of OneBall, right-click, and **choose Delete** to delete them from the scene so only the two overlapping balls are left.
- ★ Unpause your game—the balls can't overlap now, so instead they'll be rotating next to each other.

**Stop the game in Unity and Rider and save your scene. Save early, save often!**



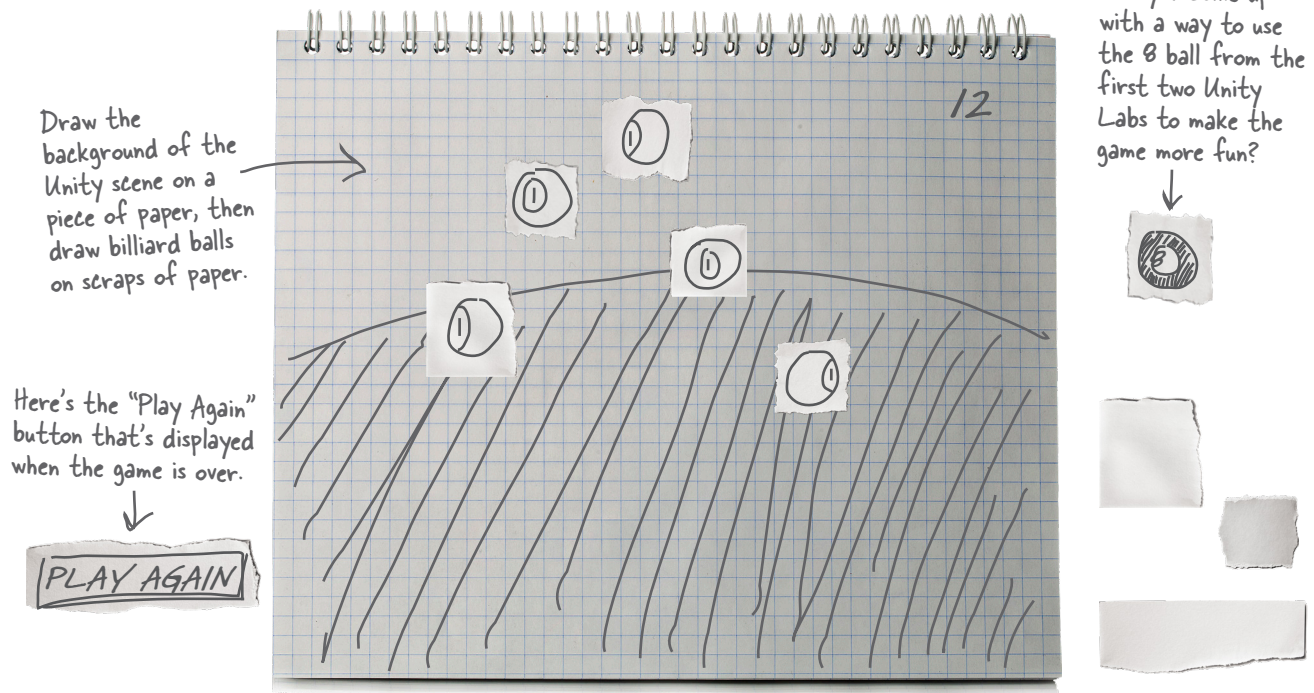
While you're running physics experiments, here's one Galileo would appreciate. Try checking the Use Gravity box while your game is running. New balls that get created will start falling, occasionally hitting another ball and knocking it out of the way.

You can use the Hierarchy window to delete GameObjects from your scene while the game is running.



### Get creative!

You're halfway done with the game! You'll finish it in the next Unity Lab. In the meantime, this is a great opportunity to practice your **paper prototyping** skills. We gave you a description of the game at the beginning of this Unity Lab. Try creating a paper prototype of the game. Can you come up with ways to make it more interesting?



### BULLET POINTS

- **Albedo** is a physics term that means the color that's reflected by an object. Unity can use texture maps for the albedo in a material.
- Unity has its own **Random class** in the UnityEngine namespace. The static Random.value method returns a random number between 0 and 1.
- A **prefab** is a GameObject that you can instantiate in your scene. You can turn any GameObject into a prefab.
- The **Instantiate method** creates a new instance of a GameObject. The Destroy method destroys it. Instances are created and destroyed at the end of the update loop.
- The **InvokeRepeating method** calls another method in the script over and over again.
- Unity calls every GameObject's Update method before each frame. That's called the **update loop**.
- You can **inspect the live instances** of your prefabs by clicking on them in the Hierarchy window.
- When you add a **Rigidbody** component to a GameObject, Unity's physics engine makes it act like a real, solid, physical object.
- The Rigidbody component lets you turn **gravity** on or off for a GameObject.

# Unity Lab #4

## User Interfaces

In the last Unity Lab you started to build a game, using a prefab to create `GameObject` instances that appear at random points in your game's 3D space and fly in circles. This Unity Lab picks up where the last one left off, allowing you to apply what you've learned about interfaces in C# and more.

Your program so far is an interesting visual simulation. The goal of this Unity Lab is to **finish building the game**. It starts off with a score of zero. Billiard balls will start to appear and fly around the screen. When the player clicks on a ball, the score goes up by 1 and the ball disappears. More and more balls appear; once 15 balls are flying around the screen, the game ends. For your game to work, your players need a way to start it and to play again once the game is over, and they'll want to see their score as they click on the balls. So you'll add a **user interface** that displays the score in the corner of the screen, and shows a button to start a new game.

## Add a score that goes up when the player clicks a ball

You've got a really interesting simulation. Now it's time to turn it into a game. **Add a new field** to the GameController class to keep track of the score—you can add it just below the OneBallPrefab field:

```
public int Score = 0;
```

Next, **add a method called ClickedOnBall to the GameController class**. This method will get called every time the player clicks on a ball:

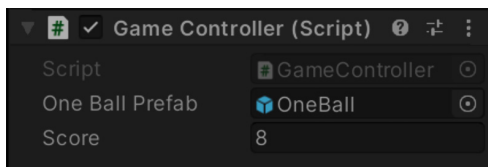
```
public void ClickedOnBall()
{
    Score++;
}
```

Unity makes it really easy for your GameObjects to respond to mouse clicks and other input. If you add a method called OnMouseDown to a script, Unity will call that method any time the GameObject it's attached to is clicked. **Add this method to the OneBallBehaviour class:**

```
void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    controller.ClickedOnBall();
    Destroy(gameObject);
}
```

The first line of the OnMouseDown method gets the instance of the GameController class, and the second line calls its ClickedOnBall method, which increments its Score field.

Now run your game. Click on Main Camera in the hierarchy and watch its Game Controller (Script) component in the Inspector. Click on some of the rotating balls—they'll disappear and the Score will go up.



### there are no Dumb Questions

**Q:** Why do we use Instantiate instead of the new keyword?

**A:** Instantiate and Destroy are *special methods that are unique to Unity*—you won't see them in your other C# projects. The Instantiate method isn't quite the same thing as the C# new keyword, because it's creating a new instance of a prefab, not a class. Unity does create new instances of objects, but it needs to do a lot of other things, like making sure that it's included in the update loop. When a GameObject's script calls Destroy(gameObject) it's telling Unity to destroy itself. The Destroy method tells Unity to destroy a GameObject—but not until after the update loop is complete.

**Q:** I'm not clear on how the first line of the OnMouseDown method works. What's going on there?

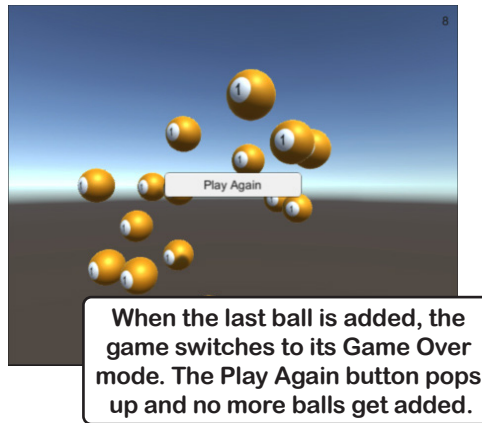
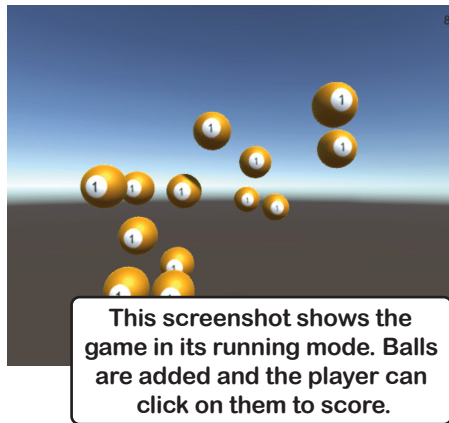
**A:** Let's break down that statement piece by piece. The first part should be pretty familiar: it declares a variable called `controller` of type `GameController`, the class that you defined in the script that you attached to the Main Camera. In the second half, we want to call a method on the GameController attached to the Main Camera. So we use `Camera.main` to get the Main Camera, and `GetComponent<GameController>()` to get the instance of GameController that we attached to it.

### Add two different modes to your game

Start up your favorite game. Do you immediately get dropped into the action? Probably not—you're probably looking at a start menu. Some games let you pause the action to look at a map. Many games let you switch between moving the player and working with an inventory, or show an animation while the player is dying that can't be interrupted. These are all examples of **game modes**.

Let's add two different modes to your billiard ball game:

- ★ **Mode #1: The game is running.** Balls are being added to the scene, and clicking on them makes them disappear and the score go up.
- ★ **Mode #2: The game is over.** Balls are no longer getting added to the scene, clicking on them doesn't do anything, and a "Game over" banner is displayed.



You'll add two modes to your game. You already have the "running" mode, so now you just need to add a "game over" mode.

Here's how you'll add the two game modes to your game:

- 1 **Make GameController.AddABall pay attention to the game mode.**  
Your new and improved AddABall method will check if the game is over, and will only instantiate a new OneBall prefab if the game is not over.
- 2 **Make OneBallBehaviour.OnMouseDown only work when the game is running.**  
When the game is over, we want the game to stop responding to mouse clicks. The player should just see the balls that were already added continue to circle until the game restarts.
- 3 **Make GameController.AddABall end the game when there are too many balls.**  
AddABall also increments its NumberOfBalls counter, so it goes up by 1 every time a ball is added. If the value reaches MaximumBalls, it sets GameOver to true to end the game.

In this lab, you're building this game in parts, and making changes along the way. You can download the code for each part from the book's GitHub repository: <https://github.com/head-first-csharp/fourth-edition>.

## Add game mode to your game

Modify your GameController and OneBallBehaviour classes to **add modes to your game** by using a Boolean field to keep track of whether or not the game is over.

### 1 *Make GameController.AddABall pay attention to the game mode.*

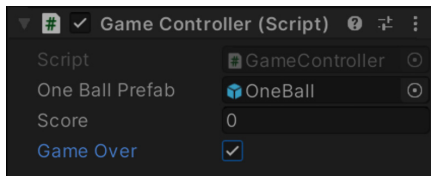
We want the GameController to know what mode the game is in. When we need to keep track of what an object knows, we use fields. Since there are two modes—running and game over—we can use a Boolean field to keep track of the mode. **Add the GameOver field** to your GameController class:

```
public bool GameOver = false;
```

The game should only add new balls to the scene if the game is running. Modify the AddABall method to add an **if** statement that only calls Instantiate if GameOver is not true:

```
public void AddABall()
{
    if (!GameOver)
    {
        Instantiate(OneBallPrefab);
    }
}
```

Now you can test it out. Start your game, then **click on Main Camera** in the Hierarchy window.



Check the Game Over box while the game is running to toggle the GameController's GameOver field. If you check it while the game is running, Unity will reset it when you stop the game.

Set the GameOver field by unchecking the box in the Script component. The game should stop adding balls until you check the box again.

### 2 *Make OneBallBehaviour.OnMouseDown only work when the game is running.*

Your OnMouseDown method already calls the GameController's ClickedOnBall method. Now **modify OnMouseDown in OneBallBehaviour** to use the GameController's GameOver field as well:

```
void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    if (!controller.GameOver)
    {
        controller.ClickedOnBall();
        Destroy(gameObject);
    }
}
```

Run your game again and test that balls disappear and the score goes up only when the game is not over.



## Unity Lab #4

### User Interfaces

3

**Make `GameController.AddABall` end the game when there are too many balls.**

The game needs to keep track of the number of balls in the scene. We'll do this by **adding two fields** to the `GameController` class to keep track of the current number of balls and the maximum number of balls:

```
public int NumberOfBalls = 0;
public int MaximumBalls = 15;
```

Every time the player clicks on a ball, the ball's `OneBallBehaviour` script calls `GameController.ClickedOnBall` to increment (add 1 to) the score. Let's also decrement (subtract 1 from) `NumberOfBalls`:

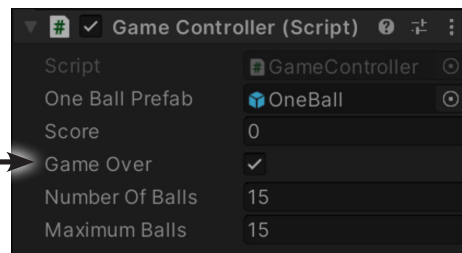
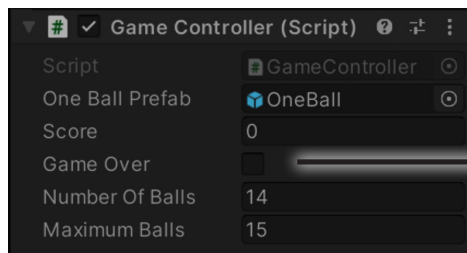
```
public void ClickedOnBall()
{
    Score++;
    NumberOfBalls--;
}
```

Now **modify the `AddABall`** method so that it only adds balls if the game is running, and ends the game if there are too many balls in the scene:

```
public void AddABall()
{
    if (!GameOver)
    {
        Instantiate(OneBallPrefab);
        NumberOfBalls++;
        if (NumberOfBalls >= MaximumBalls)
        {
            GameOver = true;
        }
    }
}
```

The `GameOver` field is true if the game is over and false if the game is running. The `NumberOfBalls` field keeps track of the number of balls currently in the scene. Once it hits the `MaximumBalls` value, the `GameController` will set `GameOver` to true.

Now test your game one more time by running it and then clicking on Main Camera in the Hierarchy window. The game should run normally, but as soon as the `NumberOfBalls` field is equal to the `MaximumBalls` field, the `AddABall` method sets its `GameOver` field to true and ends the game.



Once that happens, clicking on the balls doesn't do anything because `OneBallBehaviour.OnMouseDown` checks the `GameOver` field and only increments the score and destroys the ball if `GameOver` is false.

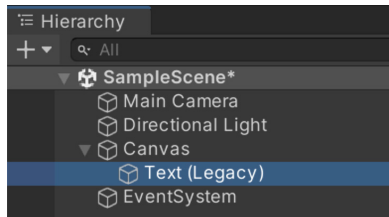
Your game needs to keep track of its game mode. Fields are a great way to do that.

### Add a UI to your game

Almost any game you can think of—from Pac Man to Super Mario Brothers to Grand Theft Auto 5 to Minecraft—features a **user interface (or UI)**. Some games, like Pac Man, have a very simple UI that just shows the score, high score, lives left, and current level. Many games feature an intricate UI incorporated into the game’s mechanics (like a weapon wheel that lets the player quickly switch between weapons). Let’s add a UI to your game.

**Choose *UI >> Legacy >> Text* from the **GameObject** menu** to add a 2D Text GameObject to your game’s UI.

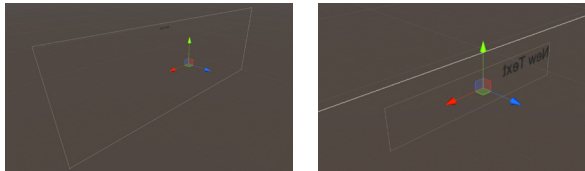
This adds a Canvas to the Hierarchy, and a Text under that Canvas:



*We’re using the legacy UI components because the Unity Labs in the book target a previous version of Unity, and we want to stay consistent with them.*

When you added the Text to your scene, Unity automatically added Canvas and Text GameObjects. Click the triangle (▼) next to Canvas to expand or collapse it—the Text GameObject will appear and disappear because it’s nested under Canvas.

Double-click on Canvas in the Hierarchy window to focus on it. It’s a 2-D rectangle. Click on its Move Gizmo and drag it around the scene. It won’t move! The Canvas that was just added will always be displayed, scaled to the size of the screen and in front of everything else in the game.

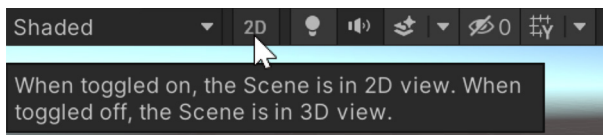


Did you notice an **EventSystem** in the Hierarchy? Unity automatically added it when you created the UI. It manages mouse, keyboard, and other inputs and sends them back to GameObjects—and it does all of that automatically, so you won’t need to work directly with it.

Then double-click on Text to focus on it—the editor will zoom in, but the default text (“New Text”) will be backward because the Main Camera is pointing at the back of the Canvas.

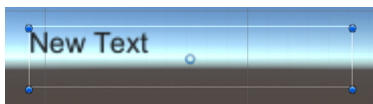
### Use the 2D view to work with the Canvas

The **2D button** at the top of the Scene window toggles 2D view on and off:



A **Canvas** is a two-dimensional GameObject that lets you lay out your game’s UI. Your game’s Canvas will have two GameObjects nested under it: the **Text** GameObject that you just added will be in the upper-right corner to display the score, and there’s a **Button** GameObject to let the player start a new game.

Click the 2D view—the editor flips around its view to shows the canvas head-on. **Double-click on Text** in the Hierarchy window to zoom in on it.



Use the mouse wheel to zoom in and out in 2D view.

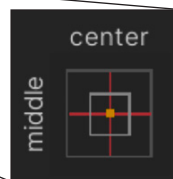
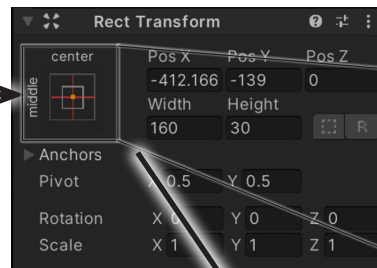
You can **click the 2D button to switch between 2D and 3D**. Click it again to return to the 3D view.

### Set up the Text that will display the score in the UI

Your game's UI will feature one Text GameObject and one Button. Each of those GameObjects will be **anchored** to a different part of the UI. For example, the Text GameObject that displays the score will show up in the upper-right corner of the screen (no matter how big or small the screen is).

Click on Text in the Hierarchy window to select it, then look at the Rect Transform component. We want the Text in the upper-right corner, so **click the Anchors box** in the Rect Transform panel.

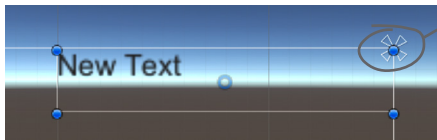
Since the Text only "lives" in the 2D Canvas, it uses a **Rect Transform** (it has that name because its position is relative to the rectangle that makes up the Canvas). Click on the Anchors box to display the anchor presets.



The Text is anchored to a specific point in the 2D Canvas.

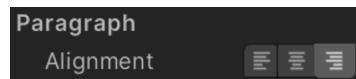
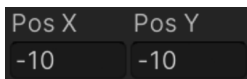
Make sure you hold down both Shift and Alt (Option on a Mac) so you set both the pivot and position.

The Anchor Presets window lets you anchor your UI GameObjects to various parts of the Canvas. **Hold down Alt and Shift** (or Option+Shift on a Mac) and **choose the top right anchor preset**. Click the same button you used to bring up the Anchor Presets window. The Text is now in the upper-right corner of the Canvas—double-click on it again to zoom into it.

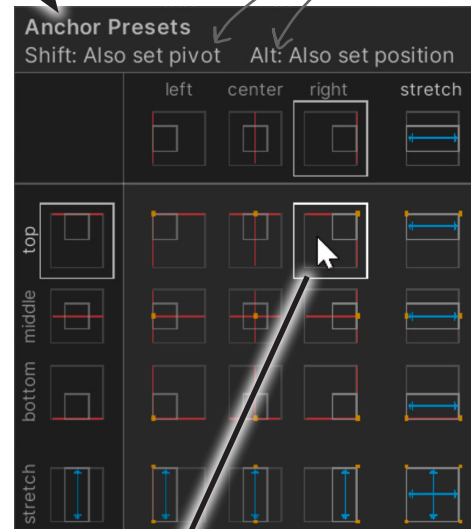


You set the anchor pivot to the top right. The Text's position is the anchor's position relative to the Canvas.

Let's add a little space above and to the right of the Text. Go back to the Rect Transform panel and **set both Pos X and Pos Y to -10** to position the text 10 units to the left and 10 down from the top-right corner. Then **set the Alignment on the Text component to right**, and use the box at the top of the Inspector to **change the GameObject's name to Score**.



Your new Text should now show up in the Hierarchy window with the name Score. It should now be right-aligned, with a small gap between the edge of the Text and the edge of the Canvas.



### Add a button that calls a method to start the game

When the game is in its “game over” mode, it will display a button labeled Play Again that calls a method to restart the game. **Add an empty StartGame method** to your GameController class (we’ll add its code later):

```
public void StartGame()
{
    // We'll add the code for this method later
}
```

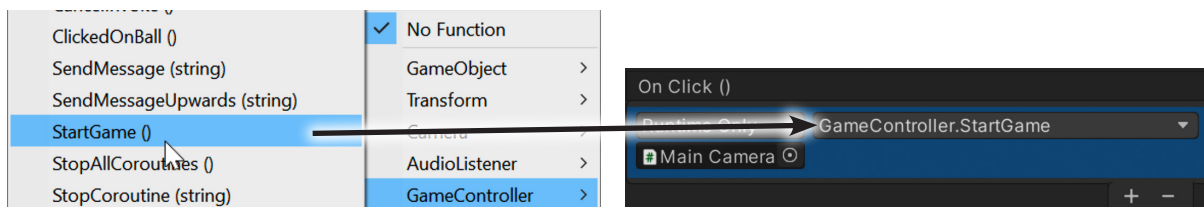
**Click on Canvas in the Hierarchy window** to focus on it. Then **choose UI >> Legacy >> Button** from the GameObject menu to add a Button. Since you’re already focused on the Canvas, the Unity editor will add the new Button and anchor it to the center of the Canvas. Did you notice that Button has a triangle next to it in the Hierarchy? Expand it—there’s a TextGameObject nested under it. Click on it and set its text to **Play Again**.



Now that the Button is set up, we just need to make it call the StartGame method on the GameController object attached to the Main Camera. A UI button is **just a GameObject with a Button component**, and you can use its On Click () box in the Inspector to hook it up to an event handler method. Click the **+** button at the bottom of the On Click () box to add an event handler, then **drag Main Camera onto the None (Object) box**.



Now the Button knows which GameObject to use for the event handler. Click the **No Function** dropdown and choose **GameController >> StartGame**. Now when the player presses the button, it will call the StartGame method on the GameController object hooked up to the Main Camera.



### Make the Play Again button and Score Text work

Your game's UI will work like this:

- ★ The game starts in the game over mode.
- ★ Clicking the Play Again button starts the game.
- ★ Text in the upper-right corner of the screen displays the current score.

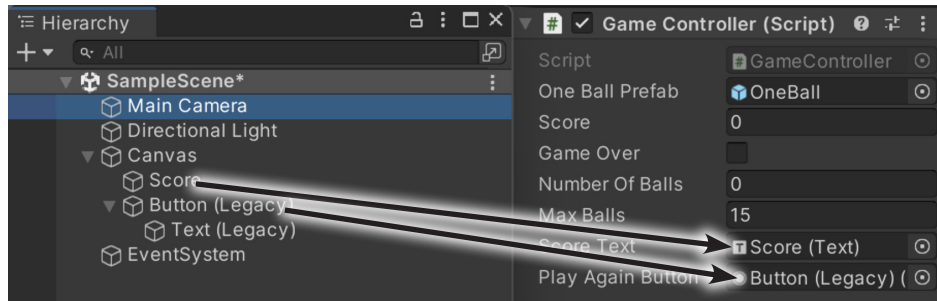
You'll be using the Text and Button classes in your code. They're in the `UnityEngine.UI` namespace, so **add this using statement** to the top of your `GameController` class:

```
using UnityEngine.UI;
```

Now you can add Text and Button fields to your `GameController` (just above the `OneBallPrefab` field):

```
public Text ScoreText;  
public Button PlayAgainButton;
```

**Click on Main Camera** in the Hierarchy window. **Drag the Text GameObject** out of the Hierarchy and **onto** the Score Text field in the Script component, **then drag the Button GameObject onto** the Play Again Button field.

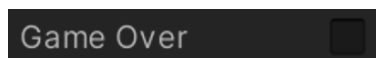


Go back to your `GameController` code and **set the GameController field's default value to true**:

```
public bool GameOver = true; ← Change this from false to true.
```

Now go back to Unity and check the Script component in the Inspector.

**Hold on, something's wrong!**



The Unity editor still shows the Game Over checkbox as unchecked—it didn't change the field value. Make sure to check the box so your game starts in the game over mode:



Now the game will start in the game over mode, and the player can click the Play Again button to start playing.



**Watch it!**

**Unity remembers your scripts' field values.**

When you wanted to change the `GameController.GameOver` field from false to true, it wasn't enough to change the code. When you add a Script component to Unity, it keeps track of the field values, and it won't reload the default values unless you reset it from the context menu (Ⓜ).

### Finish the code for the game

The GameController object attached to the Main Camera keeps track of the score in its Score field. **Add an Update method to the GameController class** to update the Score Text in the UI:

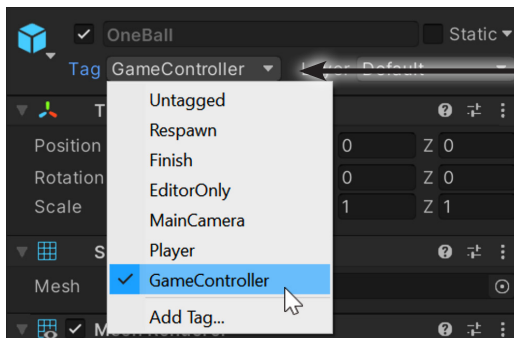
```
void Update()
{
    ScoreText.text = Score.ToString();
}
```

Next, **modify your GameController.AddABall method** to enable the Play Again button when it ends the game:

```
if (NumberOfBalls >= MaximumBalls)
{
    GameOver = true;
    PlayAgainButton.gameObject.SetActive(true);
}
```

Every GameObject has a property called `gameObject` that lets you manipulate it. You'll use its `SetActive` method to make the Play Again button visible or invisible.

There's just one more thing to do: get your StartGame method working so that it starts the game. It needs to do a few things: destroy any balls that are currently flying around the scene, disable the Play Again button, reset the score and number of balls, and set the mode to "running." You already know how to do most of those things! You just need to be able to find the balls in order to destroy them. **Click on the OneBall prefab in the Project window and set its tag:**



A **tag** is a keyword that you can attach to any of your GameObjects that you can use in your code when you need to identify them or find them. When you click on a prefab in the Project window and use this dropdown to assign a tag, that tag will be assigned to every instance of that prefab that you instantiate.

Now you have everything in place to fill in your StartGame method. It uses a **foreach** loop to find and destroy any balls left over from the previous game, hides the button, resets the score and number of balls, and changes the game mode:

```
public void StartGame()
{
    foreach (GameObject ball in GameObject.FindGameObjectsWithTag("GameController"))
    {
        Destroy(ball);
    }
    PlayAgainButton.gameObject.SetActive(false);
    Score = 0;
    NumberOfBalls = 0;
    GameOver = false;
}
```

Now run your game. It starts in "game over" mode. Press the button to start the game. The score goes up each time you click on a ball. As soon as the 15th ball is instantiated, the game ends and the Play Again button appears again.



## Unity Lab #4

### User Interfaces



#### Exercise

Here's a Unity coding challenge for you! Each of your GameObjects has a **transform.Translate** method that moves it a distance from its current position. The goal of this exercise is to modify your game so that instead of using **transform.RotateAround** to circle balls around the Y axis, your **OneBallBehaviour** script uses **transform.Translate** to make the balls fly randomly around the scene.

- **Remove** the **XRotation**, **YRotation**, and **ZRotation** fields from **OneBallBehaviour**. **Replace them with fields** to hold the X, Y, and Z speed called **XSpeed**, **YSpeed**, and **ZSpeed**. They're float fields—no need to set their values.

- **Replace all of the code in the Update method** with this line of code that calls the **transform.Translate** method:

```
transform.Translate(Time.deltaTime * XSpeed,  
                  Time.deltaTime * YSpeed, Time.deltaTime * ZSpeed);
```

The parameters represent the speed that the ball is traveling along the X, Y, or Z axis. So if **XSpeed** is 1.75, multiplying it by **Time.deltaTime** causes ball move along the X axis at a rate of 1.75 units per second.

- **Replace the DegreesPerSecond field** with a field called **Multiplier** with a value of 0.75E—the **F** is important! Use it to update the **XSpeed** field in the **Update** method, and **add two similar lines** for the **YSpeed** and **ZSpeed** fields:

```
XSpeed += Multiplier - Random.value * Multiplier * 2;
```

Part of this exercise is to **understand exactly how this line of code works**. **Random.value** is a static method that returns a random floating-point number between 0 and 1. What is this line of code doing to the **XSpeed** field?

.....

.....

.....

- Then **add a method called ResetBall** and call it from the Start method. Add this line of code to **ResetBall**:

```
XSpeed = Multiplier - Random.value * Multiplier * 2;
```

What does that line of code do?

Before you start working on the game,  
figure out what these lines of code do.

.....

.....

**Add two more lines** just like it to **ResetBall** that update **YSpeed** and **ZSpeed**. Then **move the line of code** that updates **transform.position** out of the Start method and into the **ResetBall** method.

- Modify the **OneBallBehaviour** class to **add a field called TooFar** and set it to 5. Then modify the **Update** method to check whether the ball went too far. You can check if a ball went too far along the X axis like this:

```
Mathf.Abs(transform.position.x) > TooFar
```

That checks the *absolute value* of the X position, which means that it will check if **transform.position.x** is greater than 5F or less than -5F. Here's an **if** statement that checks if the ball went too far along the X, Y, or Z axis:

```
if ((Mathf.Abs(transform.position.x) > TooFar)  
    || (Mathf.Abs(transform.position.y) > TooFar)  
    || (Mathf.Abs(transform.position.z) > TooFar)) {
```

**Modify your OneBallBehaviour.Update method** to use that **if** statement to call **ResetBall** if the ball went too far.



### Exercise Solution

Here's what the entire OneBallBehaviour class looks like after updating it following the instructions in the exercise. The key to how this game works is that each ball's speed along the X, Y, and Z axes is determined by its current XSpeed, YSpeed, and ZSpeed values. By making small changes to those values, you've made your ball move randomly throughout the scene.

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

```
public class OneBallBehaviour : MonoBehaviour
{
```

```
    public float XSpeed;
    public float YSpeed;
    public float ZSpeed;
    public float Multiplier = 0.75F;
    public float TooFar = 5;
```

You added these fields to your OneBallBehaviour class. Don't forget to add the F to 0.75F, otherwise your code won't build.

```
    static int BallCount = 0;
    public int BallNumber;
```

```
    // Start is called before the first frame update
    void Start()
```

```
    {
        BallCount++;
        BallNumber = BallCount;

        ResetBall();
    }
```

← When the ball is first instantiated, its Start method calls ResetBall to give it a random position and speed.

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

```
    {
        transform.Translate(Time.deltaTime * XSpeed,
                            Time.deltaTime * YSpeed, Time.deltaTime * ZSpeed);

        XSpeed += Multiplier - Random.value * Multiplier * 2;
        YSpeed += Multiplier - Random.value * Multiplier * 2;
        ZSpeed += Multiplier - Random.value * Multiplier * 2;

        if ((Mathf.Abs(transform.position.x) > TooFar)
            || (Mathf.Abs(transform.position.y) > TooFar)
            || (Mathf.Abs(transform.position.z) > TooFar))
        {
            ResetBall();
        }
    }
```

↖ The Update method first moves the ball, then updates the speed, and finally checks if it went out of bounds. It's OK if you did these things in a different order.



#### Exercise Solution

```
void ResetBall()
{
    XSpeed = Random.value * Multiplier;
    YSpeed = Random.value * Multiplier;
    ZSpeed = Random.value * Multiplier;

    transform.position = new Vector3(3 - Random.value * 6,
    3 - Random.value * 6, 3 - Random.value * 6);
}

void OnMouseDown()
{
    GameController controller = Camera.main.GetComponent<GameController>();
    if (!controller.GameOver)
    {
        controller.ClickedOnBall();
        Destroy(gameObject);
    }
}
```

We reset the ball when it's first instantiated or if it flies out of bounds by giving it a random speed and position. It's OK if you set the position first.

Here are our answers to the questions—did you come up with similar answers?

```
XSpeed += Multiplier - Random.value * Multiplier * 2;
```

What is this line of code doing to the XSpeed field?

Random.value \* Multiplier \* 2 finds a random number between 0 and 1.5. Subtracting that from Multiplier gives us a random number between -0.75 and 0.75. Adding that value to XSpeed causes it to either speed up or slow down a small amount for each frame.

By increasing or decreasing the ball's speed along all three axes, we're giving each ball a wobbly random path.

```
XSpeed = Multiplier - Random.value * Multiplier * 2;
```

What does that line of code do?

It sets the XSpeed field to a random value between -0.75 and 0.75. This causes some balls to start going forward along the X axis and others to go backward, all at different speeds.

**Did you notice that you *didn't* have to make any changes to the GameController class? That's because you didn't make changes to the things that GameController does, like managing the UI or the game mode. If you can make a change by modifying one class but not touching others, that can be a sign that you designed your classes well.**

### Get creative!

Can you find ways to improve your game and get practice writing code? Here are some ideas:

- ★ Is the game too easy? Too hard? Try changing the parameters that you pass to `InvokeRepeating` in your `GameController.Start` method. Try making them fields. Play around with the `MaximumBalls` value, too. Small changes in these values can make a big difference in gameplay.
- ★ We gave you texture maps for all of the billiard balls. Try adding different balls that have different behaviors. Use the scale to make some balls bigger or smaller, and change their parameters to make them go faster or slower, or move differently.
- ★ Can you figure out how to make a “shooting star” ball that flies off really quickly in one direction and is worth a lot if the player clicks on it? How about making a “sudden death” 8 ball that immediately ends the game?
- ★ Modify your `GameController.ClickedOnBall` method to take a score parameter instead of incrementing the `Score` field and add the value that you pass. Try giving different values to different balls.

*If you change fields in the `OneBallBehaviour` script, don't forget to reset the `Script` component of the `OneBall` prefab! Otherwise, it will remember the old values.*

The more practice you get writing C# code, the easier it will get. Getting creative with your game is a great opportunity to get some practice!

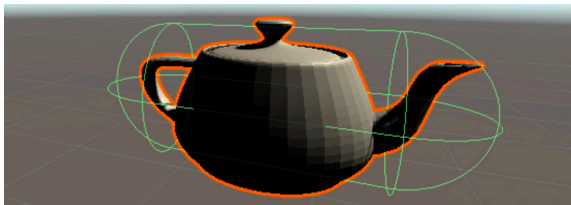
### BULLET POINTS

- Unity games display a **user interface (UI)** with controls and graphics on a flat, two-dimensional plane in front of the game's 3D scene.
- Unity provides a set of **2D UI GameObjects** specifically made for building user interfaces.
- A **Canvas** is a 2D `GameObject` that lets you lay out your game's UI. UI components like `Text` and `Button` are nested under a `Canvas` `GameObject`.
- The **2D button** at the top of the Scene window toggles 2D view on and off, which makes it easier to lay out a UI.
- When you add a **Script component** to Unity, it keeps track of the field values, and it won't reload the default values unless 2D reset it from the context menu.
- A **Button** can call any method in a script that's attached to a `GameObject`.
- You can use the Inspector to **modify field values** in your `GameObject`'s scripts. If you modify them while the game is running, they'll reset to saved values when it stops.
- The **`transform.Translate`** method moves a `GameObject` a distance from its current position.
- A **tag** is a keyword that you can attach to any of your `GameObject`s that you can use in your code when you need to identify them or find them.
- The **`GameObject.FindGameObjectsWithTag`** method returns a collection of `GameObject`s that match a given tag.

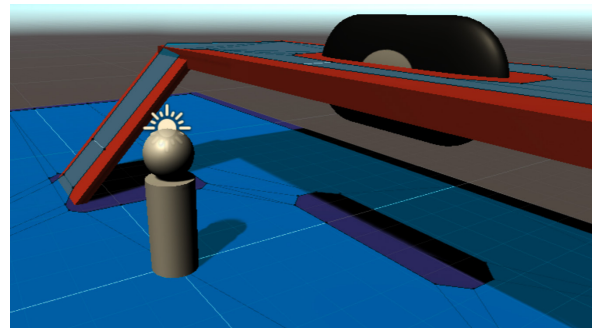
# Hungry for more Unity?

Here are just a few things you'll learn in the next **Head First C# Unity Labs!**

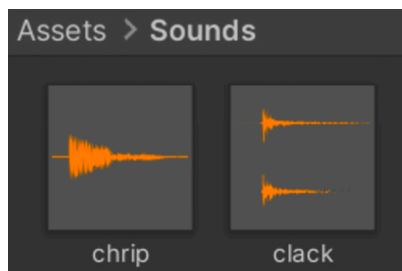
Make your `GameObjects` look like and act like real, complex physical objects.



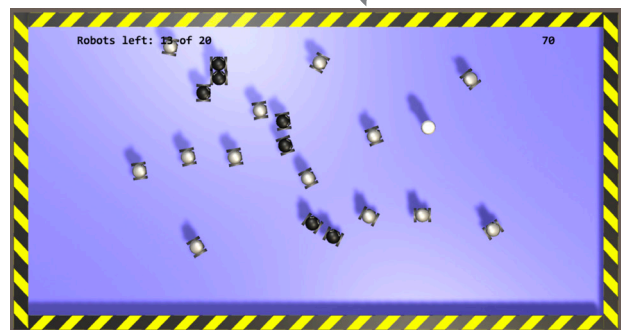
Use Unity's sophisticated AI-driven pathfinding and navigation system so your `GameObjects` can move on their own around your Unity scenes.



Add sound effects to your games!



Create a fun Unity version of a classic video game.



Download all of the **Head First C#** Unity Lab PDFs for free from our GitHub page today:

<https://github.com/head-first-csharp/fourth-edition>