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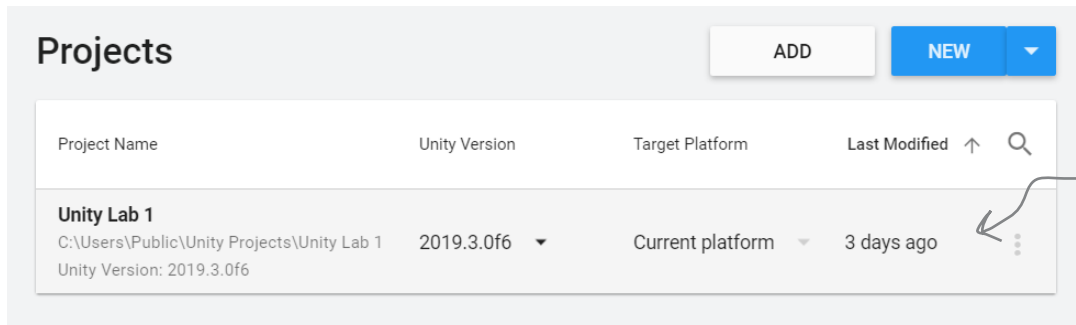
This Unity Lab picks up where the first one left off, so go to Unity Hub and open the project you created in the last lab.

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 Visual Studio to edit the script.** Remember how you set the Unity editor's preferences to make Visual Studio the script editor? That means you can just double-click on the script in the Unity editor and it will open up in Visual Studio.

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.
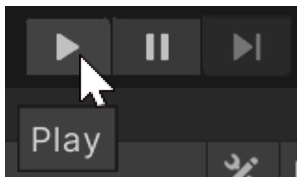

The Play button does not save your game! So make sure you save early and save often. A lot of people get in the habit of saving the scene every time they run the game.

4 **Use Unity and Visual Studio together to debug your script.** You've already seen how valuable the Visual Studio debugger is when you're trying to track down problems in your C# code. Unity and Visual Studio work together seamlessly so you can add breakpoints, use the Locals window, and work with the other familiar tools in the Visual Studio 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.
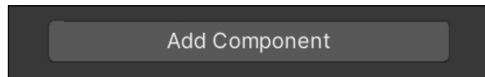
When you click it, Unity pops up a window with all of the different kinds of components that you can add—and there are *a lot* of them. **Choose "New script"** to add a new C# script to your Sphere GameObject. You'll be prompted for a name. **Name your script `BallBehaviour`**.
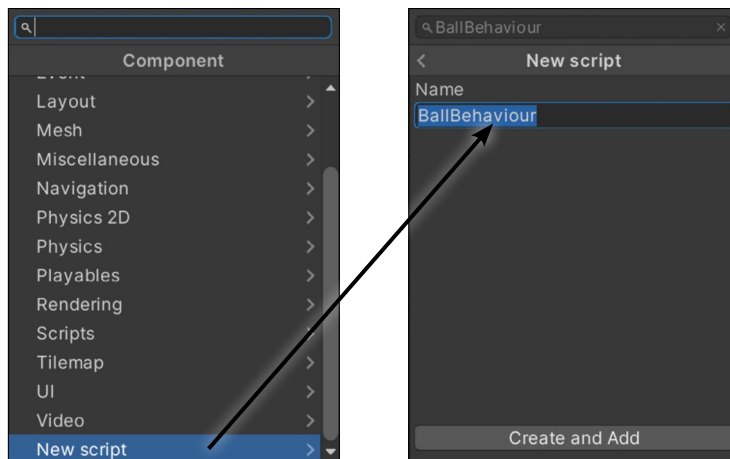

Unity code uses British spelling.

*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.
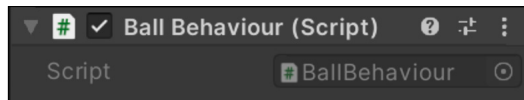
You'll also see the C# script in the Project window.
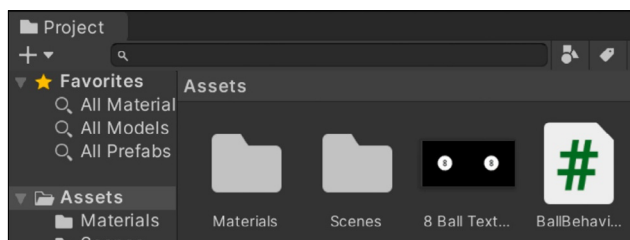
The Project window gives you a folder-based view of your project. Your Unity project is made up of files: media files, data files, C# scripts, textures, and more. Unity calls these files assets. The Project window was displaying a folder called Assets when you right-clicked inside it to import your texture, 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 Visual Studio 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 Visual Studio*. 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 Visual Studio 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 Visual Studio and open your C# script in it, go back to the beginning of Unity Lab I 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 to start your game:

Click the Play button

> Your game will start, and the 8 ball will start spinning at a speed of two rotations per second.

> If you don't see the Hierarchy window, reset the layout to Wide (click the Game tab to switch back to the Game view).

> Click on Sphere in the Hierarchy window to select it, then watch the Inspector window to see the Y rotation change in its Transform component.

> **Press the Play button again** to stop your game. Use the Play button to start and stop your game any time you want.

Your Code Up Close

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

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.

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.

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.
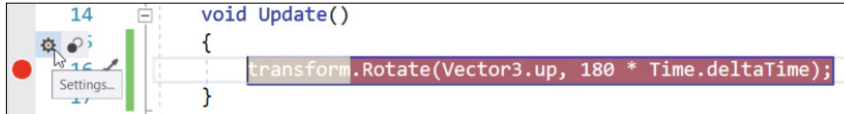
That's where the Time.deltaTime value comes in handy. Every time the Unity engine calls a GameObject's Update method—once per frame— it sets Time.deltaTime to the fraction of a second since the last frame. Since we want our ball to do a full rotation every two seconds, or 180 degrees per second, all 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.

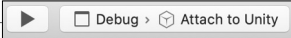Inside your Update method, multiplying any value by Time.deltaTime turns it into that value per second.

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 Visual Studio, and **add a breakpoint** on the line that you added to the Update method.

```
    14          void Update()
       ⚙ ○      {
    16              transform.Rotate(Vector3.up, 180 * Time.deltaTime);
       Settings...
                }
```

Now find the button at the top of Visual Studio that starts the debugger:

★ In Windows it looks like this—▶ Attach to Unity ▾ —or choose Debug >> Start Debugging (F5) from the
menu

★ In macOS it looks like this—▶ □ Debug › ⊙ Attach to Unity —or choose Run >> Start Debugging (⌘↵)

Click that button to **start the debugger**. Now switch back to the Unity editor. If this is the first time you're
debugging this project, the Unity editor will pop up a dialog window with these buttons:

Enable debugging for this session   Enable debugging for all projects   Cancel
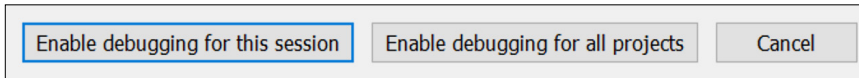
Press the "Enable debugging for this session" button (or if you want to keep that pop-up from appearing again,
press "Enable debugging for all projects"). Visual Studio is now *attached* to Unity, which means it can debug
your game.

Now **press the Play button in Unity** to start your game. Since Visual Studio 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:

★ On Windows, right-click on the breakpoint dot (🔴) at the left side of the line, choose **Conditions** from
the pop-up menu, select *Hit Count* and *Is a multiple of* from the dropdowns, and enter 500 in the box:

☑ Conditions
    Hit Count ▾   Is a multiple of ▾   500
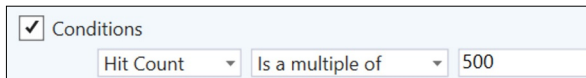
★ On macOS, right-click on the breakpoint dot (◎), choose **Edit breakpoint…** from the menu, then
choose *When hit count is a multiple of* from the dropdown and enter 500 in the box:

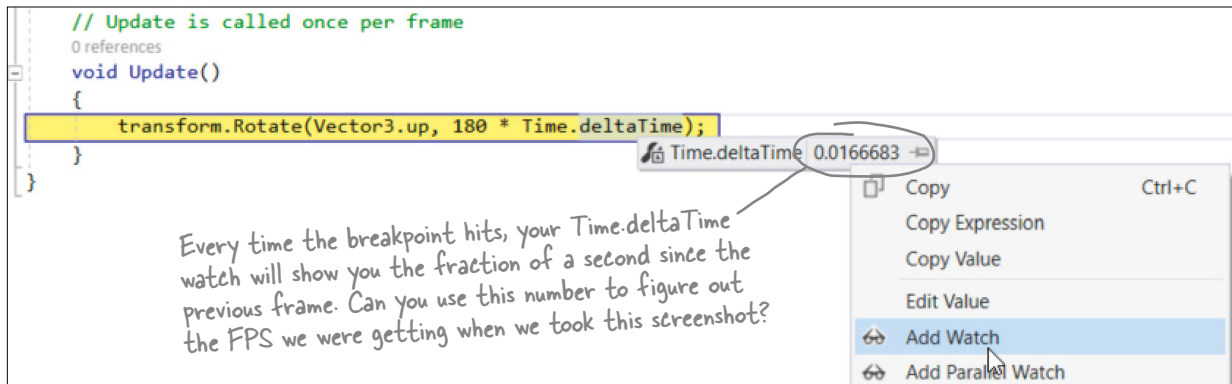When hit count is a multiple of   ▾   500 ▴▾

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
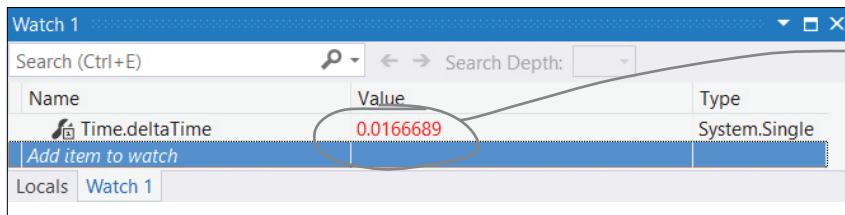the breakpoint breaks.

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 Visual Studio, **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 selecting Time.deltaTime and choosing Add Watch from the right-mouse menu.

```
      // Update is called once per frame
      0 references
      void Update()
      {
          transform.Rotate(Vector3.up, 180 * Time.deltaTime);
      }
  }
```

                                                    🔧 Time.deltaTime 0.0166683 ⬛

                                                    📋 Copy   Ctrl+C
                                                        Copy Expression
                                                        Copy Value

                                                        Edit Value
                                                    👓 Add Watch
                                                    👓 Add Parallel Watch

Every time the breakpoint hits, your Time.deltaTime watch will show you the fraction of a second since the previous frame. Can you use this number to figure out the FPS we were getting when we took this screenshot?

Continue debugging (F5 on Windows, ⇧⌘↵ on macOS), just like with the other apps you've debugged), 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.

Watch 1			▾ ☐ ✕
Search (Ctrl+E)	🔍 ▾ ← → Search Depth:	▾	
Name	Value		Type
🔧 Time.deltaTime	0.0166689		System.Single
Add item to watch			
Locals Watch 1			

Press the Continue button to get another Time.deltaTime value, then another. You can get your approximate FPS by dividing 1 ÷ Time.deltaTime.

Stop debugging (Shift+F5 on Windows, ⇧⌘↵ on macOS) to stop your program. Then **start debugging again**. Since your game is still running, the breakpoint will continue to work when you reattach Visual Studio to Unity. Once you're done debugging, **toggle your breakpoint again** so the IDE 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. Visual Studio will stay attached to Unity even when the game is stopped.

⚛ BRAIN POWER

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 toolbar: either click on it (⬚) or press the R key. You should see the Scale Gizmo appear on your cylinder:
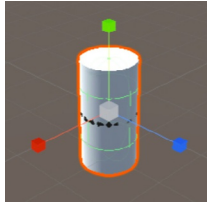

> **The Scale Gizmo looks a lot like the Move Gizmo, except that it has cubes instead of cones at the end of each axis. Your new cylinder is sitting on top of the sphere—you might see just a little of the sphere showing through the middle of the cylinder. When you make the cylinder narrower by changing 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.

Transform					
Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	0.2175312	Y	6.331524	Z	0.2783782
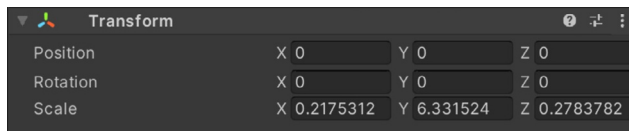
Click on the X label in the Scale row in the Transform panel and drag up and down. Make sure you click the actual X label to the left of the input box with the number. When you click the label it turns blue, and a blue box appears around the X value. As you drag your mouse up and down, the number in 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.

Transform					
Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	0.1	Y	20	Z	0.1
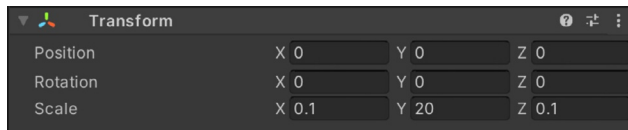
Now your sphere has a very long cylinder going through it that shows the Y axis where Y = 0.
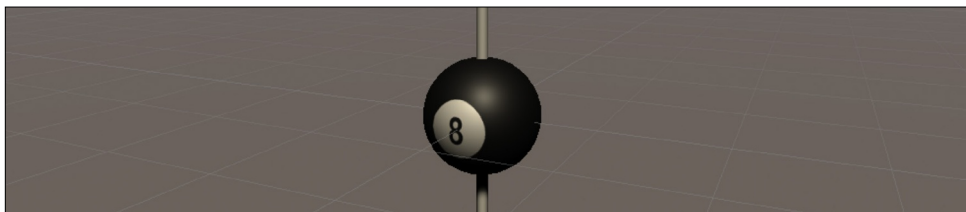
Add fields to your class for the rotation angle and speed

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.
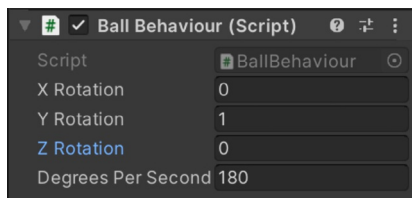
> When you add public fields to a class in your Unity script, the Script component displays input boxes that let you modify those fields. If you modify them while the game is <u>not</u> running, the updated values will get saved with your scene. You can also modify them while the game is running, but 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. <u>Stop your game</u>—and the field will <u>reset to 180</u>.

<u>While the game is stopped</u>, use the Unity editor to change the X Rotation to 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.
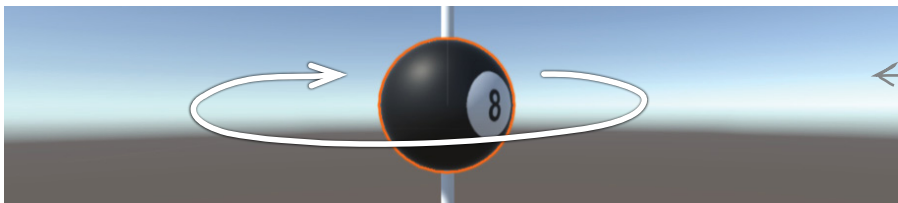
> When you use the Unity editor to set the Y Rotation field to 1 and then start your game, the ball rotates clockwise around the Y axis.

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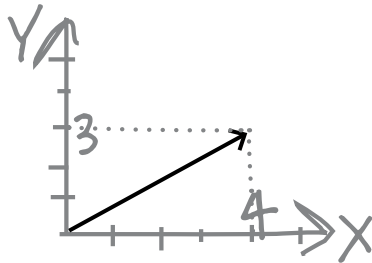
> Here's a diagram of a two-dimensional vector. You can represent it with two numbers: its value on the X axis (4) and its value on the Y axis (3), which you'd typically write as (4, 3).

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 <u>Scene view</u>

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 <u>Scene</u> view**. You may also need to **reset the Wide layout** by choosing Wide from the Layout dropdown.
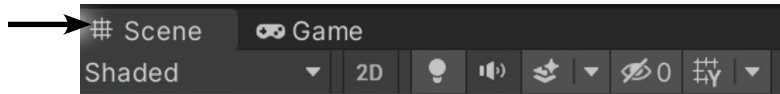
Now you're back in the familiar Scene view. Do these things to get a real sense of how 3D vectors work:

★ 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).
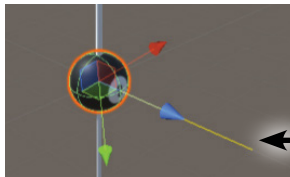
> The vector (0, 0, 3) extends 3 units along the Z axis. Look closely at the grid in the Unity editor—the vector is exactly 3 units long. Try clicking and dragging the Z Rotation label in the Script component in the Inspector. The ray will get larger or smaller as you drag. When the Z value in the vector 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.
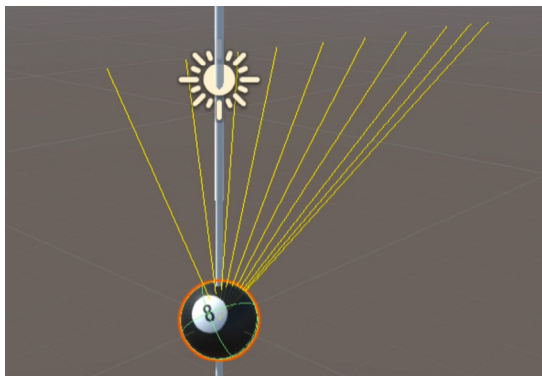
Making your ray leave a trail is a good way to help you develop an intuitive sense of how 3D vectors work.

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.

Changing the X position to 5 causes the ball to rotate in place away from the center of the scene.
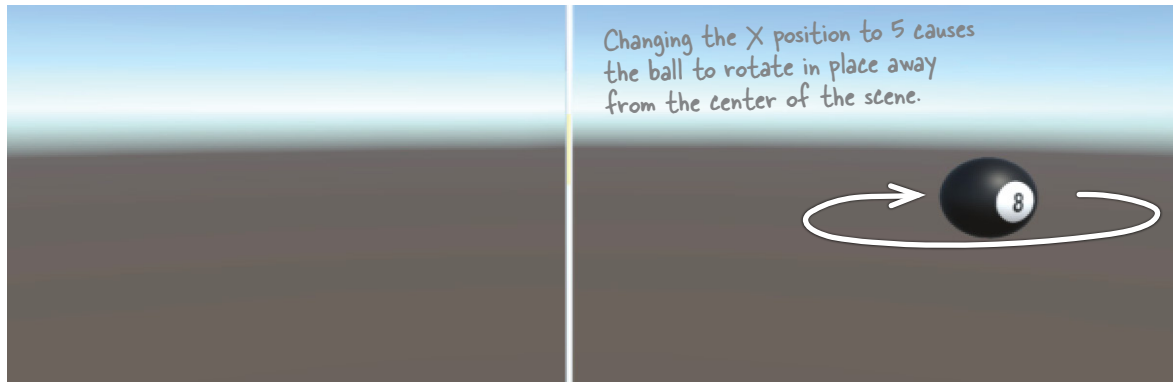
Let's modify the Update method to use a different kind of rotation. Now we'll make the ball rotate around the center point of the scene, coordinate (0, 0, 0), using the **transform.RotateAround method**, which rotates a GameObject around a point in the scene. (This is *different* from the transform.Rotate 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:

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

```
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);
}
```

Now run your code. This time it rotates the ball in a big circle around the center point:
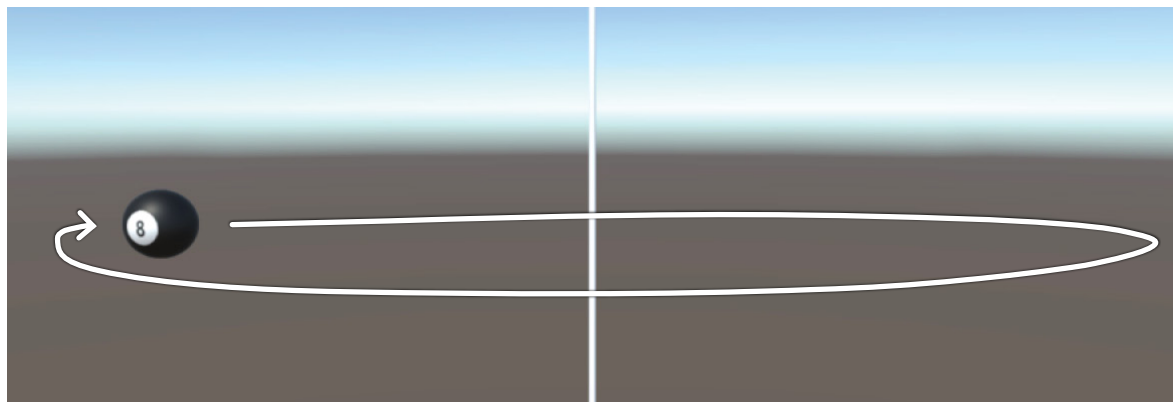
Use Unity to take a closer look at rotation and vectors

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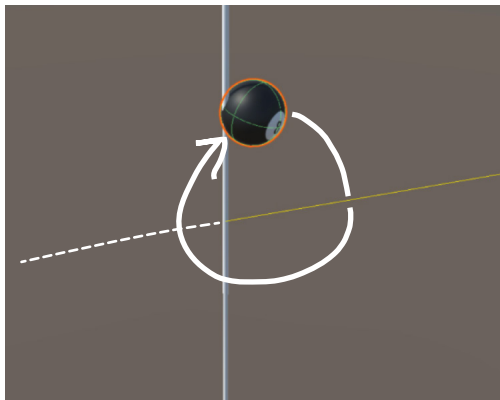
While the game is running, set the X, Y, and Z Rotation fields in the BallBehaviour Script component to 10, reset the sphere's Transform component, and change its X position to 2—as soon as you do, it starts rotating around the ray.

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 Visual Studio 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 Visual Studio 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 <u>not</u> 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.