O'REILLY®

Fourth Edition

Head First C#

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

Andrew Stellman
& Jennifer Greene

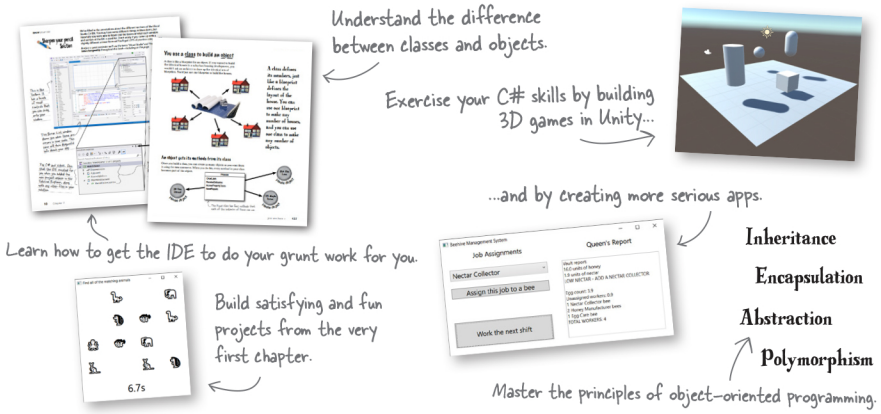
A Brain-Friendly Guide

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!



"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

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.

.NET

US $64.99 **CAN $85.99**

ISBN: 978-1-491-97670-8

9 781491 976708 56499

O'REILLY®

Head First C#

Fourth Edition

Andrew Stellman
Jennifer Greene

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First C#

Fourth Edition

by Andrew Stellman and Jennifer Greene

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*.

Series Creators:	Kathy Sierra, Bert Bates
Cover Designer:	Ellie Volckhausen
Brain Image on Spine:	Eric Freeman
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

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

ISBN: 978-1-491-97670-8

[LSI] [2020-11-13]

Unity Lab

Physics

In the previous Unity Lab projects, you had two different ways to move GameObjects around a scene. You used their transform properties and methods to translate them, rotate them, and set their positions, and you used Unity's navigation and pathfinding system to move a player automatically.

In the next two Unity Labs, you'll create a game where the player scores points by moving around the floor and running into targets—we'll use our familiar billiard balls as targets. But this time you'll use Unity's powerful **physics engine** to roll them around the scene by adding random forces to the balls.

In this lab, you'll learn the **basics of Unity physics**. You'll add **gravity** to billiard balls and watch them drop. Then you'll apply **forces** to those balls to get them to roll around. And along the way, you'll learn about the **kinematics** that drive Unity's physics.

Create a scene with a simple play area

After the last six Unity Labs, you should know exactly where to start. **Create a new Unity project** (give it a name like *Unity Labs 7 and 8*). Here are the GameObjects for the play area—notice that we included the camera in this list so it starts out at an angle that shows the whole floor. **Create these GameObjects**:

Name	Type	Position	Rotation	Scale
Main Camera	Camera	(0, 15, -9)	(65, 0, 0)	(1, 1, 1)
Floor	Cube	(0, -0.5, 0)	(0, 0, 0)	(40, 1, 20)
North Wall	Cube	(0, 1, 10.5)	(0, 0, 0)	(40, 0.75, 1)
South Wall	Cube	(0, 1, -10.5)	(0, 0, 0)	(40, 0.75, 1)
East Wall	Cube	(20.5, 0.5, 0)	(0, 90, 0)	(22, 0.75, 1)
West Wall	Cube	(-20.5, 0.5, 0)	(0, 90, 0)	(22, 0.75, 1)
▼ Player	Cylinder	(0, 1, 0)	(0, 0, 0)	(1, 1, 1)
Head	Sphere	(0, 1.5, 0)	(0, 0, 0)	(1, 1, 1)

Create the scene more quickly by duplicating GameObjects. Once you've created the North Wall GameObject, right-click on it in the Hierarchy window and choose "Duplicate" from the menu to make a copy of it.

Make sure Head is nested under Player, just like in the last Lab.

Add tiled a material

Create a Materials folder, **add a material called Floor Material** with albedo color 4E51CB, and drag it onto the Floor GameObject so it looks exactly like the floor in the last Unity Lab.

We want the walls to have black and yellow stripes. You can download a texture with a single stripe from GitHub: https://github.com/head-first-csharp/fourth-edition/tree/master/Unity_Labs/Texture_Maps

Download the file named `Yellow Stripe.png` and drag it into your Materials folder. We'll use this as the texture map. Next, **create a material called *Wall Material*** and drag the yellow stripe map into the box next to Albedo, just like you did with the billiard balls. Now **drag *Wall Material* onto each wall**.

Something doesn't look right. We want a bunch of little black and yellow stripes, but each of the walls just has one big stripe. Fix that by setting the **Tiling X** value in the *Wall Material* to 10. This makes the material

But something about those stripes still looks a little off. The shorter east and west walls look fine, but the stripes look stretched out on the longer north and south walls. The reason is that setting the Tiling value to 10 causes Unity to **repeat the texture 10 times** on each face of the shape. The north and south walls are about twice as long as the east and west walls, so their stripes will be about twice as wide.

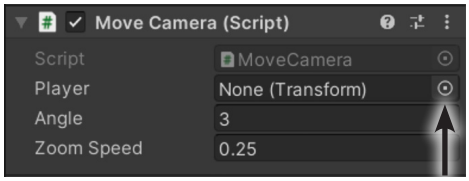
Let's do a very simple fix. **Create another material called *Wall Material 20*** that's identical to Wall Material, except that Tiling X is set to 20. Drag this new material onto the North and South walls. Now their stripes should look just like the stripes on the East and West walls.

In the first six Unity Labs, you used prefabs to instantiate GameObjects, and you built scripts to move the camera around the scene and the player around a NavMesh. Let's do a quick review of things that you've learned so far by creating a script that drops billiard balls onto the floor, a script to move the camera so it follows the player, and a script to move the player around the scene.

- **Create folders.** Use the Project window to create two more folders under Assets called Scripts and and Prefabs.
- **Create a prefab called OneBall Prefab.** This is identical to the prefab that you created in the second Unity Lab (including new a material for it), with one difference. In that Lab, you unchecked a box to keep the ball from reacting to gravity. This time, we want the balls to react to gravity, so make sure you keep "Use Gravity" checked.
- **Create a new GameController script that drops a OneBall from a random point every second.** This is very similar to the way you created instances of the ball in the third Unity Lab ("3. GameObject Instances"). Drop them all from a height of 10 units, and make sure it starts dropping balls as soon as the game starts. Make the height and repeat rate public fields so you can modify them in the Inspector window.
- **Add the MoveCamera script and drag it onto the camera.** Right-click on the Scripts folder in the Project window and choose Create >> C# Script to create the new script like you did in the last Lab... or you can take a shortcut! Open the folder for the last Unity Lab in Explorer (or Finder on a Mac). Go to the `Assets/Scripts` folder, then drag MoveCamera.cs directly into the Scripts folder in the Project window. Now you can drag it out of the Project window and onto the Main Camera in the Hierarchy window. You'll need to set the Player field.
- **Add the MoveToClick script and drag it onto the player.** Add the MoveToClick script from the last project and drag it onto the Player. You'll also need to **add a NavMesh Agent** to the Player. And you'll need to **add a NavMesh** to the scene where the entire Floor cube is Navigation Static and Walkable, just like you did in the last Unity Lab.

And here are a few useful tips to help get you started. (And don't forget—*it's okay to peek at the solution!*)

- **Watch the bottom of the Unity editor for errors.** Does the game start paused when you run it?  It could mean that there was a problem during one of your GameObjects' Start methods. Look at the status bar at the bottom of the Unity editor window for more details, it will often tell you exactly what went wrong.
- **Use the *Select GameObject* window to set GameObject fields.** In previous Unity Labs, to set a GameObject field in a script you dragged a GameObject from the Hierarchy window onto the field in the script. There's another way to do this—just **click the *Select GameObject* button** () to open a window to select a GameObject. Use the tabs at the top to switch between objects (like prefabs) in your Assets folder and GameObjects in the scene. Make sure you set the fields in both of the scripts attached to the camera.



- **Set Game view options to change the way the game is displayed.** When you click your Main Camera, is the Camera Preview a slightly different shape from the screenshots in the Unity Labs? It might be because we set the aspect ratio to 16:9. Take a minute to **read through the Game view page in the Unity manual** and experiment with the different settings. These settings will revert if you reset your layout to Wide—if you find a setting you like, you can save a new layout with your preferred settings by choosing *Save Layout...* from the layout dropdown.
- **There's another way to add a script to a GameObject.** Click the Add Component button in the Inspector window, choose New Script at the bottom of the list, and enter the script name. Unity will create the script file—but it will create the file inside the Assets folder, so make sure to drag it into the Scripts folder after you create it.

Exercise Solution

We did a quick review of things you learned in other Unity Labs, and in the process finished setting up the scene so we can run some physics experiments. Here's everything that we asked you to do:

Create a prefab called OneBall Prefab.

- Download `1 Ball Texture.png` from our GitHub project (just like you did with the other textures) and drag it into your Materials folder. **Create a new Material** called *1 Ball* with that image as the albedo map.
- Create a sphere called OneBall Prefab. Drag the new *1 Ball* material on it. **Add a Rigidbody component** to OneBall Prefab—and make sure the *Use Gravity* box is checked so the balls drop.
- **Drag OneBall Prefab** into the Prefabs folder. **Delete it** from the Hierarchy window.

Create a new GameController script that drops a OneBall from a random point every second.

- Here's the GameController script—make sure it's attached to Main Camera:

```
public class GameController : MonoBehaviour
{
    public float Height = 10F;
    public float RepeatRate = 1F;
    public GameObject Prefab;

    void Start()
    {
        InvokeRepeating("DropABall", 0F, RepeatRate);
    }

    private void DropABall()
    {
        GameObject ball = Instantiate(Prefab);
        ball.transform.position =
            new Vector3(10f - Random.value * 20f, 5f, 5f - Random.value * 10f);
    }
}
```

Try temporarily setting Height to 3 and RepeatRate to .01 and then run the game. It's not really part of this Lab, it just looks really interesting. Don't forget to set them back.

Did you use the Add Component button to add a C# script to a GameObject? Choose Scripts to choose from a list of scripts under your Assets folder.

- Use the Assets tab in the *Select GameObject* window to **set the Prefab field to *OneBall Prefab***. If you don't do that, you'll get this error when you start the game: *The variable Prefab of GameController has not been assigned.*

Add the MoveCamera script and drag it onto the Main Camera.

- **Copy the MoveCamera script** from the last Unity Lab into the Scripts folder.
- Use the Scene tab in the *Select GameObject* window to **set the Player field to the *Player* GameObject**. If you don't set the field, you'll get this error when you start the game: *UnassignedReferenceException: The variable Player of MoveCamera has not been assigned.*

Add the MoveToClick script and drag it onto the player

- **Copy the MoveToClick script** from the last Unity Lab into the Scripts folder.
- **Add a NavMesh Agent** to the Player GameObject. If you don't add one, you'll get this error when you start the game: *There is no 'NavMeshAgent' attached to the "Player" game object, but a script is trying to access it.*
- Use the AI >> Navigation window to **make the Floor walkable and bake the NavMesh**, just like in Unity Lab 6.

Now that your scene is all set up, let's run a few physics experiments.

Sharpen your pencil

We want the balls to roll around the play area so the player can catch them. We'll do that later using code, but for now, let's use the Inspector window to tilt the floor. Click on Main Camera in the Hierarchy and temporarily **set the repeat rate to 0** so it only drops a single ball, then **Start the game**.

1. **While the game is running**, click on Floor in the Hierarchy window to show its components. Then in the Inspector window, **rotate the floor along the X-axis** by clicking on the X label in the Rotation row and dragging your mouse up and down. What happened?

.....

.....

.....

2. **Stop the game**. Make sure Floor is still selected in the Hierarchy window, then **add a Rigidbody component** to the floor (but don't change any of its settings). Start your game again. What happened?

.....

.....

.....

3. **Stop your game**. **Uncheck the "Use Gravity" box** in your Floor GameObject's Rigidbody component. Start your game again. What happened?

.....

.....

.....

4. **Stop your game**. **Check the "Is Kinematic" box** in your Floor GameObject's Rigidbody component. Start your game again. As the game is running, **drag your mouse** over the X, Y, and Z Rotation labels in the Floor GameObject's Transform component to **tilt the floor**. What happened?

.....

.....

.....

Once you're done running your experiments, think about how you want the game to behave. Can you figure out what you need to do to make the play area act the way you'd expect it to?

Sharpen your pencil

We want the balls to roll around the play area so the player can catch them. We'll do that later using code, but for now, let's use the Inspector window to tilt the floor. Click on Main Camera in the Hierarchy and temporarily **set the repeat rate to 0** so it only drops a single ball, then **Start the game**.

1. **While the game is running**, click on Floor in the Hierarchy window to show its components. Then in the Inspector window, **rotate the floor along the X-axis** by clicking on the X label in the Rotation row and dragging your mouse up and down. What happened?

The ball isn't moving. When the ball first dropped, it hit the floor and stopped. When I rotate the floor in one direction, the ball floats above it. When I rotate the floor in the other direction, it moves right through the ball.

This happens because without a Rigidbody component, the Floor isn't part of Unity's physics simulation at all. But something's weird here. If the ball is supposed to use gravity, why doesn't it keep falling? Something's weird here...

2. **Stop the game**. Make sure Floor is still selected in the Hierarchy window, then **add a Rigidbody component** to the floor (but don't change any of its settings). Start your game again. What happened?

The floor and the ball both fell, because they were both affected by gravity. When I watch carefully as they fall, the ball's shadow stayed in the same position on the floor because they were both falling at exactly the same rate.

3. **Stop your game**. **Uncheck the "Use Gravity" box** in your Floor GameObject's Rigidbody component. Start your game again. What happened?

The ball started falling. As soon as it hit the floor, the floor started falling too—it looked like it was floating, like it was in outer space.

4. **Stop your game**. **Check the "Is Kinematic" box** in your Floor GameObject's Rigidbody component. Start your game again. As the game is running, **drag your mouse** over the X, Y, and Z Rotation labels in the Floor GameObject's Transform component to **tilt the floor**. What happened?

The ball stopped falling as soon as it hit the floor. When the floor tilted, the ball rolled around exactly how you'd expect it to. If I rotate the floor faster than the ball falls, gravity kicks in and it falls down to the floor. If I rotate it too fast, the ball flies right through it.

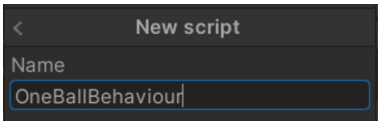
When you're done running your experiments, don't forget to reset the Repeat Rate field in the Main Camera's GameController script component back to 1. But you should keep the Rigidbody component that you added to the Floor GameObject. The ball and floor are now interacting with each other the way that we expect them to, and we want to keep that behavior for the game. Also, add a Rigidbody component to each of the walls with "Is Kinematic" checked and "Use Gravity" unchecked.

Add a script to your ball prefab that uses physics

Starting to get a feel for how physics makes your ball and floor interact just like physical objects? Let's take the next step and write some code that uses physics. First, make sure that you've got your GameObjects set up:

- Floor has a Rigidbody component with Is Kinematic checked and Use Gravity unchecked.
- OneBall Prefab has a Rigidbody component with Is Kinematic unchecked and Use Gravity checked.
- The Main Camera's Repeat Rate field is set to 1.

Now we're ready to use forces to move your billiard balls around the floor.



1 Add a script called OneBallBehavior to your prefab.

Go to the Project window and click on OneBall Prefab in the Prefabs folder to edit the prefab. Then click the **Add Component** button, choose New Script, and enter the name **OneBallBehaviour**, click New Script, then click Create and Add. This will add the script to the Assets folder, so use the Project window to drag it into the Scripts folder.

2 Store a reference to the Rigidbody component in a field.

In the last two Unity Labs you used a field to store a reference to the player's NavMesh Agent. Now you'll do exactly the same thing, except this time you'll store a reference to the Rigidbody component.

```
public class OneBallBehaviour : MonoBehaviour
{
    private Rigidbody rigidBody;

    void Awake() {
        rigidBody = GetComponent<Rigidbody>();
    }
```

The Rigidbody.AddForce method adds a force, like kicking the GameObject in a direction. The Rigidbody.velocity property is a vector that gets or sets the speed in a specific direction. A larger vector means faster speed. You can stop a GameObject's movement by setting its Rigidbody.velocity to Vector3.zero.

3 Add forces to your billiard balls.

Add a method called MoveMe, and use InvokeRepeating to call it over and over again. MoveMe does two things: first, it resets the velocity so the ball stops moving; then it adds a force in a random direction.

```
    void Start() {
        // Wait one second, then call the MoveMe method every 1.5seconds
        InvokeRepeating("MoveMe", 1f, 1.5f);
    }

    private void MoveMe() {
        // Remove the force MoveMe added the last time it was invoked
        rigidBody.velocity = Vector3.zero;
        // Add a force in a random direction
        rigidBody.AddForce(Random.insideUnitSphere * 500f);
    }
}
```

The MoveMe method stops the ball, then adds a force in a random direction.

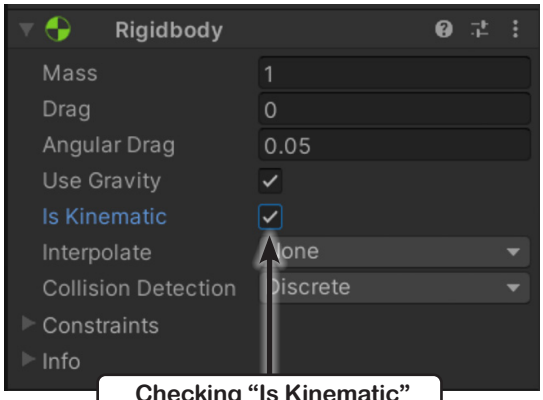
We want the ball to suddenly change direction, so first we stop it, then we add a new force.

Now run your game. The balls should start flying around in random directions, changing directions every 1.5 seconds. Sometimes they'll fly off the floor and gravity will pull them down into the abyss. Remember, the camera's MoveClick script is making it follow the player, so click on the floor to move the camera and get a better view. You can also switch to the Scene view to see what's happening.

The Unity physics engine

Did you ever take a physics class? If you did, you studied things like how objects move, accelerate, and bounce off of each other, and how they move when force is applied—like when they're hit by other objects or affected by gravity. You probably guessed that there's a lot of math involved in simulating realistic physics in video games. Luckily, you don't need to know that math to include realistic physics in your game! The **Unity physics engine** does it for you.

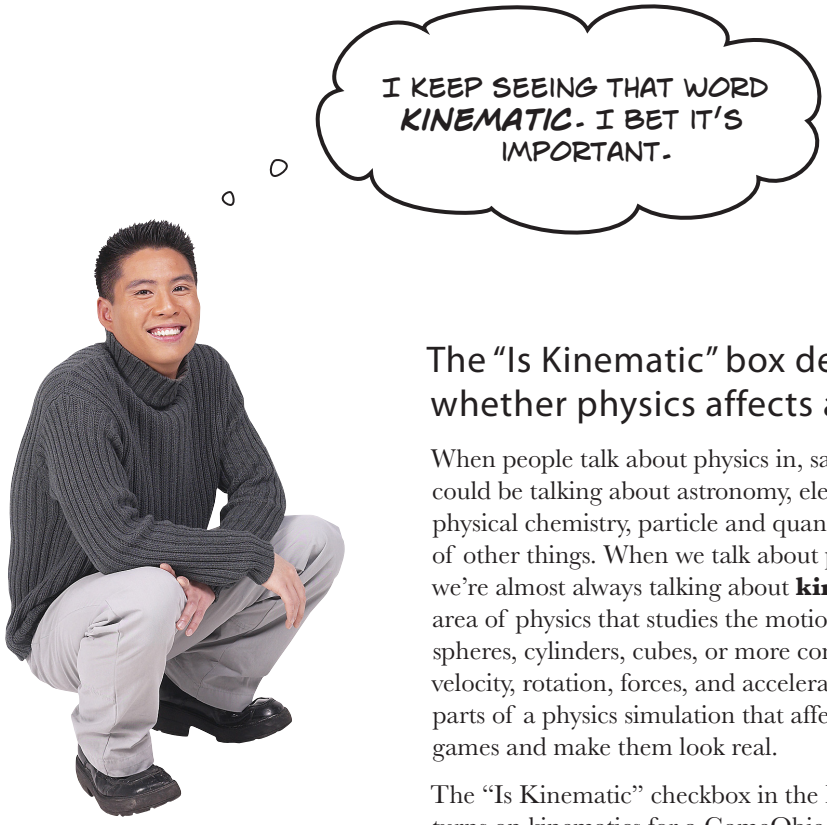
You use Unity's physics engine the same way you use the navigation AI: by **adding a component** to your GameObject that moves it around the scene. Unity's navigation system provides tools that let your GameObjects intelligently move around a scene. You already saw that you make your GameObject work with a navigation mesh by adding a NavMesh Agent component to it. The physics engine works the same way, except instead of making GameObjects navigate, it enables them to act under the control of physics: they can receive forces, respond to torque, and move and respond the way physical objects do by **adding a Rigidbody component**, which moves the GameObject around in response to those forces. You can add physical forces using scripts—for example, you can call a method to "nudge" your objects around (which you'll do in the next part of this Lab). You can also use physics to set up your scene so objects interact with each other.



Checking "Is Kinematic" prevents a Rigidbody from reacting to forces... and that includes gravity! So if you check Is Kinematic, your GameObject won't respond to gravity—even if the Use Gravity box is checked.

Unity uses an open-source **physics engine** called GameWorks PhysX. It's maintained by NVIDIA, the company that makes graphics hardware. The beauty of Unity's physics engine is that you don't actually need to know how any of it works. It just does. And you can enable any GameObject to work with the physics engine by adding a Rigidbody component to it.

You can learn more about GameWorks PhysX here:
https://developer.nvidia.com/gameworks-physx-overview



The "Is Kinematic" box determines whether physics affects a rigidbody.

When people talk about physics in, say, a university class, they could be talking about astronomy, electricity and magnetism, physical chemistry, particle and quantum physics, or a lot of other things. When we talk about physics in video games, we're almost always talking about **kinematics**. That's the area of physics that studies the motion of points, bodies (like spheres, cylinders, cubes, or more complex objects), mass, velocity, rotation, forces, and acceleration—all the different parts of a physics simulation that affect objects in 3D video games and make them look real.

The "Is Kinematic" checkbox in the Rigidbody component turns on kinematics for a GameObject. When you check this box, the physics engine keeps your GameObject's Rigidbody component in the physics simulation, but it ***prevents it from reacting to forces or collisions***. You saw that first-hand—when the "Is Kinematic" box was unchecked, the floor acted like it was in free fall. Once it was checked, the balls reacted to it but weren't able to knock it around, and you could still make it move by using its Transform component.

Use Debug.DrawRay to explore how forces work

In the second Unity Lab, you used Debug.DrawRay to explore vectors. Now we'll use it to get a feel for adding forces to GameObjects. Start by **adding a field called `forceAdded`** to hold the last force applied to the GameObject.

```
public class OneBallBehaviour : MonoBehaviour
{
    private Rigidbody rigidBody;
    private Vector3 forceAdded;
```

Then **modify the MoveMe method** to use the field.

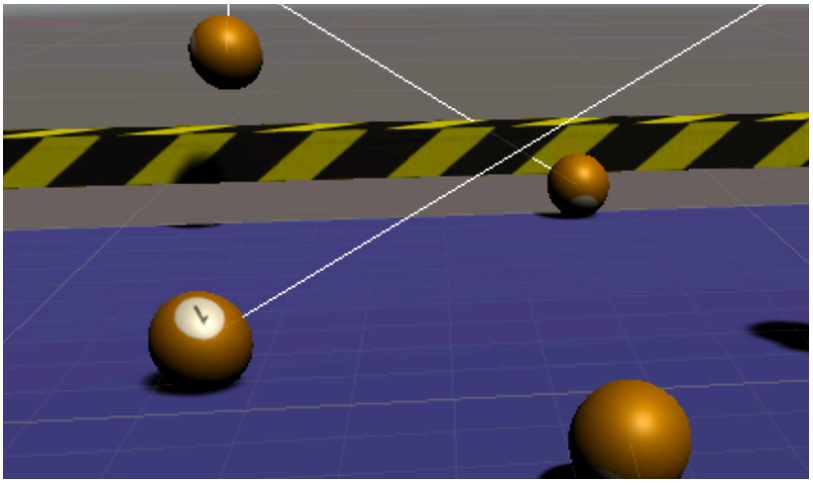
```
private void MoveMe()
{
    // Remove the force MoveMe added the last time it was invoked
    rigidBody.velocity = Vector3.zero;

    // Add a force in a random direction
    forceAdded = Random.insideUnitSphere * 500f;
    rigidBody.AddForce(forceAdded);
}
```

Finally, **call Debug.DrawRay from the Update method** so you can actually see the last force added.

```
private void Update()
{
    Debug.DrawRay(transform.position, forceAdded, Color.white);
}
```

Now run your game. Remember, ***Debug.DrawRay only affects the Scene view***, so switch to it. Now you'll see vectors coming out of each ball that show you the last force that was added to it.

You can visualize the last force that was added to each ball by storing its vector in a field and using it to Debug.DrawRay to draw it in the Scene view.

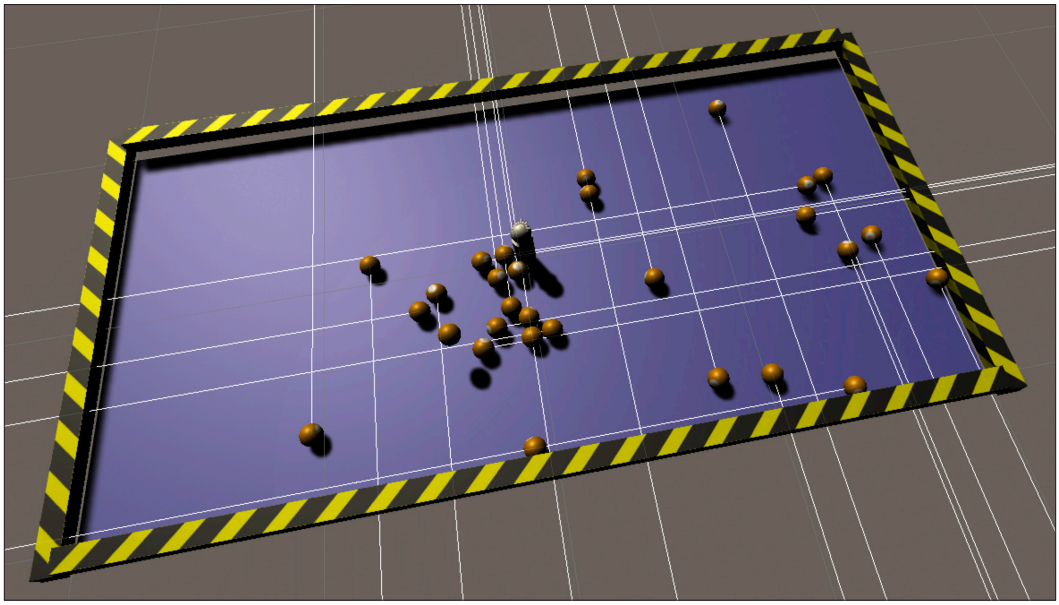


Use Debug.DrawRay to explore how forces work

Let's try one more thing. The balls are moving around in a pretty chaotic manner because we're adding random forces in all directions, including directions away from the floor. Let's make this look more orderly by only moving the balls forward, backward, left, and right along the floor. Replace the MoveMe method with this new one:

```
private void MoveMe()
{
    if (forceAdded == Vector3.left * 500f) {
         forceAdded = Vector3.forward * 500f;
    } else if (forceAdded == Vector3.forward * 500f) {
        forceAdded = Vector3.right * 500f;
    } else if (forceAdded == Vector3.right * 500f) {
        forceAdded = Vector3.back * 500f;
    } else {
        forceAdded = Vector3.left * 500f;
    }

    rigidBody.velocity = Vector3.zero;
    rigidBody.AddForce(forceAdded);
}
```

Now run your game again. Now the balls are moving in a much more orderly manner. Keep watching as more and more balls are added. Now **switch to the Scene view** to see the rays that show the last force added to each ball. Follow an individual ball and you'll see that it was first nudged forward, then right, then back, then left.



Now the balls are only moving in four directions, and the rays make it more obvious how that works.

Rigidbody Forces Up Close

Let's take a closer look at the statement that's actually moving the balls around:

```
rigidBody.AddForce(new Vector3(Multiplier - Random.value * Multiplier * 2, 0,
                               Multiplier - Random.value * Multiplier * 2));
```

The AddForce adds a **force** to a Rigidbody, causing it to move. If you haven't taken a physics class (or if it's been a while since you've taken one), think of that as giving a "nudge" to the object to get it to move.

Your OneBallBehavior.MoveMe method calls the GameObject's Rigidbody.AddForce method and **passes it a vector**. We experimented with vectors back in the second Unity Lab, so you know that a vector has a direction and a length. The **direction** of the vector tells AddForce which way to nudge the ball, and its **magnitude** (length) says how hard to nudge it.

When you call AddForce with just a vector argument, Unity applies a **continuous** force to the GameObject.

But you can also pass an optional **ForceMode.Acceleration** argument after the vector. That tells it to accelerate, which means it gets faster and faster, accelerating more and more each time you call AddForce to accelerate the object.

Your MoveMe method is calling AddForce with no parameters, which not only adds a continuous force, but also takes **mass** into account (which you can set in the Rigidbody component). If you've ever read about Galileo dropping objects off of the Leaning Tower of Pisa, you learned that gravity has the same effect on all objects, regardless of their mass. That's how gravity works: the longer an object falls, the faster it gets. And that's how ForceMode.Acceleration works, too.

If you want AddForce to take mass into account, pass it the **ForceMode.Impulse** argument. That will cause more massive objects to accelerate more slowly. And you can also apply a continuous force that doesn't take mass into account by passing AddForce the **ForceMode.VelocityChange** argument.

Start a second instance of Unity and create a new Unity project to experiment with different force modes

It's worth taking a few minutes to experiment with the different forces. Try **creating a new Unity project**, adding a sphere—make sure it has a Rigidbody with "Use gravity" unchecked—and adding different kinds of forces to it. Use the **same Awake method and private field** as the script you just added, then **add this OnMouseDown method**:

```
public float Thrust = 100F;
void OnMouseDown() {
    rigidBody.AddForce(Vector3.right * Thrust, ForceMode.Acceleration);
}
```

Experiment with the different ForceMode options to get a sense of how they work.

It helps to add a script that keeps the camera pointed at the object. And you might want to add make it set itself back to zero on right mouse click, and add other objects in the background so you can see exactly how fast your object is going.

Make your billiard balls move faster and faster

Right now your OneBallBehaviour script sets Rigidbody.velocity to Vector3.zero to stop it from moving before adding a new force. But what happens if you don't stop the ball before adding a force? The same thing that happens if you kick a ball that's already moving—you just add more force to it. If you kick it in the same direction it's already going, it goes faster. If you kick it in a different direction, it goes off at an angle.

So let's modify your OneBallBehaviour script to keep adding bigger and bigger forces to your ball. We'll start by **adding float fields called Multiplier and MaxMultiplier**:

```
public class OneBallBehaviour : MonoBehaviour
{
    private Rigidbody rigidBody;
    private Vector3 forceAdded;
    public float Multiplier = 100f;
    public float MaxMultiplier = 3000f;
```

Did you notice that your scripts all extend the MonoBehaviour class? Try changing the uppercase 'B' to a lowercase 'b' in the rigidBody field name—you'll get a compiler warning about hiding an inherited member. You learned about hiding members in Chapter 6.

Now **modify your MoveMe method** to add a random force:

```
private void MoveMe()
{
    forceAdded = new Vector3(Multiplier - Random.value * Multiplier * 2,
                  0, Multiplier - Random.value * Multiplier * 2);
    rigidBody.AddForce(forceAdded);
    Multiplier += 100f;
    if (Multiplier > MaxMultiplier) Destroy(gameObject);
}
```

Take a closer look at the MoveMe method. Here's how it works:

1. The first line generates a random vector. It adds a random force in X and Z, but no force in Y (because we don't want the ball to go up or down, but keep in contact with the floor).

2. The X and Z directions are calculated like this: `Multiplier - Random.value * Multiplier * 2` That's just like what you did in the third Unity Lab to generate a random number between -6 and 6, except this time instead of the literal `6` you're using the Multiplier field.

3. The second statement in the MoveMe method actually adds the force. Unlike before, it doesn't use Rigidbody.velocity to stop the ball, so the forces keep getting added to the ball, so more and more force is added to the ball each time, causing it to move faster and faster.

4. The third statement increases the value in the Multiplier field. The first time MoveMe is called, Multiplier is 100, the second time it's 200, the third time it's 300, etc.

What does the last line of the MoveMe method do?

.....

.....

.....

MINI Sharpen your pencil Solution

What does the last line of the MoveMe method do?

It checks the Multiplier field to see if it exceeded the maximum value of 3000, then it destroys this instance of OneBall. Since Multiplier gets increased by 50 each half-second, it will take 30 seconds before Multiplier hits 3000, and causes each OneBall to disappear from the floor after 30 seconds.

Get creative!

In the next Unity Lab, you'll take the code you've created so far and turn it into a game. This is a great chance to get creative with another paper prototype. Here are the rules of the game you'll finish building in the next Unity Lab:

- The game starts with the player on an empty floor. Billiard balls are dropped one by one onto the floor.
- When the player collides with a ball, it disappears and the score goes up by 1.
- When two balls collide, they make a "clack" sound. When the player gets a ball, it makes a "chirp" sound.
- The game is over when there the total number of balls on the floor gets too big.

Try building a paper prototype of that game. Draw the floor on a piece of paper, then tear off scraps of paper for the player and billiard balls. Can you come up with new mechanics to make the game more interesting?

BULLET POINTS

- Unity's **physics engine** makes your GameObjects respond to nudges, bumps, collisions, and gravity, just like a real object would.
- Adding a **Rigidbody component** to a Gamebject tells Unity to include it in its physics simulation.
- The **Tiling** value of a material causes the texture map to repeat that many times for each face.
- Clicking the **select button (◎) next to a field** opens a window that lets you choose an appropriate object.
- You can use the **Add Component button to add a C# script** to a GameObject, or to multiple GameObjects at the same time.
- Adding a Rigidbody component causes a GameObject to be included in the **physics simulation**, which will automatically move it around the scene in response to simulated physical forces.
- The Rigidbody's **"Use Gravity" checkbox** causes the GameObject to respond to gravity.
- The Rigidbody's **"Is Kinematic" checkbox** turns on kinematics, which keeps the Rigidbody ins the physics simulation but prevents it from reacting to forces or collisions.
- Gravity is a force, so if you make a Rigidbody kinematic it **won't respond to gravity**, even if the "Use Gravity" box is checked.
- The **Rigidbody.AddForce method** adds a force to a GameObject, causing it to move in the scene. It takes a vector as a parameter and uses its direction and magnitude to determine the direction and strength of the force to add to the GameObject.
- You can add **different kinds of forces** to a rigidbody, including continuous or acceleration forces that either do or don't take mass into account.

Unity provides a really detailed—and very readable!—manual with information about all of the tools you can use in scripts. Choose Scripting Reference from the Help menu, search for Rigidbody, and scroll down to the section called Public Methods. You can also look up Rigidbody.AddForce and ForceMode directly. It's worth taking a little time to read through those manual pages.