

O'REILLY®

Fourth  
Edition

# Head First

# C#

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

---

Andrew Stellman  
& Jennifer Greene



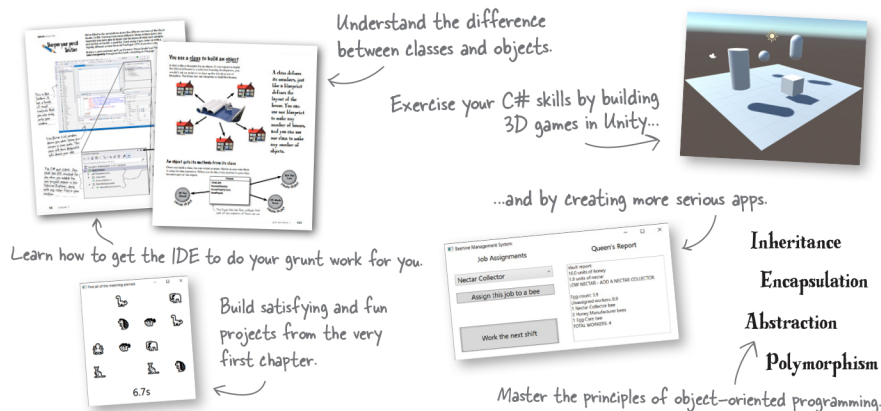
A Brain-Friendly Guide

# 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!



## 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

Beijing • Boston • Farnham • Sebastopol • Tokyo

**O'REILLY®**

# 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.

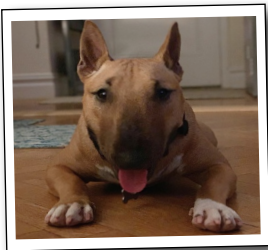
Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

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).

<b>Series Creators:</b>	Kathy Sierra, Bert Bates
<b>Cover Designer:</b>	Ellie Volckhausen
<b>Brain Image on Spine:</b>	Eric Freeman
<b>Editors:</b>	Nicole Taché, Amanda Quinn
<b>Proofreader:</b>	Rachel Head
<b>Indexer:</b>	Potomac Indexing, LLC
<b>Illustrator:</b>	Jose Marzan
<b>Page Viewers:</b>	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.

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]

[2020-11-13]

# Unity Lab

## Unity Boss Battle

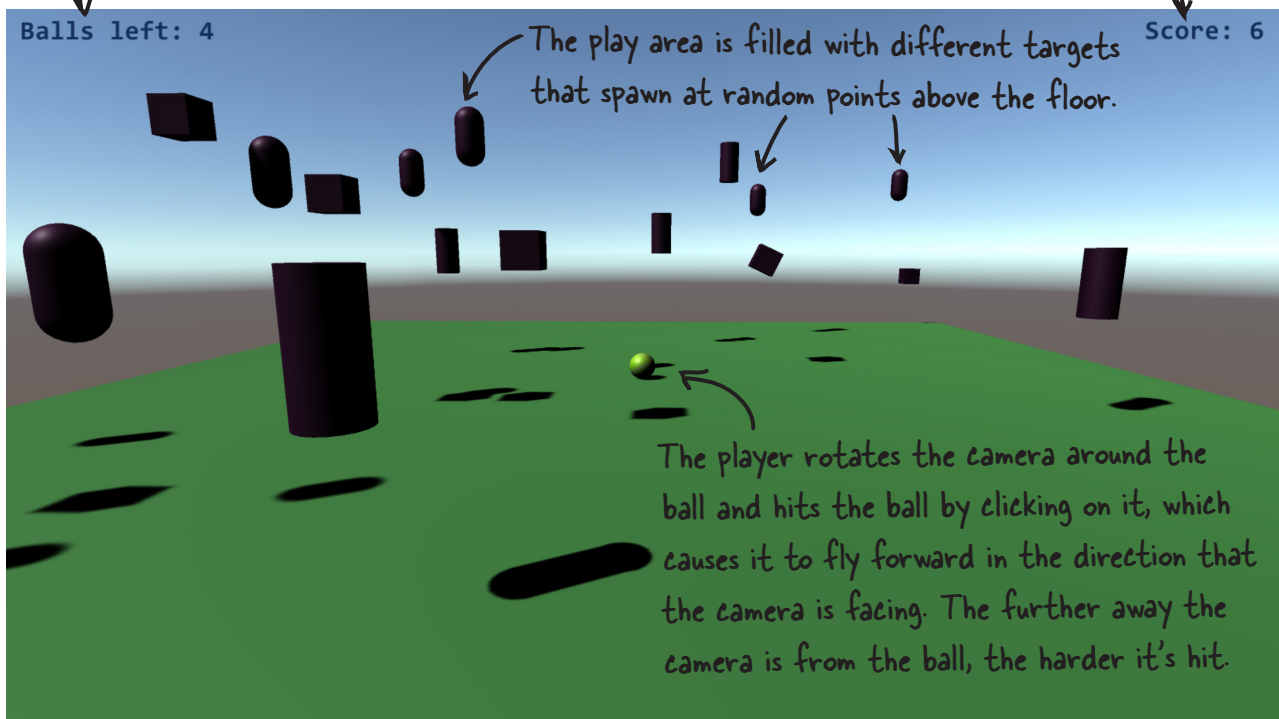
If you've played a lot of video games (and we're pretty sure you have!) then you've had to fight a whole lot of **boss battles**—those fights at the end of a level or section where you face off against an opponent that's bigger and stronger than what you've seen so far. Well, now it's time for a Unity boss battle: a Unity project that's bigger and more complex than the projects you've built so far in the other Unity Labs.

In this Unity Lab, you'll build a **more complex physics-based game** where you knock targets out of a scene by hitting a ball into them. It's got **more complex mechanics** than previous Labs that use physics and collision detection to drive the game. You'll rely on lots of tools that you've learned so far about C# and Unity. And we'll throw a few new things at you, too—because an important part of being a Unity developer (or any kind of developer, in fact!) is figuring out how to use new tools on the fly.

### The challenge: build a physics-based game

You've learned a LOT about Unity and C#—and now you get a chance to put that knowledge to the test. In this Unity Lab, you'll create a game that combines many of the techniques and concepts that you've learned so far in this book (plus one or two new things that we'll give you along the way). The goal is to **build a physics-based game** where the player uses a ball to hit targets out of the play area, scoring a point for each target knocked out.

The goal of the game is to use the ball to knock targets out of the play area. The player scores a point for each target that flies out of bounds. A player only gets a certain number of balls per game, and loses a ball any time they hit a ball and it falls off the floor without colliding with a target first.



When a player knocks a target out of the play area, a new one spawns at a random point above the floor. The game chooses a random shape for each new target.



### A Unity Lab in three parts

There's an old joke about long projects that project managers like to tell each other:

Q: "How do you eat an elephant?"     A: "One bite at a time!"

The authors of Head First C#  
do not condone eating elephants.  
In fact, we ♥ elephants.

Okay, so it's not particularly funny. But it makes an important point about longer projects: no matter how big they are, you can always break them down into small chunks that you can "eat" one at a time.

So to help guide you through this project, we've broken it down into three parts:

- ★ **Part 1: Set up the play area.** You'll set up the play area, creating the floor and the ball, and scattering randomly shaped targets at random locations over it.
- ★ **Part 2: Add the ball hitting mechanic.** You'll use the MoveCamera script you created in Chapter 6 as a starting point. Then you'll use raycasting to launch the ball in the direction that the camera is pointing.
- ★ **Part 3: Add a score and balls left, and a UI to display them.** You'll add fields to track the score, the number of balls left, and whether or not the game is over. Then you'll add a UI so players can start the game and see their score.

### Does this Unity Lab seem a little... well, intimidating?

*We built this lab to give you practice... and to give you a challenge.*

Every project, no matter how big or complex, can be broken down into smaller pieces that you can do one at a time. Start by finding one part of it that you know you can do, and do that part first. Then find another one and do that. Keep doing that, and pretty soon you're done!

### You have lots of space to get creative!

We intentionally did not include any sounds and only included a minimal UI in this Unity Lab. We wanted to give you a complete project with a lot of opportunities to use your creativity to build on it. At the end of this Lab, we'll give you some ideas to help you expand the game. See if you can find more ways to make it interesting!

And remember, *it's **not cheating to peek at our solution***. In fact, just the opposite—it's a great way to learn and understand one approach to the problem. If you found a different way to build this game, that's equally valid!

### You can do this!

There are an almost infinite number of ways to solve any programming problem. Your code may look completely different than ours—and that's okay! If it works, you did the project right.

**Ready? Great! Let's get started.**

## Part 1: Set up the play area

When the game starts, it spawns targets scattered around the play area. So in this first part of the lab, you'll:

- ★ Add a GameObject for the floor with its material set to a color you like (we used 61A454)
- ★ Add a GameObject for the ball with its material set to a texture map
- ★ You'll create prefabs for each of the types of target
- ★ You'll add a script for the target behavior that spawns each target at a random location above the floor
- ★ You'll create an empty GameObject and give it the tag GameController
- ★ You'll add a script to the GameController object that spawns the targets

← In the previous Unity Labs, you attached the game controller script to the Main Camera. This time you'll attach it to an empty GameObject. Those are both valid approaches.

### Create the ball and floor, and game controller

Start by **adding three GameObjects**: a plane for the floor, a sphere for the ball, and an **empty GameObject** for the game controller. Unity created your project with Player and GameController tags, so you just need to add a Floor tag. You'll also need to **add a material for the floor** with albedo color 61A454.

Name	Type	Position	Rotation	Scale	Tag
Main Camera	Camera	(0, 5, -15)	(0, 0, 0)	(1, 1, 1)	MainCamera
Directional Light – set its color to E8E8E8 and its intensity to 1.15	Light	(0,3, 0)	(50, -30, 0)	(1, 1, 1)	
Floor – <i>don't</i> add a Rigidbody	Plane	(0, 0, 0)	(0, 0, 0)	(4, 1, 4)	Floor
Ball – with a Rigidbody component that has “Use Gravity” checked and its Sphere Collider’s material set to a Physic material called <i>Ball Physic</i> with Bounciness set to 1	Sphere	(0, 10, 0)	(0, 0, 0)	(1, 1, 1)	Player
Game Controller	Empty	(0, 0, 0)	(0, 0, 0)	(1, 1, 1)	GameController

We want the ball to look a little more interesting, so **download Tennis ball texture.png** from the book's GitHub page and use it to **create a material**, then drag it onto Ball. We want the ball to bounce a little, so **add a Physic material** called *Ball Physic* with Bounciness set to 1 and drag it onto Ball too.

### Create a prefab for each of the different target types

The game will have targets shaped like cubes, cylinders, and capsules, so **add a prefab for each of them**:

- ★ Add a cube GameObject called *t*, a cylinder GameObject called *Cylinder Target*, and a capsule GameObject called *Capsule Target*, then create a new tag called Target and add it to each target GameObject
- ★ Create a new material called Target Material with albedo color 200020 and drag it onto each target
- ★ Add a Rigidbody with “Use Gravity” **unchecked** to each target GameObject
- ★ Turn each target into a prefab by dragging it into a new folder called Prefabs, then delete each of the targets from the scene so it only exists as a prefab



## 9. Unity Boss Battle



### Long Exercise

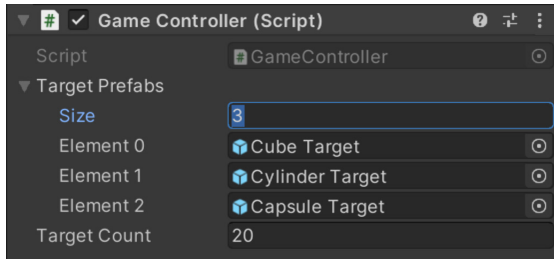
Create a **GameController** script that spawns random prefabs and a **TargetBehaviour** script that positions each spawned target at a random location above the floor.

#### Use a list of prefabs to spawn random target shapes

Create a new script called GameController and add it to the empty GameObject that you created. Then add two fields:

```
public List<GameObject> TargetPrefabs;  
public int TargetCount = 20;
```

Use the Inspector to populate the TargetPrefabs collection:



Make sure you added the tags to each of the GameObjects and prefabs.

When you add an collection field, you can use the Inspector to set its size and targets. Use the Select GameObject window to populate the collection with the three prefabs that you created for the targets.

TargetCount contains the number of targets in the play area at any time.

Here's how to populate the TargetPrefabs collection:

- Click on the Game Controller GameObject and **expand the Target Prefabs field** in the script component.
- The default size is 0. **Set its size to 3**—as soon as you do, the editor adds three elements to the collection.
- Use the select button (🔍) to bring up the Select GameObject window and **choose a prefab from the Assets tab**.

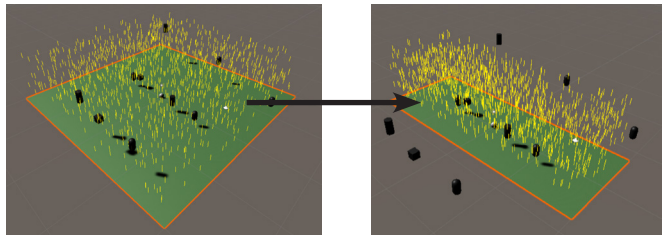
Once the collection is populated, modify the Awake method to instantiate TargetCount targets, choosing random prefabs from the TargetPrefabs list. Remember, UnityEngine.Random uses floats, but *you'll need ints* to choose a random prefab from the TargetPrefabs list, so add a field with a reference to an instance of System.Random.

#### Add a TargetBehaviour script to each prefab to make each target spawn at a random location above the floor

Use the TargetBehaviour.Awake method to make each target set its position to a random point above the floor.

- Use `GameObject.FindGameObjectWithTag("Floor")` to get a reference to the Floor GameObject
- Find the size of the plane by getting a Bounds: `var bounds = floor.GetComponent<Renderer>().bounds;` and using the center (`bounds.center.x` and `bounds.center.z`) and the X and Z sizes (`bounds.size.x` and `bounds.size.z`)—so the minimum X position is `centerX - (sizeX / 2) + 0.5f`
- Create a vector with random X, Y, and Z values. Use the Random.Range method to get a random value between two numbers—for example, get the Y position like this: `Random.Range(MinimumHeight, MaximumHeight)`

Create a method called `RandomPositionOverFloor` that returns a `Vector3` with a random point somewhere above the floor. Then add an Update method that uses `Debug.DrawRay`, and test the method by starting your game and changing the X or Z scale of the floor: when you make the floor narrower, the rays should stay on top of the floor.



You can add a script to a prefab by clicking its Add Component button in the Inspector, choosing Script—the Unity editor will list all of the scripts in the project so you can choose one.

## 9. Unity Boss Battle



### Long Exercise Solution

Here are the **GameController** and **TargetBehaviour** classes that choose random prefabs and scatter them at random locations above the floor. We included the Update method with `Debug.DrawRay` in the solution (feel free to keep it, or delete it once your code works).

```
public class GameController : MonoBehaviour
{
    public List<GameObject> TargetPrefabs;
    public int TargetCount = 20;
    private System.Random random = new System.Random();

    void Awake()
    {
        for (int i = 0; i < TargetCount; i++)
            Instantiate(TargetPrefabs[random.Next(0, 3)]);
    }
}
```

We used `System.Random` and not `UnityEngine.Random` because we needed int values (and not floats) to choose a random prefab from the `TargetPrefabs` collection.

```
public class TargetBehaviour : MonoBehaviour
{
    private GameObject floor;
    public float MinimumHeight = 2f;
    public float MaximumHeight = 10f;
```

```
    private void Awake()
    {
        floor = GameObject.FindGameObjectWithTag("Floor");
        transform.position = RandomPositionOverFloor();
    }
```

**It's okay if your code looks different from ours. There are MANY ways to write the same program. If you did things differently, take the time to understand the way we built our code.**

```
    private Vector3 RandomPositionOverFloor()
    {
```

```
        var bounds = floor.GetComponent<Renderer>().bounds;
        var centerX = bounds.center.x;
        var centerZ = bounds.center.z;
        var sizeX = bounds.size.x;
        var sizeZ = bounds.size.z;
        var minX = centerX - (sizeX / 2) + 0.5f;
        var maxX = centerX + (sizeX / 2) - 0.5f;
        var minZ = centerZ - (sizeZ / 2) + 0.5f;
        var maxZ = centerZ + (sizeZ / 2) - 0.5f;
        var randomPosition = new Vector3(
            Random.Range(minX, maxX),
            Random.Range(MinimumHeight, MaximumHeight),
            Random.Range(minZ, maxZ));
        return randomPosition;
    }
```

Go back to Unity Lab 4 and look at how you set the position in the `ResetBall` method. These calculations are very similar.

```
    void Update()
    {
        Debug.DrawRay(RandomPositionOverFloor(), Vector3.up, Color.yellow, 1f);
    }
}
```

`RandomPositionOverFloor` returns a new `Vector3` with a random X and Z position over the floor, and a random height that uses the `MinimumHeight` and `MaximumHeight` fields.

Try removing the `TargetBehaviour` script from the prefabs—the targets explode outwards in every direction. Why do you think that happens?

## Part 2: Add the ball-hitting mechanic

Let's take advantage of the MoveCamera script that you created back in Chapter 6 to create a mechanic for hitting the ball. Here's how it will work:

- ★ The player uses the arrow keys to rotate the camera around the ball, and the mouse scroll wheel to zoom in and out. This camera movement works just like it did in the previous Unity Labs (so you can reuse the same script that you've been using since Chapter 6).
- ★ When the player clicks on the ball, the game finds the distance between the camera's viewport and the ball and uses that to determine the amount of force to add to the ball.
- ★ If the ball falls off of the floor, it resets to its initial position.



### Long Exercise

The ball hitting mechanic has two parts: positioning the camera and applying force to the ball when the user clicks on it. To implement it, you'll need scripts for the Main Camera and the Ball GameObject.

**Add the MoveCamera script from Unity Lab 6 to your project, then drag it onto your Main Camera GameObject.**

- **Reuse the same MoveCamera script you created before.** You can copy and paste the script you used in Unity Lab 6. Drag it onto your camera, just like you did with GameController earlier. We want the camera to always look at the ball, so **set the MoveCamera.Player** field to reference the Ball GameObject.
- **Experiment with the MoveCamera field settings.** The MoveCamera.Angle field determines the speed that the camera rotates when the player uses the arrow keys, and the ZoomSpeed field determines the speed that it zooms in and out when the player uses the scroll wheel. Find settings that feel comfortable for your game and save them.

**Add a new BallBehaviour script and drag it onto your Ball GameObject.**

*You can also use the Add Component button in the Inspector to add scripts.*

- **Have the Update method call a new method called HitBallMechanic that uses raycasting to add a force to the ball when the player clicks on it.** This is really similar to the MoveToClick script from the last Lab, except it uses raycasting to add a force to the ball.
  1. Use the Awake method to set a Camera field called cameraComponent with a reference to the main camera.
  2. Add a private bool field called clicking. You'll use it to detect the click so you only hit the ball when the mouse button is first pressed (but not while it's still pressed). Add a private float field called multiplier and set it to 100f.
  3. Check if the user is not currently clicking but the mouse is down—if so, set clicking to true to track that the user is clicking **so we only add force once per click**. Then check if the hit object's tag is Ball, get its Rigidbody, and add a force: `rigidBody.AddForce(ray.direction * hit.distance * multiplier);`
  4. Add an else clause to the if statement that uses the Input.GetMouseButtonUp(0) method (to check if the right mouse button is not being clicked) that resets clicking to false to indicate that the user is not currently clicking.
- **Have the Update method call a new method called OutOfBoundsCheck that resets the ball's position if it falls off of the floor.** In the Awake method save the Vector3 with its starting position and references to the GameController and the ball's Rigidbody. If the ball's Y position **falls below -10**, it's rolled off the ground, so reset its position to the starting position, call `gameController.PlayerOutOfBounds`, and make the ball **stop moving in any direction** by setting its Rigidbody's velocity and angularVelocity to `Vector3.zero`.

## 9. Unity Boss Battle



### Long Exercise Solution

Here's the **BallBehaviour** script that gets attached to the ball. The Update method calls the out-of-bounds check and the hit mechanic, which uses raycasting to figure out when the player clicks the ball and the collision information to figure out how hard to hit the ball.

```
public class BallBehaviour : MonoBehaviour
{
```

```
    private bool clicking = false;
    private float multiplier = 100f;
    private Rigidbody rigidBody;
    private Camera mainCamera;
    private Vector3 startingPosition;
```

This script uses its Awake method to get references to its Rigidbody component and the main camera. It also saves the starting position so it can reset the ball if it goes out of bounds.

```
    void Awake()
```

```
    {
        rigidBody = gameObject.GetComponent<Rigidbody>();
        mainCamera = GameObject.FindGameObjectWithTag("MainCamera").GetComponent<Camera>();
        startingPosition = transform.position;
    }
```

```
    void Update()
```

```
    {
        HitBallMechanic();
        OutOfBoundsCheck();
    }
```

This is a simple way to check if the ball is out of bounds: if its Y position drops below -10 it's fallen off the floor. Try choosing a different Y position - the lower you let it fall, the longer the delay before the ball gets reset.

```
    private void OutOfBoundsCheck()
```

```
    {
        if (transform.position.y < -10f)
        {
            transform.position = startingPosition;
            rigidBody.velocity = Vector3.zero;
            rigidBody.angularVelocity = Vector3.zero;
        }
    }
```

We gave you this method in the instructions to stop the ball so it stays still after it gets reset.

```
    private void HitBallMechanic()
```

```
    {
        if (!clicking && Input.GetMouseButtonDown(0))
        {
            clicking = true;
            Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;
            if (Physics.Raycast(ray, out hit, 100))
            {
                GameObject hitObject = hit.collider.gameObject;
                if (hitObject.tag == "Player")
                {
                    Rigidbody rigidBody = hitObject.GetComponent<Rigidbody>();
                    rigidBody.AddForce(ray.direction * hit.distance * multiplier);
                }
            }
        }
        else if (clicking && Input.GetMouseButtonUp(0)) clicking = false;
    }
}
```

If the clicking field is false and Input.GetMouseButtonDown(0) is true, that means this is the first frame that a mouse click was detected.

Make sure that the Ball is tagged "Player" the and targets are tagged "Target" to make the ball-hitting mechanic work.

Here's a great application of vector arithmetic. Multiplying the ray's direction vector by the distance from the gives the right force to add.

## Part 3: Add a score and balls left, and a UI to display them

Let's turn this into a game. We'll start by adding two simple mechanics:

- ★ If the player hits a target out of bounds, it disappears, a new target appears, and the player scores a point.
- ★ The player starts with a limited number of balls. If the player hits a ball and it ***didn't collide*** with a target ***before*** it went out of bounds, that ball is lost and the number of balls left is decremented.

Then we'll add a very simple UI with a "Play Again" button and text in the corners of the UI to display score and balls left. Now the player has a limited number of chances to hit the targets. They get all the shots they want—as long as those shots collide with a target. If they don't, they lose a ball. Lose the last ball and the game ends.



### Exercise

Add a "target out of bounds" mechanic that adds a point to the player's score when a target flies out of bounds, then add a "Game Over" mode so you can end the game when the player runs out of balls.

**Modify GameController to add a "Game Over" mode, score, and ball limit.**

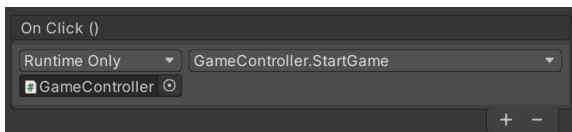
- **Add BallsPerGame and GameOver.** Add a public int field called BallsPerGame—set it to 5 as a starting point. Then add a read-only Boolean property GameOver that defaults to true. You'll also need private fields to keep track of the number of balls left and the score.
- **Add a public StartGame method.** When the game starts, it should set the private fields that track the number of balls left and the score, then set the GameOver property to false. Add a Start method that calls StartGame.
- **Add a public PlayerScored method.** This will get called by a target when it flies out of bounds, just before it destroys itself. It should increment the private score field and instantiate a new target to replace the destroyed one.
- **Add a public BallLost method.** Decrement the balls left and set GameOver to true if the player runs out of balls.

**Modify BallBehaviour and TargetBehaviour.**

- **Make BallBehaviour call BallLost if it goes out of bounds without hitting a target.** Add a Boolean field that's set to false when the ball is hit and true if it collides with a target. Use that field to decide whether to call BallLost.
- **Make TargetBehaviour call PlayerScored and destroy itself if it goes out of bounds.** Add a public field called TooFar and set it to 20f. Check if it went out of bounds on the X, Y, or Z axes. Here's an example of how you can check if it went out of bounds on the X-axis: `Mathf.Abs(transform.position.x) > tooFar`

**Add a UI with a Text for the score, a Text for the balls left, and a "Play Again" button**

- **Add a UI with two Text GameObjects and a Button.** Choose a font, color, and position for each Text, just like you did in previous Unity Labs. Make the button's OnClick call GameController.StartGame.



- **Modify GameController to update the UI.** Add two Text fields to GameController, and an Update method that sets their Text properties. Then make StartGame hide the button, and BallLost show it again when the game ends.

## 9. Unity Boss Battle



### Long Exercise Solution

The **GameController** needs to keep track of whether or not the game is over, update the UI, and give the other **GameObjects** methods to call when the player scores or loses a ball.

```
public class GameController : MonoBehaviour
{
```

```
    public bool GameOver = true;
    public int BallsPerGame = 5;
    private int ballsLeft = 0;
    private int score;
    public Text ScoreText;
    public Text BallsLeftText;
    public Button PlayAgainButton;
```

Don't forget that you need a using **UnityEngine.UI**; declaration to add a **Text** or **Button** field. Make sure you use the **Inspector** window to set their values.

```
    public List<GameObject> TargetPrefabs;
    public int TargetCount = 20;
    private System.Random random = new System.Random();
```

```
    void Awake()
```

```
    {
        for (int i = 0; i < TargetCount; i++)
            Instantiate(TargetPrefabs[random.Next(0, 3)]);
    }
```

This is the part of the **GameController** class that you built in Part 2 of the lab.

```
    private void Start()
```

```
    {
        StartGame();
    }
```

```
    public void StartGame()
```

```
    {
        ballsLeft = BallsPerGame;
        score = 0;
        GameOver = false;
        PlayAgainButton.gameObject.SetActive(false);
    }
```

When the game starts, the game controller resets its fields and hides the "Play Again" button.

```
    public void PlayerScored()
```

```
    {
        score++;
        Instantiate(TargetPrefabs[random.Next(0, 3)]);
    }
```

When the player scores, the game controller increments the score and instantiates a new target to replace the one that destroyed itself.

```
    public void BallLost()
```

```
    {
        ballsLeft--;
        if (ballsLeft <= 0)
        {
            GameOver = true;
            PlayAgainButton.gameObject.SetActive(true);
        }
    }
```

**BallLost** is called by the ball when it goes out of bounds without hitting a target, so it needs to decrement **ballsLeft**, and if that's the last ball it ends the game and shows the "Play Again" button.

```
    private void Update()
```

```
    {
        ScoreText.text = $"Score: {score}";
        BallsLeftText.text = $"Balls left: {ballsLeft}";
    }
```

The **Update** method updates the score and balls left text in the UI. Don't forget to use the **Inspector** to set the fields.

```
}
```



## 9. Unity Boss Battle



The **BallBehaviour** class needs to track if it hit a target after the player launched it. The **hitTarget** field to false when the player hits the ball, and true if it collides with a target.

```
public class BallBehaviour : MonoBehaviour
{
    private bool clicking = false;
    private float multiplier = 100f;
    private Rigidbody rigidBody;
    private Camera mainCamera;
    private Vector3 startingPosition;
    private GameController gameController;

    void Awake()
    {
        gameController = GameObject.FindGameObjectWithTag("GameController")
            .GetComponent<GameController>();
        rigidBody = gameObject.GetComponent<Rigidbody>();
        mainCamera = GameObject.FindGameObjectWithTag("MainCamera").GetComponent<Camera>();
        startingPosition = transform.position;
    }

    private bool hitTarget;

    void Update()
    {
        if (!gameController.GameOver)
        {
            HitBallMechanic();
            OutOfBoundsCheck();
        }
    }

    private void OutOfBoundsCheck()
    {
        if (transform.position.y < -10f)
        {
            transform.position = startingPosition;
            rigidBody.velocity = Vector3.zero;
            rigidBody.angularVelocity = Vector3.zero;
            if (!hitTarget)
                gameController.BallLost();
        }
    }

    private void HitBallMechanic()
    {
        if (!clicking && Input.GetMouseButtonDown(0))
        {
            hitTarget = false;

            // The rest of the HitBallMechanic method is the same as before
        }
    }

    void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.CompareTag("Target"))
            hitTarget = true;
    }
}
```

We'll need a reference to **GameController** to call its **BallLost** method.

This field is true if the ball is currently in flight and it's collided with a target.

The ball only responds to clicks and checks if it's out of bounds if the game is still running.

If the ball goes out of bounds without hitting a target, it should call **BallLost** after it resets itself.

We need to reset **hitTarget** each time the ball is hit.

We can use collision detection to figure out if the ball hit a target while it was in flight.

If your target prefabs don't have the "Target" tag, the collision enter event handler won't register the hit correctly and the player will lose a ball.

Did you use **==** to check the tag of the other **GameObject** in the collision? If you did, check the Quick Fix menu. The IDE gives you an option to automatically convert it to call the **GameObject.CompareTag** method, which is more efficient.

[🔧] (parameter) **Collision** collision  
Comparing tags using **==** is inefficient.  
[Show potential fixes](#) (Alt+Enter or Ctrl+.)



## Long Exercise Solution

Here's the updated **TargetBehaviour** script that checks if the target went out of bounds. If it did, it tells the GameController that the player scored, then destroys itself.

```
public class TargetBehaviour : MonoBehaviour
{
    private GameObject floor;
    public float MinimumHeight = 2f;
    public float MaximumHeight = 10f;
    private float tooFar = 20f;
    private GameController gameController;

    private void Awake()
    {
        floor = GameObject.FindGameObjectWithTag("Floor");
        transform.position = RandomPositionOverFloor();
        gameController = GameObject.FindGameObjectWithTag("GameController")
            .GetComponent<GameController>();
    }

    private Vector3 RandomPositionOverFloor()
    {
        // The RandomPositionOverFloor method stays the same
    }

    private void Update()
    {
        if (Mathf.Abs(transform.position.x) > tooFar || Mathf.Abs(transform.position.y) > tooFar
            || Mathf.Abs(transform.position.z) > tooFar)
        {
            gameController.PlayerScored();
            Destroy(gameObject);
        }
    }
}
```

Did you remember that you could use the **Add Component** button to the same script to multiple GameObjects? That's an easy way to add the **TargetBehaviour** script to all of the targets.

We'll need a reference to the game controller to call its **PlayerScored** method.

Here's where absolute value is especially important. If you just check the X, Y, or Z position you'll only detect when the ball flies off of one end of the floor.

The **Update** method checks if the target went out of bounds. If it did, it tells the game controller that the player scored (so it can increment the score and instantiate a new target), then it destroys itself.

## Get creative!

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

- ★ Experiment with the mechanics of the game to see how they affect the dynamics. Try changing the number of targets, the ball physics, making the ball get hit harder, adding or removing balls, making the camera move more quickly or slowly. (If you need a refresher on dynamics, mechanics, and feedback loops in games, flip back to Chapter 6.)
- ★ Can you figure out how to make the game score extra points if it lands on the floor without going out of bounds? How about if the player hits the ball a second time before it goes out of bounds? You'll need to watch for a collision with the floor.
- ★ Try making the play area get smaller after each target is hit. **Try adding sounds!**
- ★ Can you find other ways to make the game more interesting? How about having the player score more for higher targets, so the player needs to launch the ball into the air?
- ★ Try making each target keep track of how many other targets it collided with, and score more for extra target-on-target collisions. Try thinking of ways to make this game better!

A really big project can be divided into smaller parts. You can tackle them one by one until the project's done.