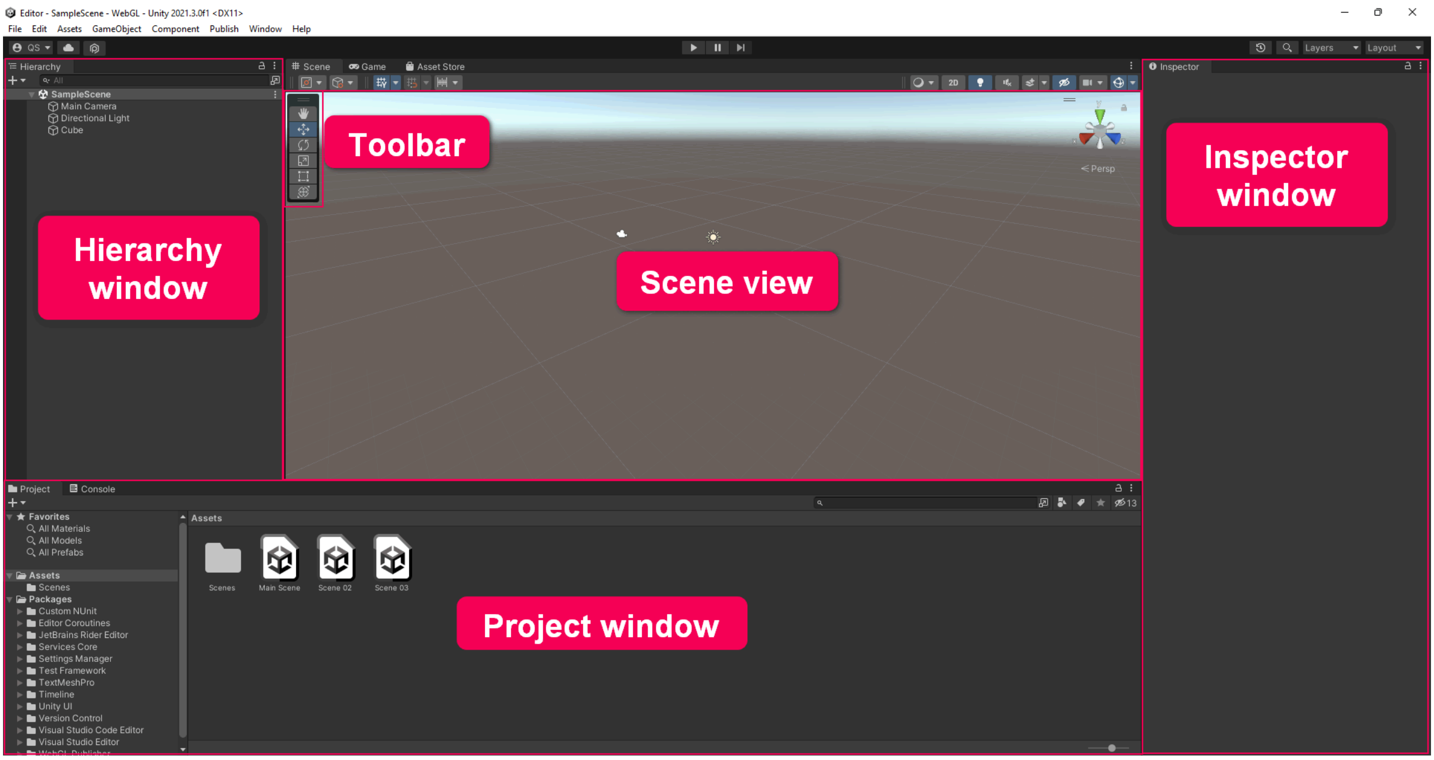
Code libraries: collections of high-level instructions for common behaviors

A GameObject:

- is Unity's name for an object that is active in a game. These can contain many components, including: a 3D model, texture information, collision information, C# code, and so on
- any object in the game—usually meaning anything that you see on screen—is a GameObject

---

Unity Editor



- The **Hierarchy** is where you can organize all the things in your project.
  - These things are called **GameObjects**.
    - If you add a **GameObject** to your project in Scene view, it will be listed in the **Hierarchy**. If you delete a **GameObject** from the scene, it will no longer be listed.
- The **Project window** is where you can find all the files (assets) available for use in your project, whether you use them or not.
  - The Project window works like a file explorer, organized in folders. You can drag assets directly from the Project window into the Scene view to add them to the scene.

Note: the difference between the Project and Hierarchy windows: the Hierarchy contains all the GameObjects in the current scene, and the Project window contains all the assets available to your entire project.

- The **Inspector** is where you'll find and configure detailed information about GameObjects.
- When you select a GameObject in Scene view or in the Hierarchy, you'll see its **components** in the Inspector. Components describe the properties and behaviors of GameObjects.
- Use the **toolbar** buttons to change your point of view in the scene, and start and stop Play Mode.

- The scene navigation functions are hosted in a floating toolbar in your scene view. These functions allow you to move, rotate and scale your selected GameObjects.

Scenes

- Projects in Unity Editor are organized into **scenes**.
- **Scenes** are containers for everything in the experience you are creating.
- One way to think about **scenes** is as discrete experiences.
  - For example, each level in a game could be a separate scene, and the game's main menu could be another.
- A Unity project can have one scene or more than a hundred, depending on its scope and complexity. There aren't strict rules about exactly how you should organize a Unity project into scenes, except that a project in Unity must have at least one scene.

To review the scenes in a Unity project:

- 1. In the Project window, use the navigation panel or search bar to find the **Scenes folder**
- 2. Double-click a scene to open it. You'll see the contents of that scene in the Hierarchy and Scene view.

Navigating Scene

One way to think about navigating in this window (scene view) is like operating a drone camera — it lets you examine your GameObjects from any angle or distance.
Basics:

- **Pan**: Select the Hand tool in the Toolbar, and click and drag in the Scene view to move your view.
- **Zoom**: Holding **Option**, **right-click** and **drag** in the Scene view to zoom.
- **Orbit**: Holding **Option**, **left-click** and **drag** to orbit around the current pivot point.
  - Note: this option is not available in 2D mode.
- **Focus** (**Frame Select**): When a GameObject is selected, select **F** with your cursor in the Scene view to focus your view on that GameObject.
  - Note: If your cursor is not in the Scene view, Frame Select will not work.

Flythrough mode: to navigate in the Scene view by flying around in first person

- Click and hold the **right mouse** button.
- Use **WASD** to move the view left/right/forward/backward.
- Use **Q** and **E** to move the view up and down.
- Select and hold **Shift** to move faster.

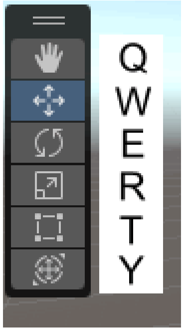
Note: flythrough mode → not available in 2D mode → Instead, holding the right mouse button down while moving the cursor pans around the Scene view.

Tools

- Effective use of the transform and scene navigation tools is an essential skill in Unity, which allows you to position, scale, and view your GameObjects efficiently in 3D space.
- **Note**: If you are in a scene without any GameObjects to practice with, right-click in the Hierarchy window and select **3D Object** > **Cube.**

Toolbar Keyboard Shortcuts: QWERTY:

Using these keys, you can switch quickly between the tools and keep your mouse in the Scene view.



Q: Hand tool, to pan your view 

W: Move tool, to select and change position 

E: Rotate tool, to select and rotate 

R: Scale tool, to select and change size 

T: Rect Transform tool, to scale in 2D 

Y: Transform tool, to move, scale, and rotate with one Gizmo 

- For each of the transform tools, a Gizmo appears that allows you to manipulate the GameObject along each specific axis. As you manipulate these controls, the values in the Transform Component change accordingly.
- You will discover your own system for navigating the Scene view and manipulating GameObjects efficiently. For example, your system might be to rest the fingers of your non-mouse hand on the QWER keys to change tools, rest your thumb on the **ALT** key to **orbit** your view of the scene, and move your index finger to the **F** key to **focus** on a GameObject as needed.

Manual:
https://docs.unity3d.com/Manual/SceneViewNavigation.html

Unity shortcuts reference

Scene view navigation:

- **View**: Hold right click + drag.

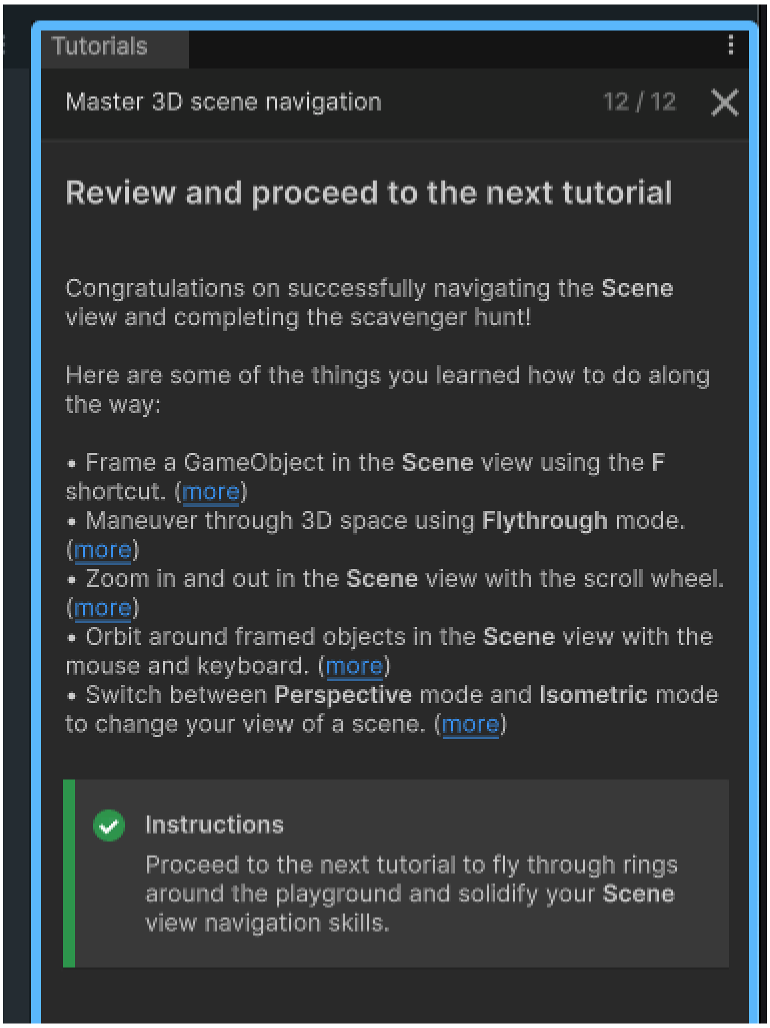
- **Frame**: Press F in the Scene view | or double-click a GameObject in the Hierarchy window.

- **Orbit**: Option + left click drag. (Rotates scene perspective)
- **Zoom**: Scroll | or: **hold Option** + **right click drag**
- **Flythrough mode:** right click + WASD + QE (↑↓)

Scene view tools shortcuts:
- View: **Q**
- Move: **W**
- Rotate: **E**
- Scale: **R**
- Rect: **T**
- Transform: **Y**

Miscellaneous shortcuts:
- Undo: Ctrl+Z (macOS: Cmd+Z)
- Save: Ctrl+S (macOS: Cmd+S)

2004

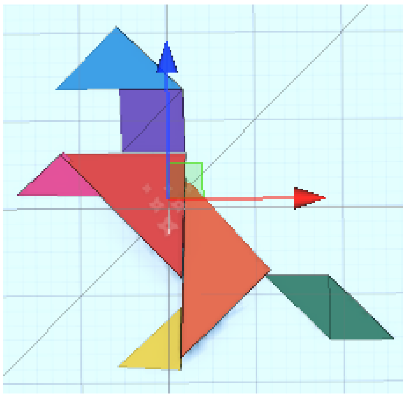


Move → with **Global** vs. **Local**

- The move tool  gives you access to use either **Global** or **Local** coordinates

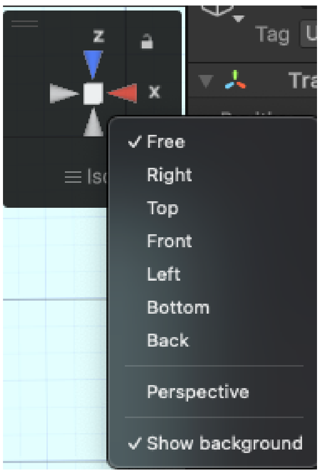
Global coordinates are kind of like north, south, east, and west, which are constant. Local coordinates are more like left and right, which change depending on which way you're facing.

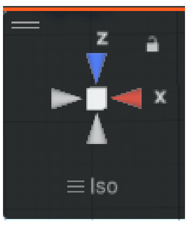
- Using move, the axes arrows change directions (depending on global vs. local)



Move → with **Perspective & Isometric mode**

With the move tool : you can also disable **Perspective** → changing your view to **Isometric mode**





- In **Isometric mode** you only see the Z-axis and X-axis
- You can then move GameObjects freely along these two axis

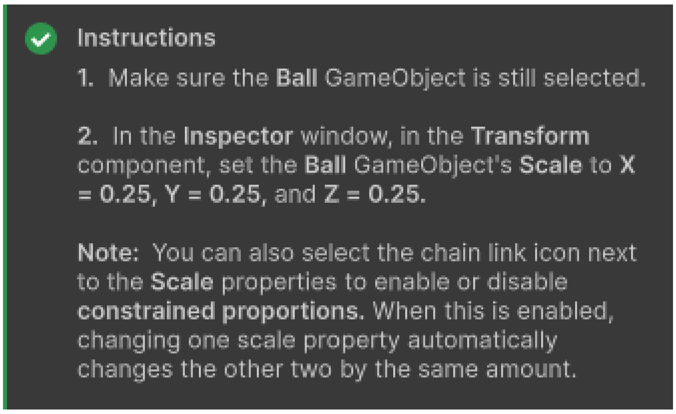
2. Right-click the **Scene** view gizmo and choose to disable **Perspective**.

This action will switch your view to Isometric mode. In this mode, notice that you can now only see the Z-axis (blue arrow pointing forward) and the X-axis (red arrow pointing to the right). You can no longer see the global Y-axis arrow.

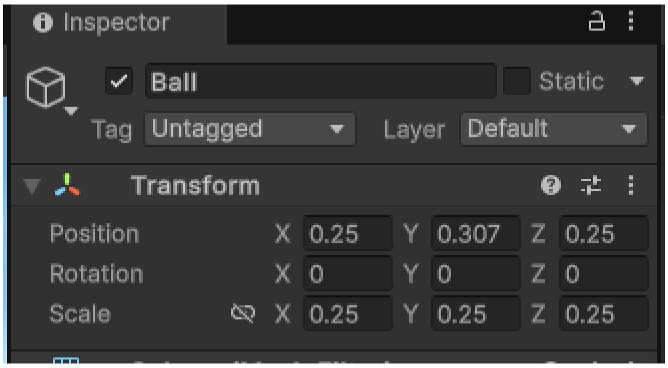
You'll notice you can now move the piece freely along these two axes, just as you would slide a puzzle piece across a table.

Scale

Scale: determines size of an object



- 1 unit (unity) = 1 meter (real world measurement)



Inspector → **Transform**

Each GameObject has **transform** (controls **position** [X, Y Z], **rotation**, **scale**)

Now you're going to position the ball high above the floor near the window so that it can fall to the ground and bounce toward the corner.

Rather than using the Transform tools in the **Scene** view to drag the ball around, you'll enter the ball's precise position directly in the **Inspector** window in the **Transform** component.

Every single GameObject in the scene has a **Transform** component, which controls its position, rotation, and scale.

The **Position** values are measured in meters along each of the 3 axes (X, Y, and Z) relative to the origin of the scene.

Instructions

1. With the **Ball** GameObject still selected, locate the **Transform** component in the **Inspector** window

2. Set its Transform position to **X = 2**, **Y = 3**, and **Z = -1**:

• X = 2 will position the ball 2 meters to the right of the origin on the X-axis.
• Y = 3 will position the ball 3 meters up from the origin on the Y-axis.
• Z = -1 will position the ball 1 meter back from the origin on the Z-axis.

Note: You can tell which direction is positive for each axis by looking at the **Scene** view gizmo. The colored cones point toward the positive direction for each axis.

Create & Apply **New Material** (to object) | Create new **custom asset**

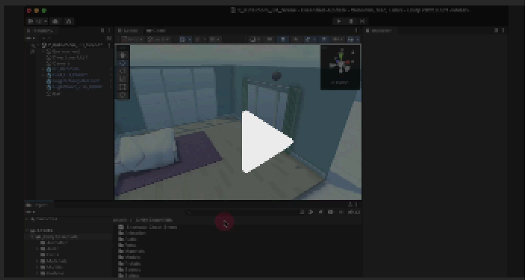
Create new material:

- In desired Project folder (e.g.: materials) → right click → Create → Material

Tutorials

Make a bouncy ball 5 / 12

Create and apply a new material



The sphere primitive you added for the ball has a default material that is a very dull gray color — if it's a kid's ball, it should be more exciting!

Materials define the visual appearance of objects in Unity. Here, you'll create a new material in your project's assets and apply it to the sphere.

Instructions

1. In the **Project** window, navigate to **_Unity Essentials** > **Materials.**

2. Right-click inside the **Materials** folder and select **Create** > **Folder**, and name it "My Materials".

   This is where you'll keep your custom materials.

3. Inside the new **My Materials** folder, right-click and select **Create** > **Material** then rename the new material "Ball_Mat" (short for "Ball Material").

   You have just created a new asset in the **Project** window, which means that this material can be used in any scene in your entire project. You've added a new custom asset! You can see from the **Create** menu when you right-clicked that you can make all kinds of assets, including scripts, animations, fonts, and more.

4. Drag the new **Ball_Mat** material directly onto the **Ball** GameObject in the **Scene** view, which should make it turn the default plain white material color.

Note: Remember to save your scene often with **Ctrl+S** (macOS: **Cmd+S**).

Adjust material

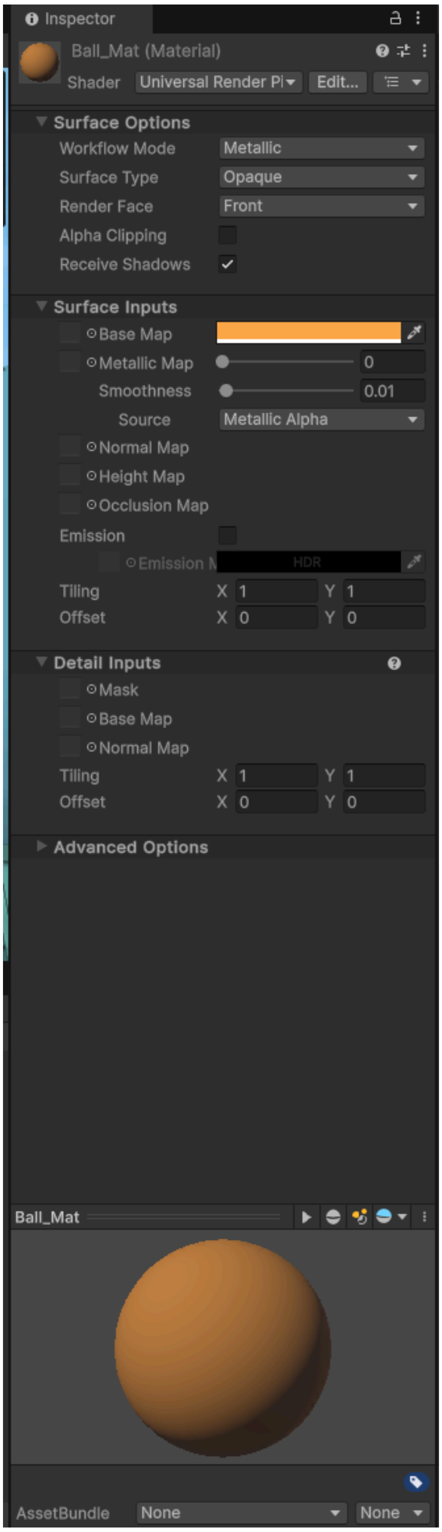
- **Instructions**

  1. Make sure that **Ball_Mat** material is selected in the **Project** window.

  2. In the **Inspector** window, in the **Surface Inputs** section, for the **Base Map** property, select the white color swatch and use the **Color** window to choose a new color.

     Observe how your changes affect the ball's appearance.

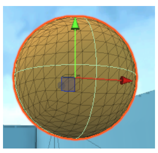
  3. Adjust the sliders for the **Metallic Map** and **Smoothness** properties to your liking.



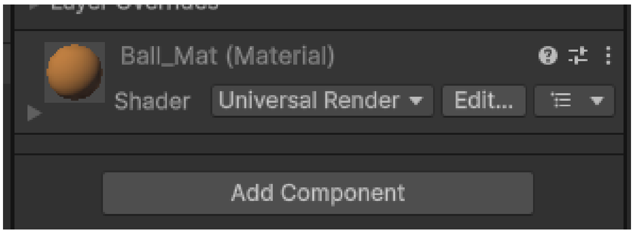
R

Adding **Rigidbody** to object → (**adds mass & gravity response**):

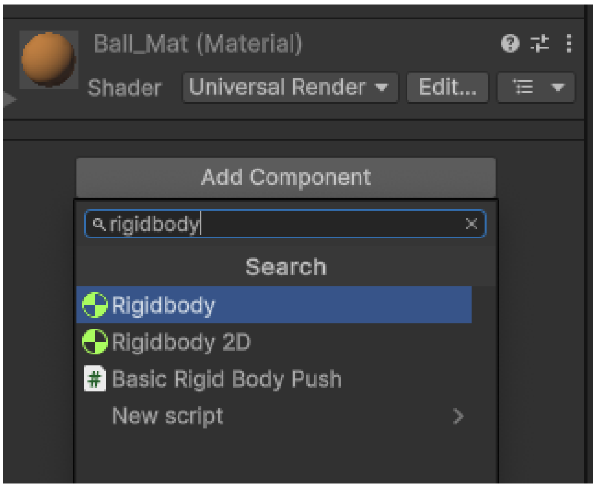
1. Select desired GameObject in Hierarchy window (or scene view)

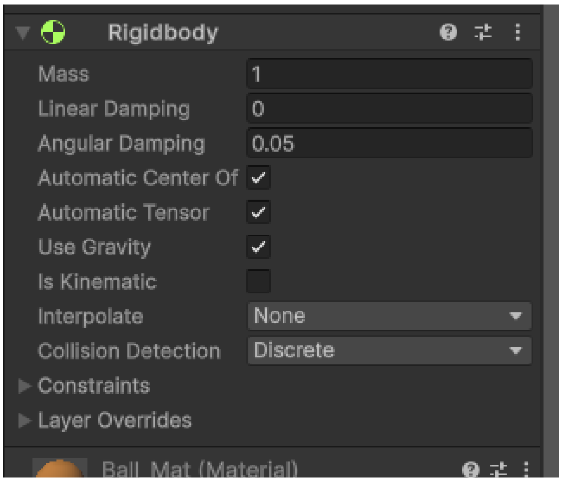


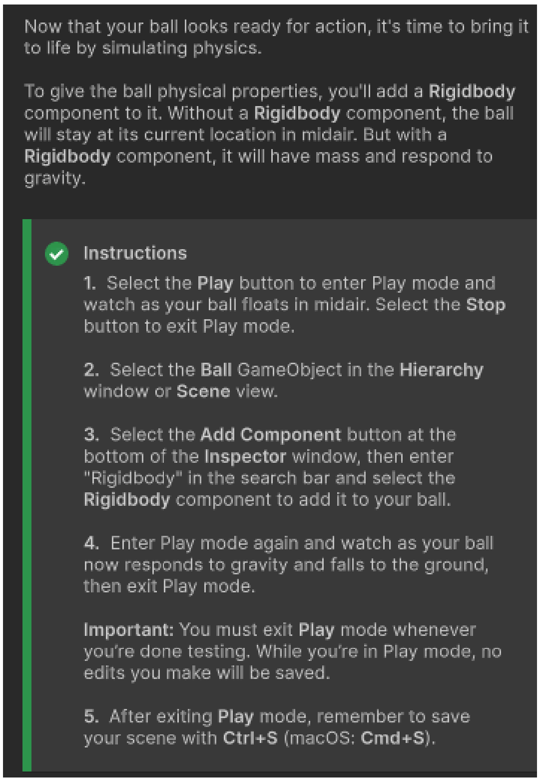
2. Select **Add Component** (inspector)



3. Search>select: “**Rigidbody**”







Now that your ball looks ready for action, it's time to bring it to life by simulating physics.

To give the ball physical properties, you'll add a **Rigidbody** component to it. Without a **Rigidbody** component, the ball will stay at its current location in midair. But with a **Rigidbody** component, it will have mass and respond to gravity.

Instructions

1. Select the **Play** button to enter Play mode and watch as your ball floats in midair. Select the **Stop** button to exit Play mode.

2. Select the **Ball** GameObject in the **Hierarchy** window or **Scene** view.

3. Select the **Add Component** button at the bottom of the **Inspector** window, then enter "Rigidbody" in the search bar and select the **Rigidbody** component to add it to your ball.

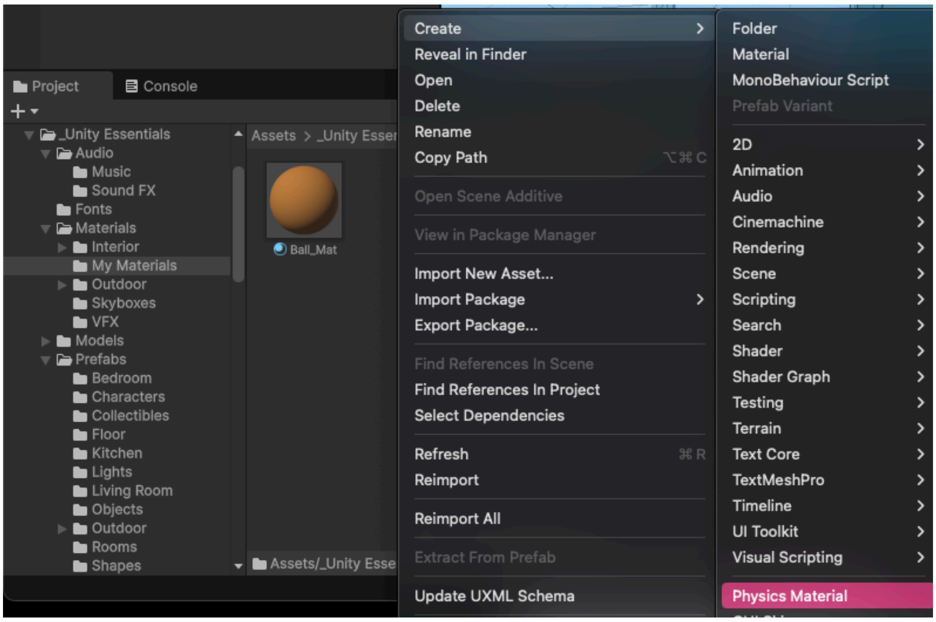
4. Enter Play mode again and watch as your ball now responds to gravity and falls to the ground, then exit Play mode.

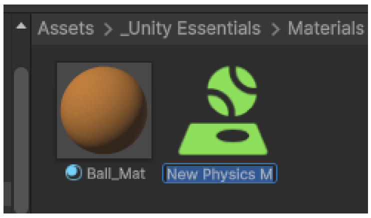
Important: You must exit **Play** mode whenever you're done testing. While you're in Play mode, no edits you make will be saved.

5. After exiting **Play** mode, remember to save your scene with **Ctrl+S** (macOS: **Cmd+S**).

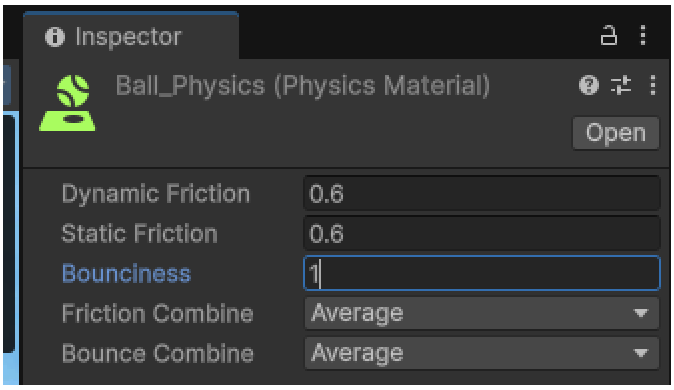
Create>Apply **Physics Material** (can make GameObject bounce)

- **Create** (right click in projects) → **Physics Material**

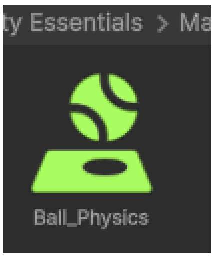


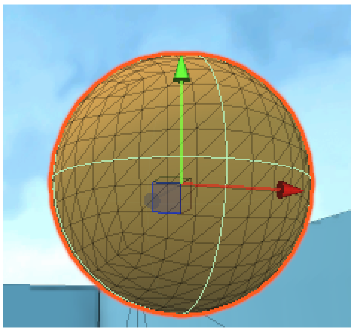


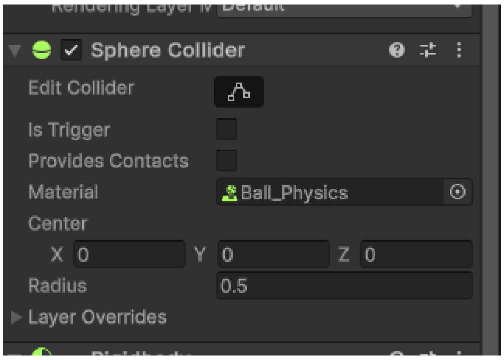
- Set **bounciness** to 1 (maximum) in inspectorO



- Drag and drop physics material to GameObject (in scene view)







Your bouncy ball's bounce is probably pretty disappointing. It behaves more like a bag of sand than a lively, bouncy ball.

To make it bounce, you'll create and apply a Physics Material. Unlike standard materials, which control the visual aspects of GameObjects, **Physics Materials** govern the physical behavior of GameObjects, like how they bounce upon impact.

Instructions

1. Right-click inside the **My Materials** folder, and select **Create** > **Physics Material.** Rename the new material "Ball_Physics".

2. In the **Inspector** window, set the **Bounciness** property of this Physics Material to **1** for maximum bounce effect.

3. Drag and drop the Physics Material onto the ball in the **Scene** view.

   You won't be able to tell from the **Scene** view that you've successfully applied the new Physics Material until you test it.

4. To observe the changes, select the **Play** button and watch as your ball bounces.

5. Remember to exit **Play** mode after you're done testing.

Review: Object Components (in inspector)

- **Transform**
- **Mesh Filter**
- **Mesh Renderer**
- **Sphere Collider**
- **Rigidbody**

You have customized this ball a lot at this point.

In this step, let's take a moment to review all of the components that contribute to the ball's behavior and appearance in Unity.

Instructions

1. Select the **Ball** GameObject in the **Hierarchy** window, then examine the **Inspector** window.

2. Locate the **Transform** component.

   The **Transform** component sets the position, rotation, and scale of the ball. This is the only component that is required for every GameObject.

3. Locate the **Mesh Filter** component.

   The **Mesh Filter** component determines the shape of your GameObject. A mesh is a wireframe 3D model. In the case of the ball, the Mesh is set to a Sphere shape.

4. Locate the **Mesh Renderer** component.

   The **Mesh Renderer** component controls the external appearance of the ball. Try to find the material you applied to the ball. When you apply a material, it shows up in this component.

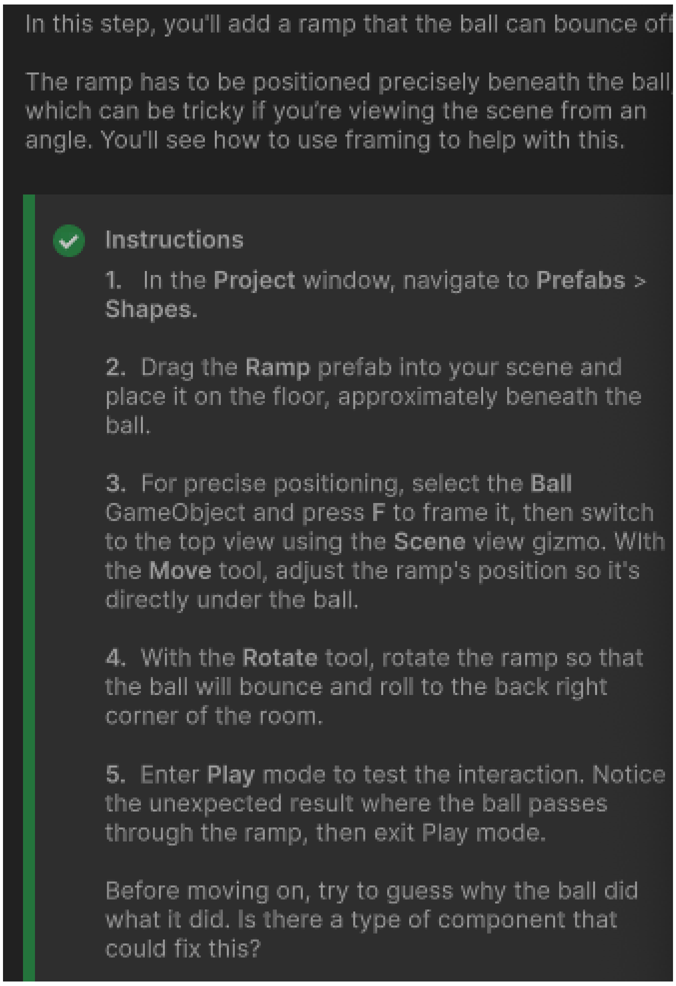
5. Locate the **Sphere Collider** component.

   The **Sphere Collider** component defines the physical boundaries of GameObjects for collision purposes. Try to find the **Physics Material** you created and applied.

6. Locate the **Rigidbody** component.

   The **Rigidbody** component integrates the ball into Unity's physics system, giving it properties like mass and drag.

Ball and Ramp Test Collision


In this step, you'll add a ramp that the ball can bounce off

The ramp has to be positioned precisely beneath the ball which can be tricky if you're viewing the scene from an angle. You'll see how to use framing to help with this.

Instructions

1. In the **Project** window, navigate to **Prefabs** > **Shapes.**

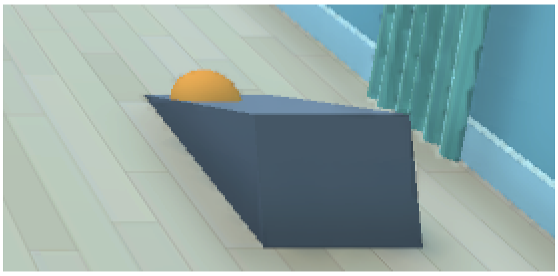
2. Drag the **Ramp** prefab into your scene and place it on the floor, approximately beneath the ball.

3. For precise positioning, select the **Ball** GameObject and press **F** to frame it, then switch to the top view using the **Scene** view gizmo. With the **Move** tool, adjust the ramp's position so it's directly under the ball.

4. With the **Rotate** tool, rotate the ramp so that the ball will bounce and roll to the back right corner of the room.

5. Enter **Play** mode to test the interaction. Notice the unexpected result where the ball passes through the ramp, then exit Play mode.

Before moving on, try to guess why the ball did what it did. Is there a type of component that could fix this?

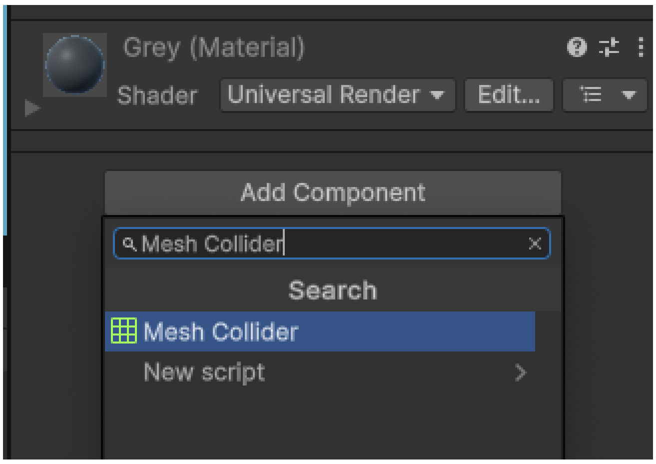


Result: the ball didn't react to the ramp (it fell inside it) → it needs physics

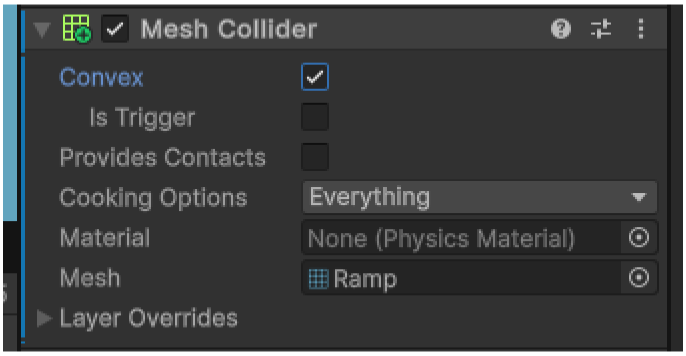
Solution: add **collider** (**Mesh collider**) component to ramp to fix it

Mesh Collider (essential for physics interactions)

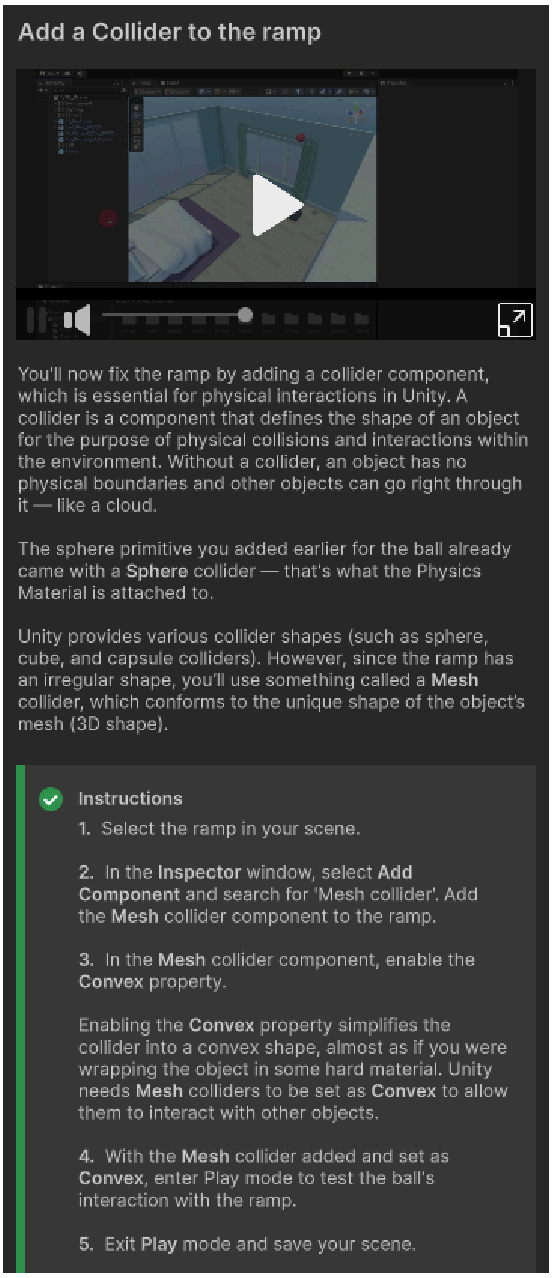
- Search & 'Add Component' (in inspector) → "**Mesh Collider**"



- Enable **Convex** property (makes object solid [wrapped in hard material])



-



Add a Collider to the ramp

You'll now fix the ramp by adding a collider component, which is essential for physical interactions in Unity. A collider is a component that defines the shape of an object for the purpose of physical collisions and interactions within the environment. Without a collider, an object has no physical boundaries and other objects can go right through it — like a cloud.

The sphere primitive you added earlier for the ball already came with a **Sphere** collider — that's what the Physics Material is attached to.

Unity provides various collider shapes (such as sphere, cube, and capsule colliders). However, since the ramp has an irregular shape, you'll use something called a **Mesh** collider, which conforms to the unique shape of the object's mesh (3D shape).

Instructions

1. Select the ramp in your scene.

2. In the **Inspector** window, select **Add Component** and search for 'Mesh collider'. Add the **Mesh** collider component to the ramp.

3. In the **Mesh** collider component, enable the **Convex** property.

   Enabling the **Convex** property simplifies the collider into a convex shape, almost as if you were wrapping the object in some hard material. Unity needs **Mesh** colliders to be set as **Convex** to allow them to interact with other objects.

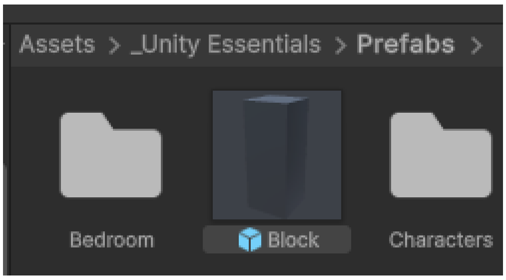
4. With the **Mesh** collider added and set as **Convex**, enter Play mode to test the ball's interaction with the ramp.

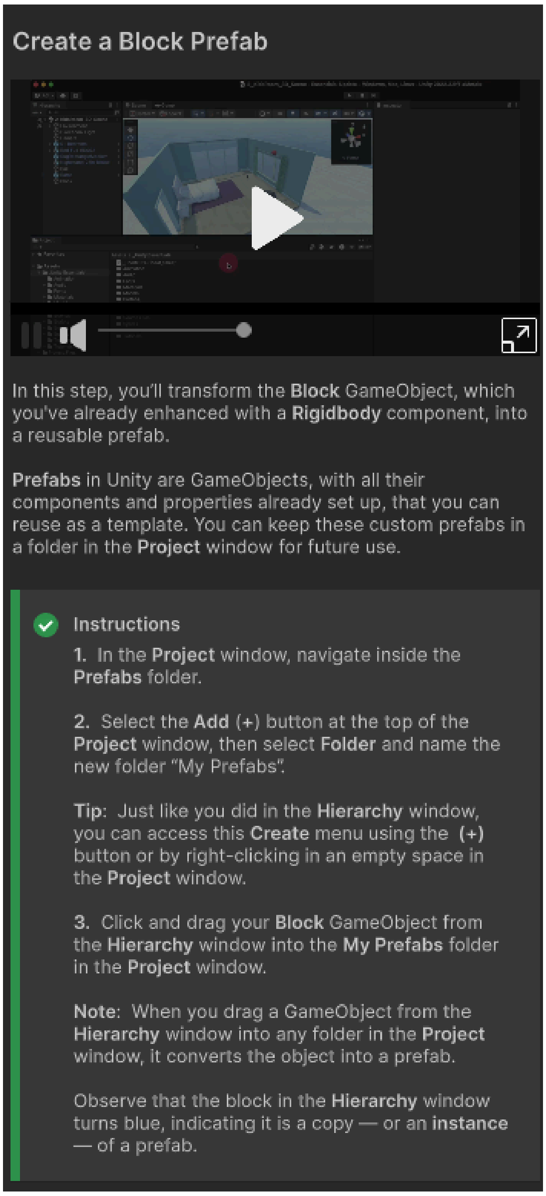
5. Exit **Play** mode and save your scene.

-

Prefabs: reusable gameBlocks (w/ components and properties setup Can be kept in a folder for later use

- Click and drag 

- 



Create a Block Prefab

In this step, you'll transform the **Block** GameObject, which you've already enhanced with a **Rigidbody** component, into a reusable prefab.

Prefabs in Unity are GameObjects, with all their components and properties already set up, that you can reuse as a template. You can keep these custom prefabs in a folder in the **Project** window for future use.

Instructions

1. In the **Project** window, navigate inside the **Prefabs** folder.

2. Select the **Add** (+) button at the top of the **Project** window, then select **Folder** and name the new folder "My Prefabs".

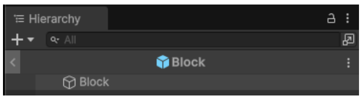
Tip: Just like you did in the **Hierarchy** window, you can access this **Create** menu using the (+) button or by right-clicking in an empty space in the **Project** window.

3. Click and drag your **Block** GameObject from the **Hierarchy** window into the **My Prefabs** folder in the **Project** window.

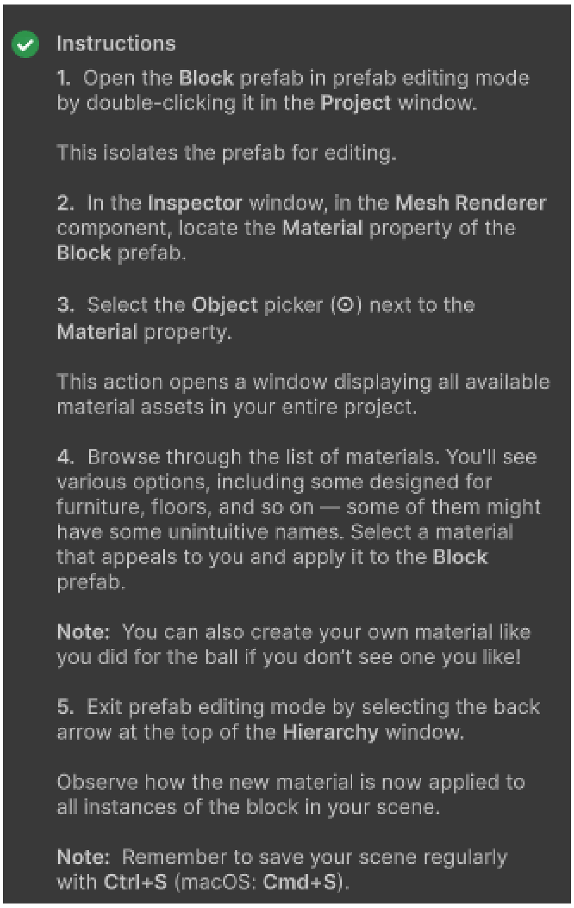
Note: When you drag a GameObject from the **Hierarchy** window into any folder in the **Project** window, it converts the object into a prefab.

Observe that the block in the **Hierarchy** window turns blue, indicating it is a copy — or an **instance** — of a prefab.

Editing prefabs later:

- To edit prefabs later → double click in project folder (e.g: weight)
- When done: press back arrow (in hierarchy) 

Add Material to Prefab:

- 

Instructions

1. Open the **Block** prefab in prefab editing mode by double-clicking it in the **Project** window.

This isolates the prefab for editing.

2. In the **Inspector** window, in the **Mesh Renderer** component, locate the **Material** property of the **Block** prefab.

3. Select the **Object** picker (⊙) next to the **Material** property.

This action opens a window displaying all available material assets in your entire project.

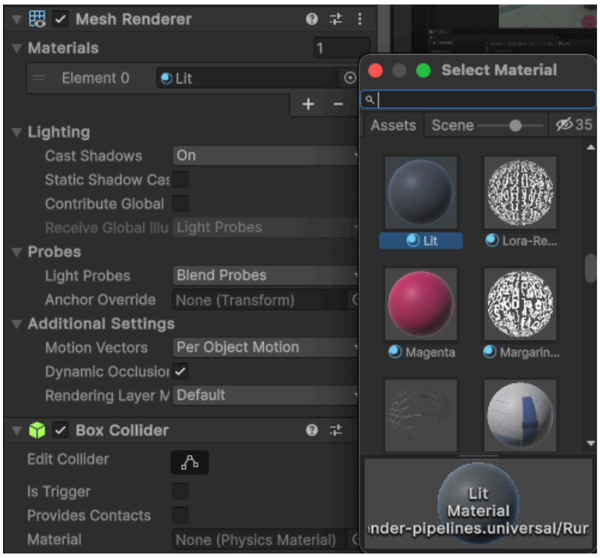
4. Browse through the list of materials. You'll see various options, including some designed for furniture, floors, and so on — some of them might have some unintuitive names. Select a material that appeals to you and apply it to the **Block** prefab.

Note: You can also create your own material like you did for the ball if you don't see one you like!

5. Exit prefab editing mode by selecting the back arrow at the top of the **Hierarchy** window.

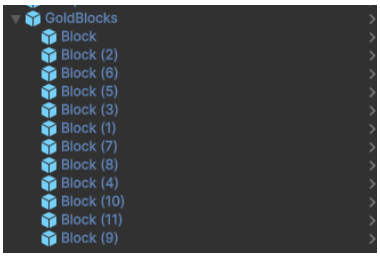
Observe how the new material is now applied to all instances of the block in your scene.

Note: Remember to save your scene regularly with **Ctrl+S** (macOS: **Cmd+S**).

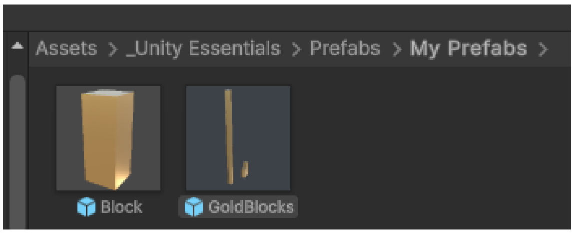
- 

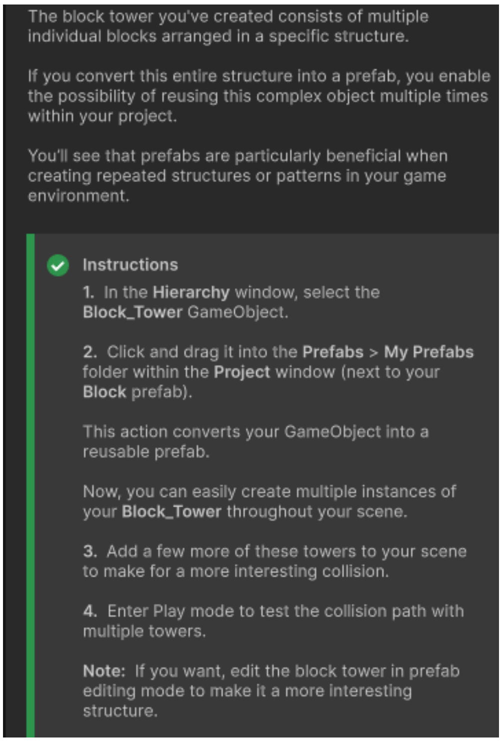
Save Prefab group

- Click & drag (from [parent] hierarchy → project folder)



→





The block tower you've created consists of multiple individual blocks arranged in a specific structure.

If you convert this entire structure into a prefab, you enable the possibility of reusing this complex object multiple times within your project.

You'll see that prefabs are particularly beneficial when creating repeated structures or patterns in your game environment.

Instructions

1. In the **Hierarchy** window, select the **Block_Tower** GameObject.

2. Click and drag it into the **Prefabs** > **My Prefabs** folder within the **Project** window (next to your **Block** prefab).

This action converts your GameObject into a reusable prefab.

Now, you can easily create multiple instances of your **Block_Tower** throughout your scene.

3. Add a few more of these towers to your scene to make for a more interesting collision.

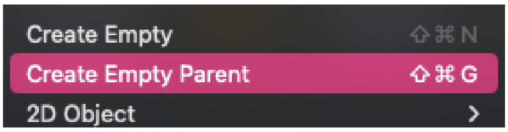
4. Enter Play mode to test the collision path with multiple towers.

Note: If you want, edit the block tower in prefab editing mode to make it a more interesting structure.

parent gameObject

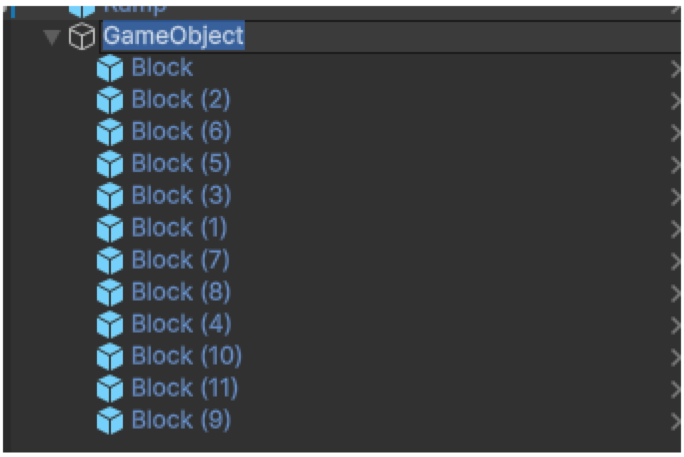
parent gameObject : To organize (many) blocks (gameObjects)

- Hold **shift** → click **first** and **last** gameObject
  - (or cmd + each extra gameObject)

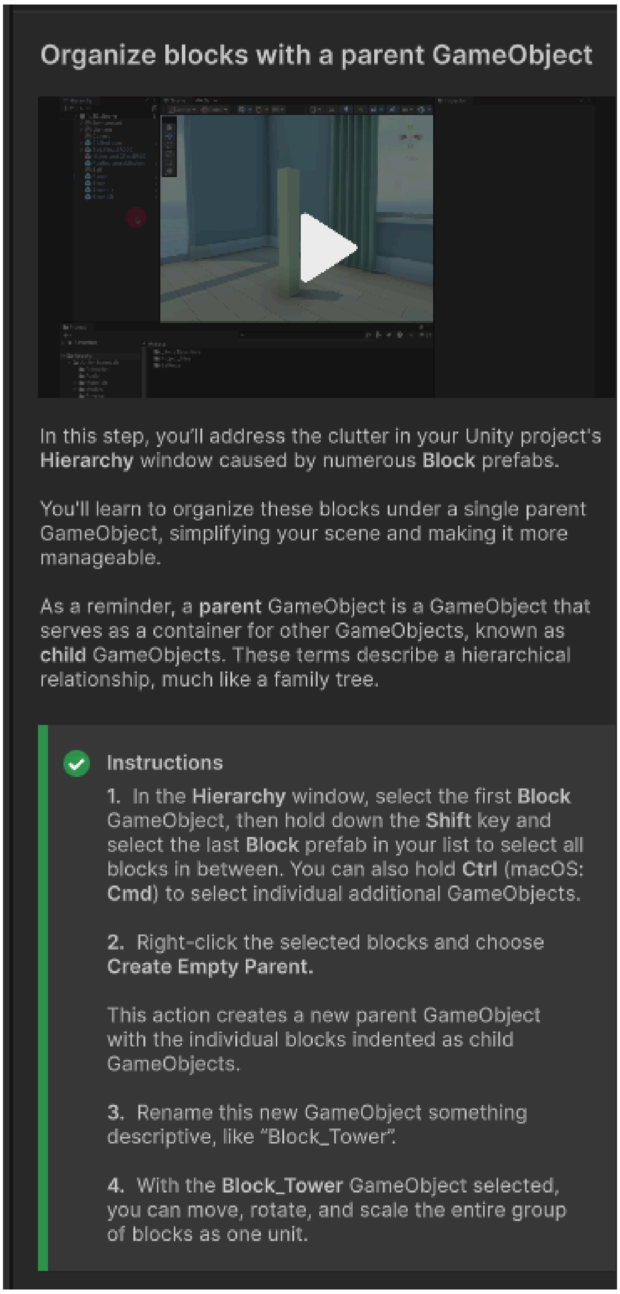


- Right click → **create empty parent** (↑ **cmd G**)
- Makes new parent → **"GameObjects"**, and

  Individual blocks indented as **child GameObjects rename**



- With the GameObject selected → you can **move, rotate, scale** the entire group of blocks as one unit



Organize blocks with a parent GameObject

In this step, you'll address the clutter in your Unity project's **Hierarchy** window caused by numerous **Block** prefabs.

You'll learn to organize these blocks under a single parent GameObject, simplifying your scene and making it more manageable.

As a reminder, a **parent** GameObject is a GameObject that serves as a container for other GameObjects, known as **child** GameObjects. These terms describe a hierarchical relationship, much like a family tree.

Instructions

1. In the **Hierarchy** window, select the first **Block** GameObject, then hold down the **Shift** key and select the last **Block** prefab in your list to select all blocks in between. You can also hold **Ctrl** (macOS: **Cmd**) to select individual additional GameObjects.

2. Right-click the selected blocks and choose **Create Empty Parent.**

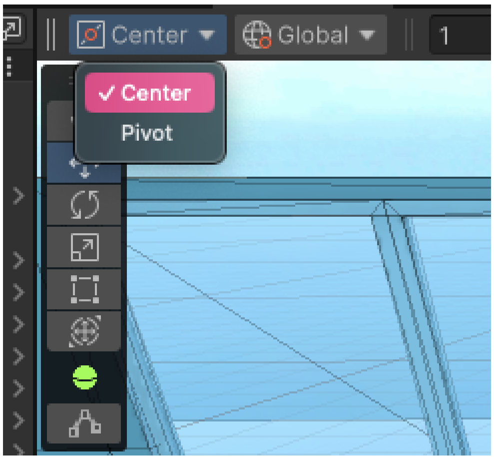
   This action creates a new parent GameObject with the individual blocks indented as child GameObjects.

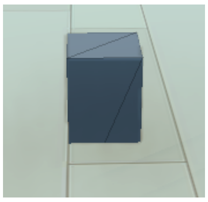
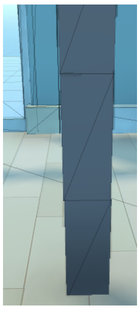
3. Rename this new GameObject something descriptive, like "Block_Tower".

4. With the **Block_Tower** GameObject selected, you can move, rotate, and scale the entire group of blocks as one unit.

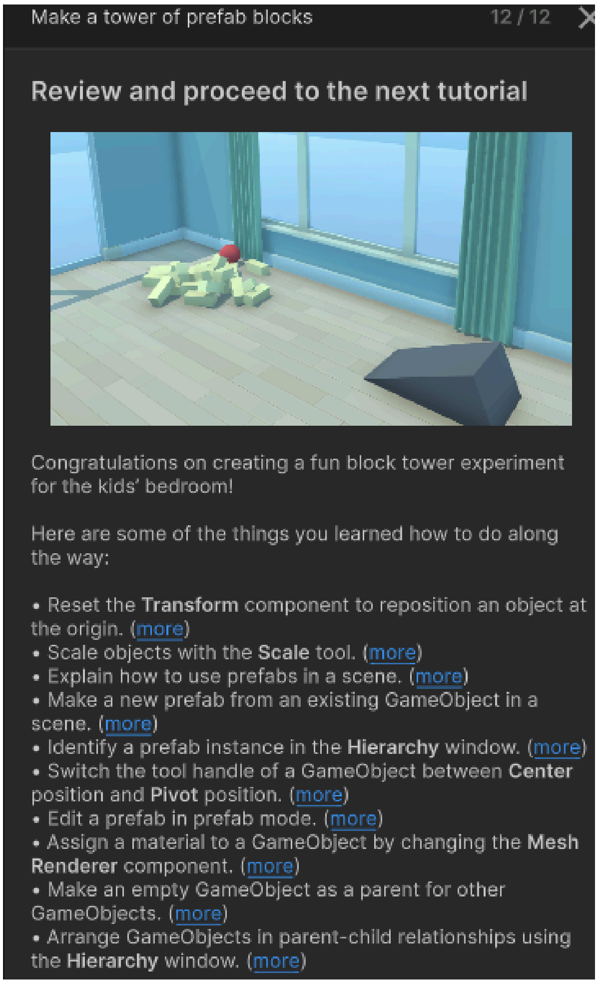
Center vs. Pivot

Useful when placing objects



- With **Pivot** → blocks end up halfway through the floor at their pivot point 
- With **Center:** they stack perfectly 

Review: Tower of prefab blocks

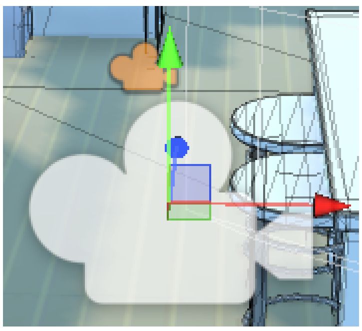


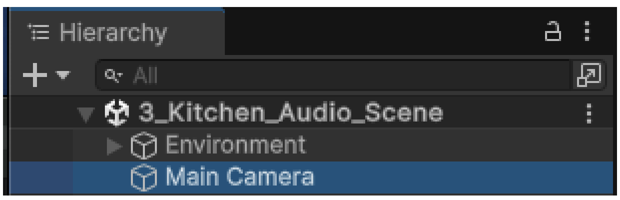
Audio

Create Immersive Soundscape:

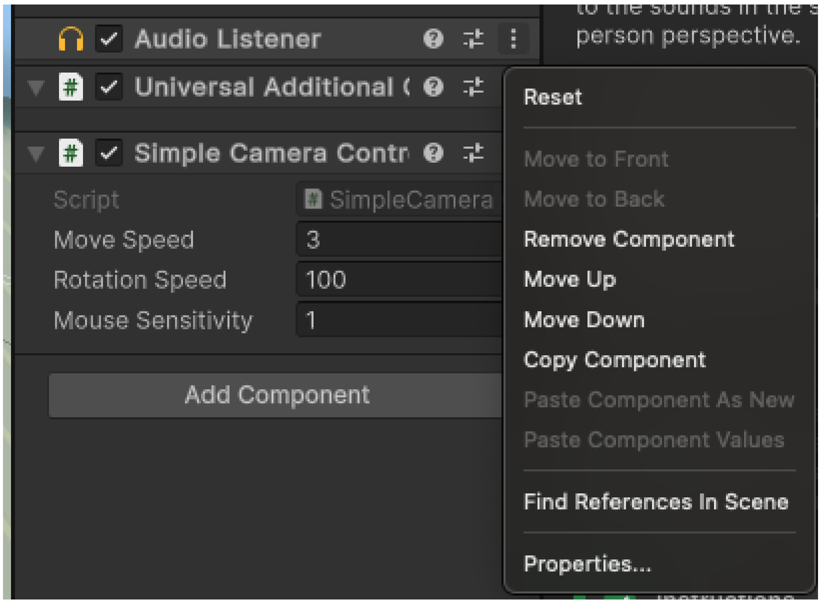
Audio Listener

Main Camera (hierarchy))



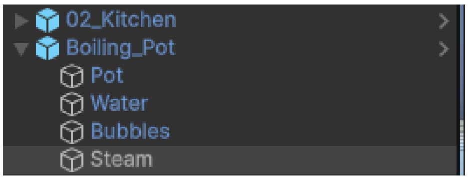


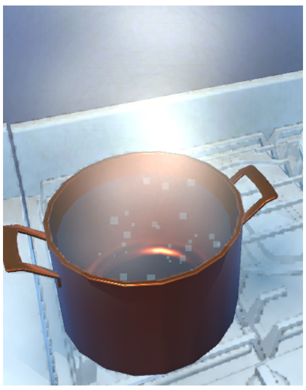
Main Camera → **Audio listener** (inspector)

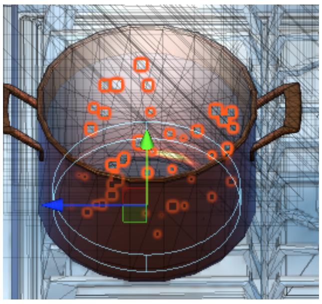


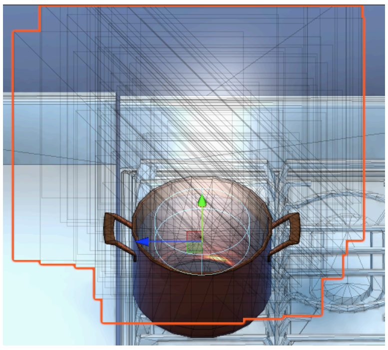
Particle System

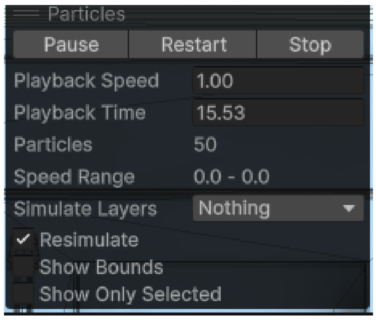
Particle System for pots steam and bubbles

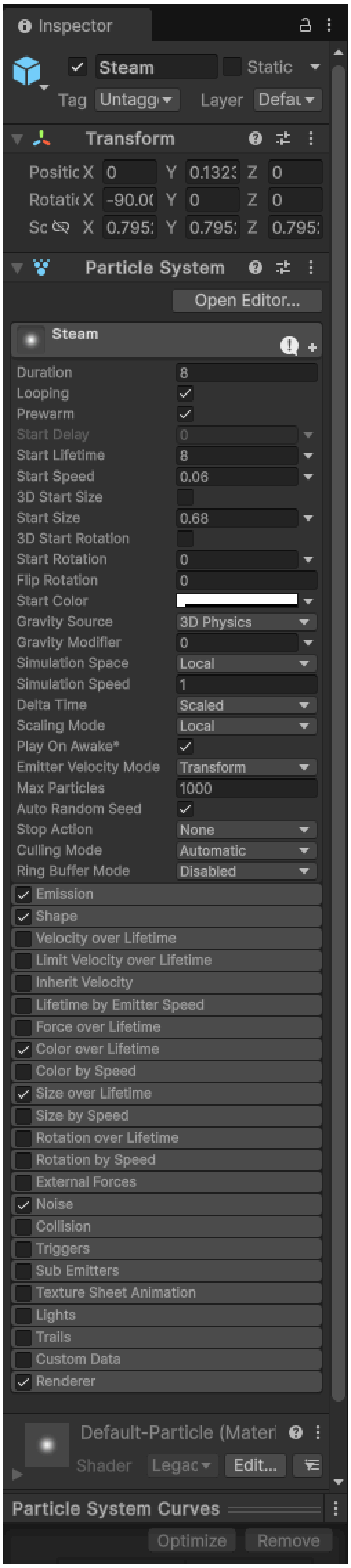






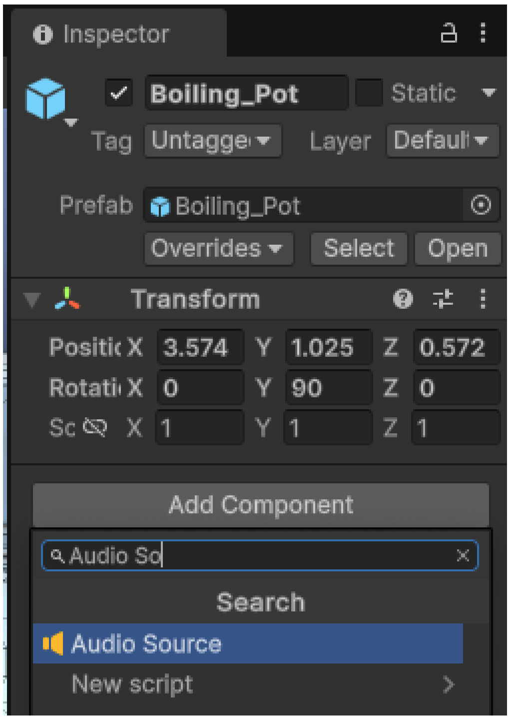




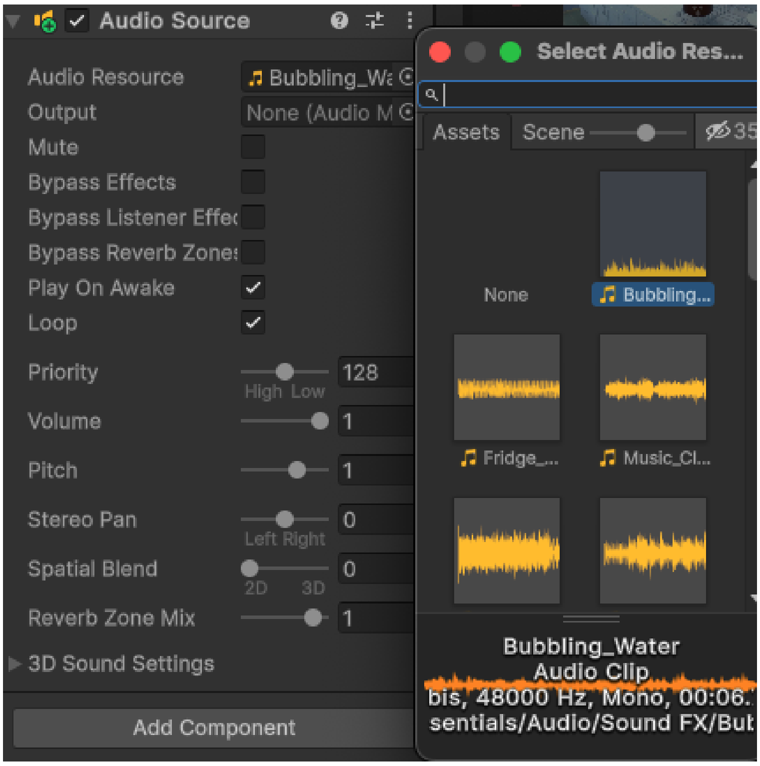


Audio Source (component)

Add Component (on object's Inspector) → **Audio source**

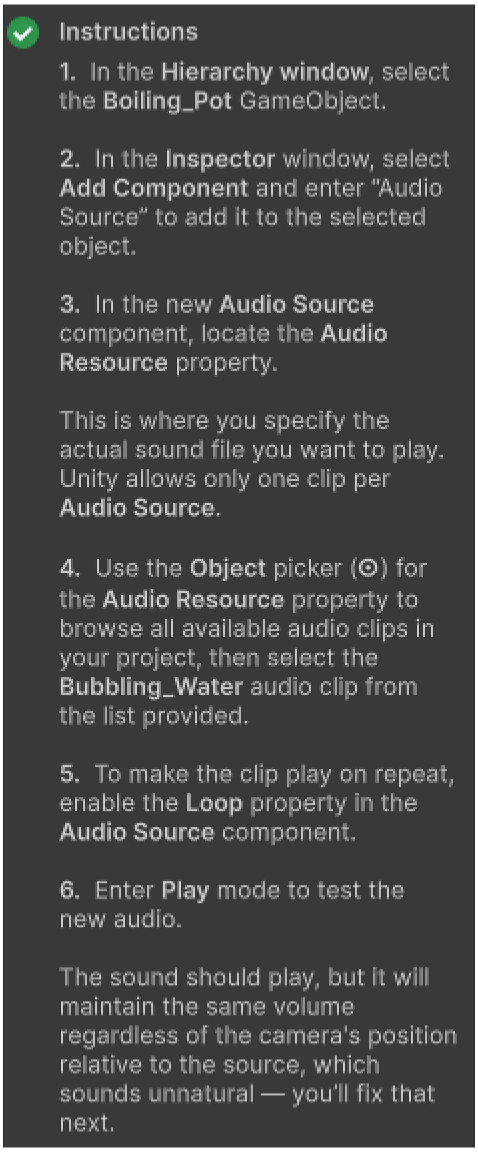


- **Audio Resource** property



    - Use Object picker  → **Audio Resource** property → to select audio clips to add
- **Loop** 

    - Loop → plays on repeat



Instructions

1. In the **Hierarchy window**, select the **Boiling_Pot** GameObject.

2. In the **Inspector** window, select **Add Component** and enter "Audio Source" to add it to the selected object.

3. In the new **Audio Source** component, locate the **Audio Resource** property.

This is where you specify the actual sound file you want to play. Unity allows only one clip per **Audio Source**.

4. Use the **Object** picker (⊙) for the **Audio Resource** property to browse all available audio clips in your project, then select the **Bubbling_Water** audio clip from the list provided.

5. To make the clip play on repeat, enable the **Loop** property in the **Audio Source** component.

6. Enter **Play** mode to test the new audio.

The sound should play, but it will maintain the same volume regardless of the camera's position relative to the source, which sounds unnatural — you'll fix that next.

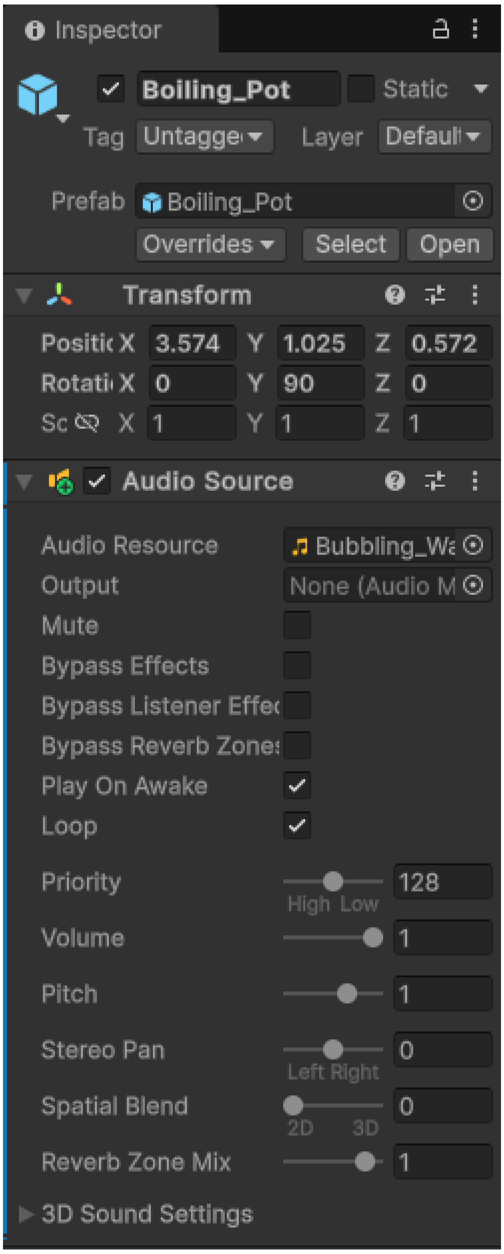
2D → **3D Audio Source**

Notes:

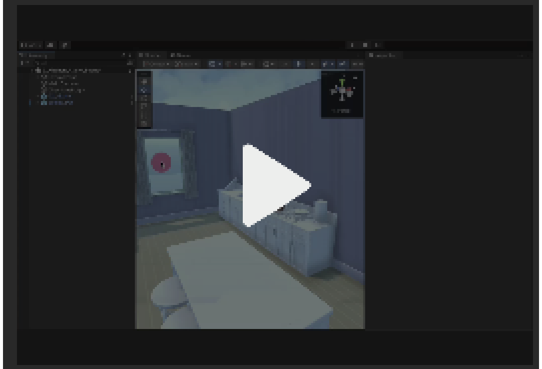
- *Default Unity scene audio → 2D (stays the same, regardless of player position/orientation in game)*
- ***3D Audio** = spatialized (sound changes [via listeners position, orientation relative to audio source)*
    - *Mimics real world sound behaviour*

Make audio clip 3D → Spatial Blend audio source property

- (Select GameObjects') → Audiosource (inspector) → **Spatial Blend** property
- Drag **Spatial Blend** property slider from **0** (**fully 2D**) → **1** (**fully 3D**)


Inspector
Boiling_Pot
Static
Tag Untagged
Layer Default
Prefab Boiling_Pot
Overrides
Select
Open
Transform
Position X 3.574 Y 1.025 Z 0.572
Rotation X 0 Y 90 Z 0
Scale X 1 Y 1 Z 1
Audio Source
Audio Resource Bubbling_Wa
Output None (Audio M
Mute
Bypass Effects
Bypass Listener Effec
Bypass Reverb Zones
Play On Awake
Loop
Priority 128
High Low
Volume 1
Pitch 1
Stereo Pan 0
Left Right
Spatial Blend 0
2D 3D
Reverb Zone Mix 1
3D Sound Settings

Make it a 3D Audio Source



In real life, sounds change in volume in each ear based on your distance and direction relative to the sound source.

By default, audio in a Unity scene is **2D Audio,** which means that it stays the same regardless of the listener's position or orientation in the game world.

3D Audio, on the other hand, is **spatialized:** the sound changes based on the listener's position and orientation relative to the **Audio Source**, mimicking how sound behaves in the real world.

Let's make the Boiling_Pot audio clip a 3D experience!

Instructions

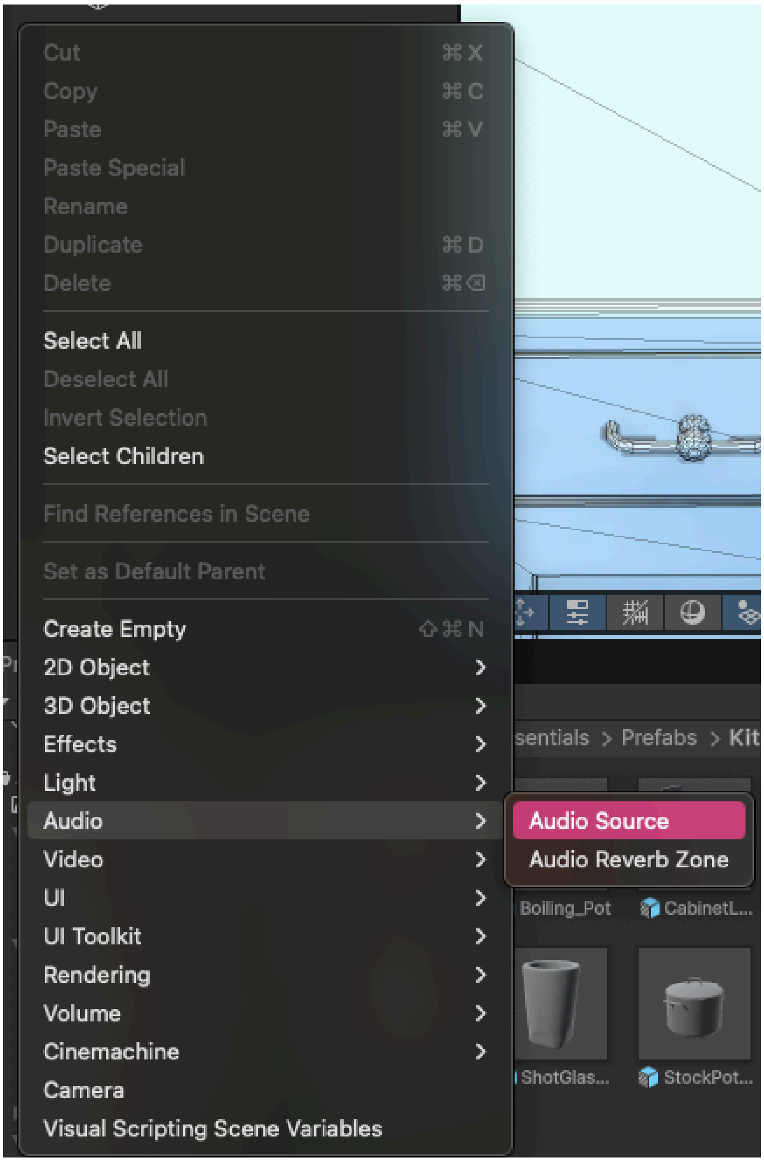
1. In the **Boiling Pot** GameObject's **Audio Source** component, locate the **Spatial Blend** property and drag the slider from **0** (fully 2D) to **1** (fully 3D).

2. Enter **Play** mode again to test your new 3D spatial audio effect. Move closer and farther away, and be sure to experience the stereo effect by rotating your view left and right relative to the boiling pot.

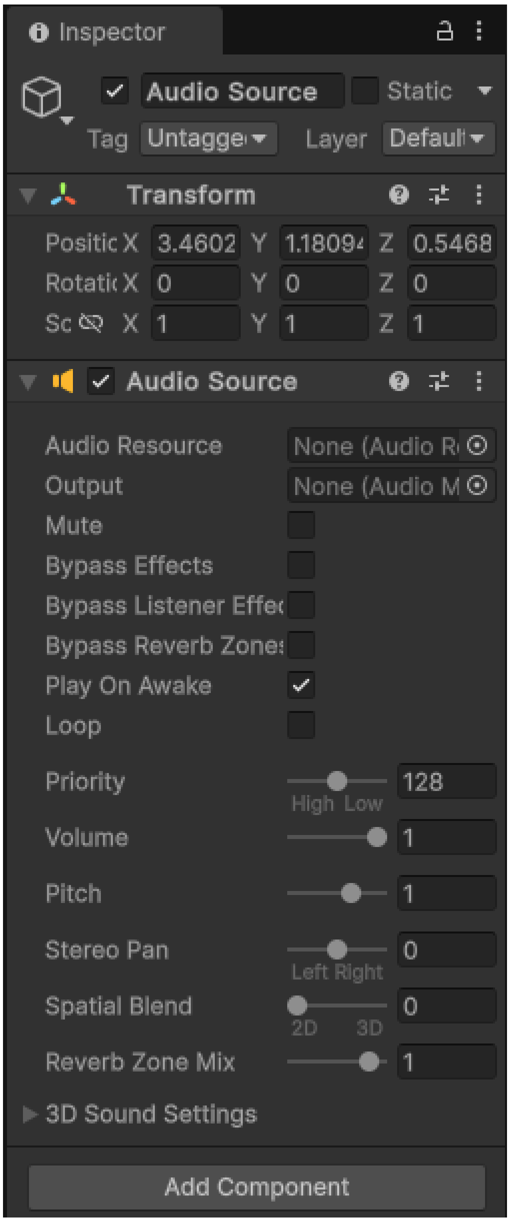
3. Exit **Play** mode.

Add background Music

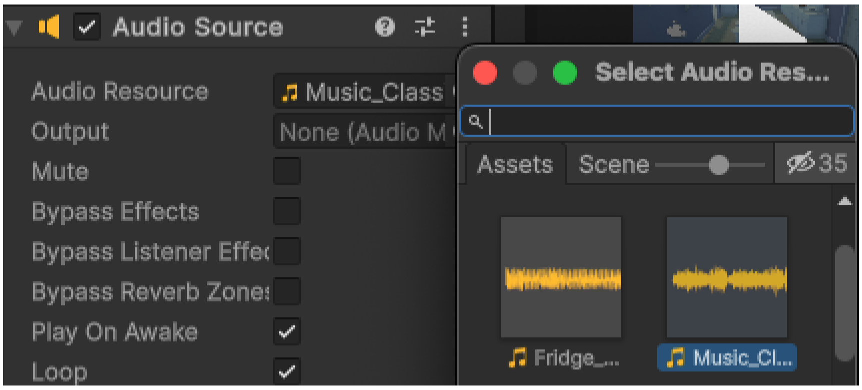
- (from Hierarchy) → Right click → **Audio** → **Audio Source**



- Creates new GameObject called "Audio Source" (with Transform and Audio Source components only)

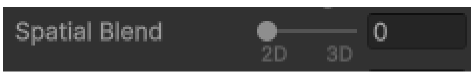


- (in Inspect → Audio Source) **Set Audio Resource**



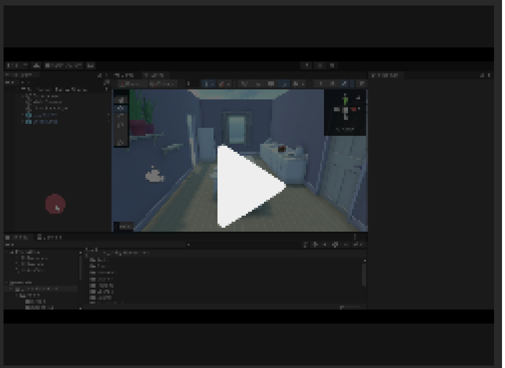
- Enable **Loop**

- Keep it (bg music) **2D** for spatial consistency



-

Add background music



In this step you'll add background music, which will set the emotional tone of your scene. The track you select can completely alter the ambiance of the scene, whether it's a calm classical piece, something more upbeat, or even something a little spooky.

The boiling pot is meant to mimic real-world 3D sound coming from the environment. Background music, on the other hand, isn't produced by anything in the environment and is purely used to enhance the overall atmosphere of the scene.

Let's set a mood in this scene with some background music.

Instructions

1. In the **Hierarchy** window, right-click in the empty space and select **Audio > Audio Source.**

This will create a new GameObject that only has a **Transform** component and an **Audio Source** component.

2. Rename the GameObject "Background Music" for clarity.

3. In the **Project** window, browse the available music options in the **_Unity Essentials > Audio > Music** folder.

You can preview the selected music in the **Preview** panel at the bottom of the **Inspector** window.

4. In the **Background Music** GameObject's **Audio Source** component, set the **Audio Resource** to your desired music, then enable **Loop.**

5. Adjust the **Audio Source's Volume** property to a more subtle level, like **0.1 - 0.5,** to ensure the background music doesn't overpower other sound in the scene.

6. Keep the **Spatial Blend** of this **Audio Source** component as the default 2D so that the background music's presence is consistent throughout the scene without positional changes.

7. Enter **Play** mode to test your background music and how it interacts with the sound of the boiling pot.

You may want to change the volume of the boiling pot or the music to get the balance just right.

Note: Remember to save your scene often with **Ctrl+S** (macOS: **Cmd+S**).

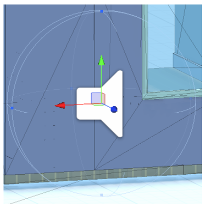
Audio Source & Custom Script: Add random bird sounds outside

Notes:

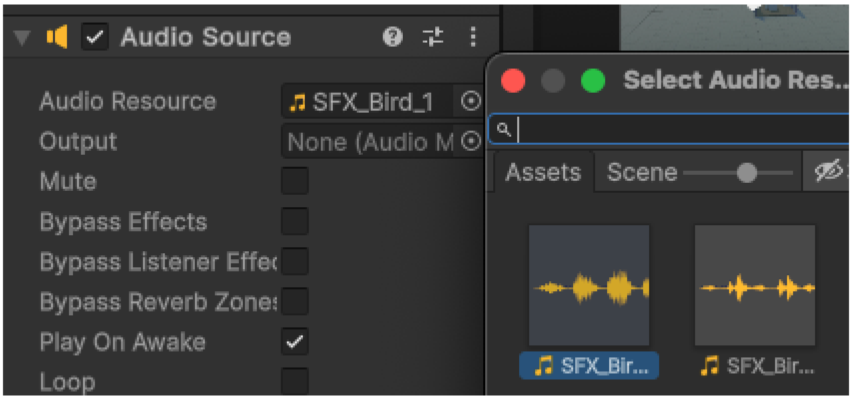
- *Position **Audio Sources** outside kitchen*
- *Use **custom script** to play bird chirping randomly*

Steps:

- Create new **Audio Source** in scene → position it where sound should originate (e.g., outside house)

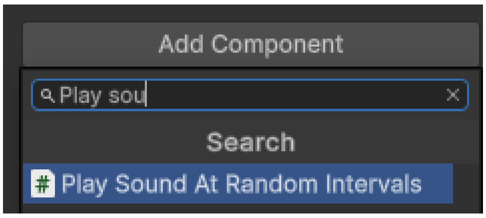


- Position Audio Source (where sound should originate [e.g., outside house])
  - *rename audio source*
- Select sound → **Audio Resource**



- Make sound 3D → Set **Spatial Blend** → **3D** (**1**)

- Disable '**Play on Awake**' property (disables sound playing right away)
- **Add Component** (inspector) → '**Play Sound at Random Intervals**'

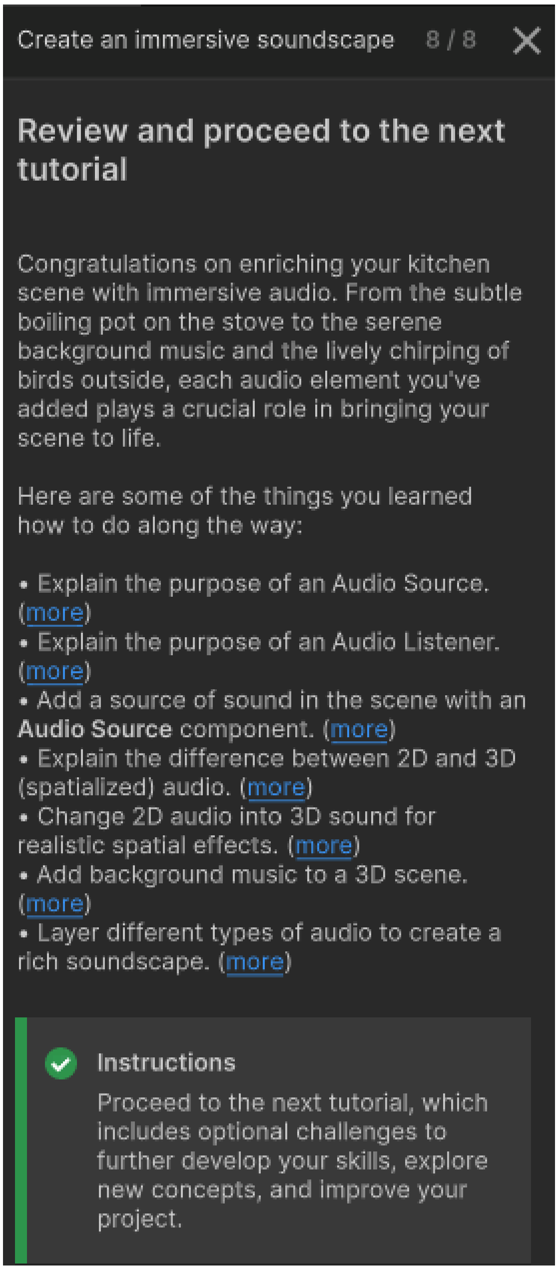


  - *script e.g:*

```
public class PlaySoundAtRandomIntervals   MonoBehaviour
{
    public float minSeconds = 5f; // Minimum interval to wait before playing sound.
    public float maxSeconds = 15f; // Maximum interval to wait before playing sound.

    private AudioSource audioSource;

    private void Start()
    {
        audioSource = GetComponent<AudioSource>();
        StartCoroutine(PlaySound());
    }
```

```
    private IEnumerator PlaySound()
    {
        while (true)
        {
            float waitTime = Random.Range(minSeconds, maxSeconds);
            yield return new WaitForSeconds(waitTime);
            audioSource.Play();
        }
    }
}
```

Review: Soundscape

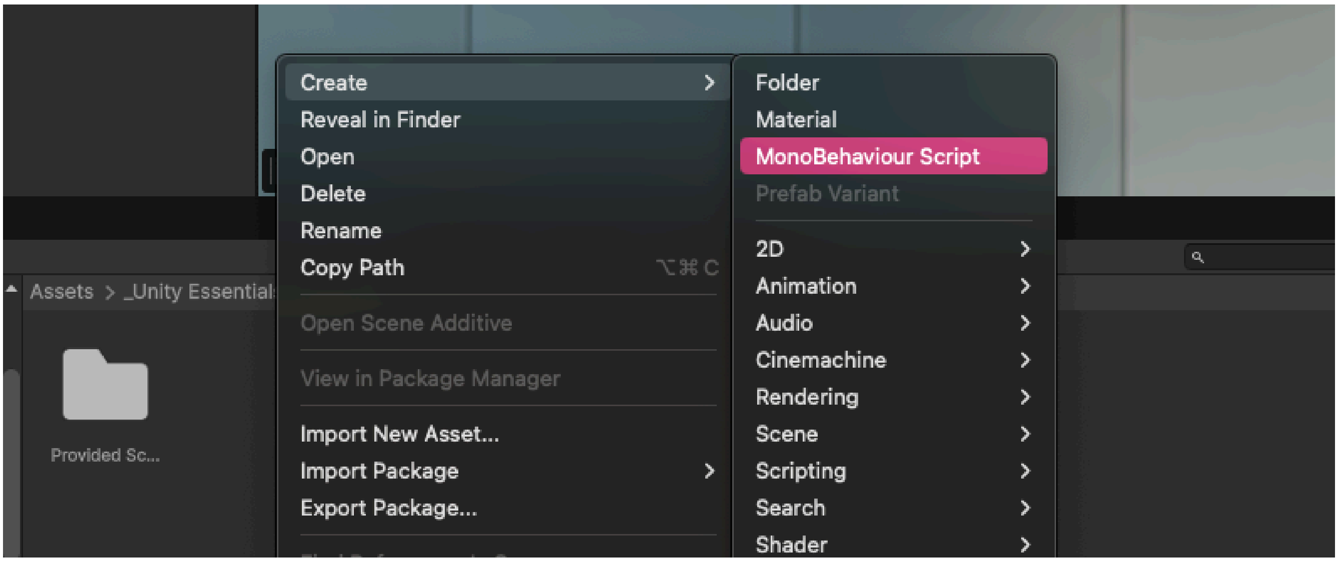


C# Script

Add Movement Script

Create "PlayerController" Script

- Create → Monobehaviour Script

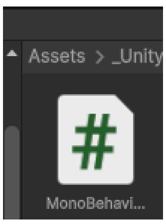


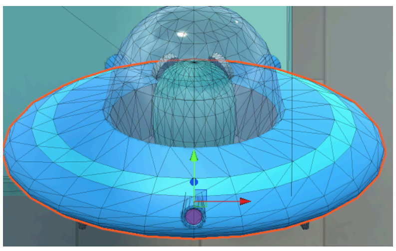
- Rename → "MonoBehaviourScript"

Note: If you make typo in script name and have to rename it, it will be easier to delete and create new script

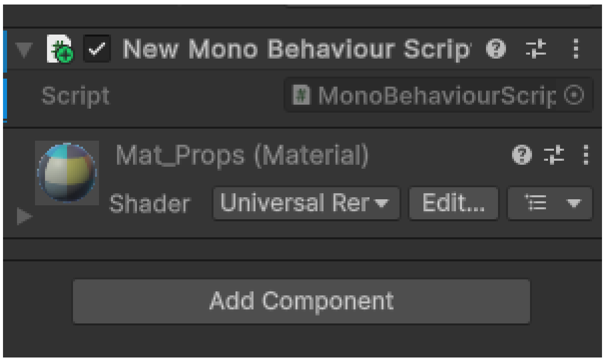
Apply Script to Player GameObject:

- Drag created script (from Project window) → player



→


- The script will show up on player inspector



Open script in Rider (by double clicking script) (cannot edit script in Unity):

- For script editing

Edit Player Properties in Play Mode:

- Enter **Play mode** → (In inspector) → adjust **Speed**, and/or **Rotation Speed** → note values if needed (they disappear)

- Change **properties** (speed, rotation speed) if needed

Note:

- Adding scripts (Unity Manual):
- https://docs.unity3d.com/6000.0/Documentation/Manual/creating-scripts.html

Getting started with scripting (Unity Manual):
https://docs.unity3d.com/6000.0/Documentation/Manual/scripting-get-started.html

Topic	Description
Introduction to scripting	Understand the role of scripting in your project and how it works at a high level.
Creating scripts	Create scripts to add programmable functionality to your application.
Naming scripts	Name your scripts to help the compiler identify your classes. Use namespaces to logically organize your scripts and prevent class name conflicts.
Inspecting scripts	View and edit the fields of your C# scripts in the **Inspector** window.
Fundamental Unity types	Some of the fundamental built-in Unity C# types, which you'll need to inherit from or use in your own code.

Script

Script Structure

Default Script Structure

```
// Using statements: link script to necessary libraries (so when you enter something like
RigidBody, script knows what that means)
using UnityEngine;

// class declaration:
// public class named "Collectible", that inherits unity's MonoBehaviour class
(MonoBehaviour is required class for all gameObjects in Unity)
public class Collectible : MonoBehaviour
{
   // Start is called once before the first execution of Update after the MonoBehaviour
is created
   void Start()
   {

   }

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

   }
}
```

- **MonoBehaviour:** required class for all GameObjects → allows you to add your script as a component to a GameObject
- **Start**() function: executed once at beginning of game
- **Update**() function: runs once per frame
    - Code within this function will run 30-60 fps → suitable for actions and animations (e.g: rotation code)

Adding Transform Rotation code

transform.Rotate(0, 1, 0);

```
   // Update is called once per frame
   void Update()
   {
       transform.Rotate(0, 1, 0);
   }
}
```

It's time to add continuous rotation from your **Collectible** script. You'll do this by manipulating the object's **Transform** component, which controls the object's position, rotation, and scale in the scene.

Specifically, you'll use the **transform.Rotate** method, a built-in Unity function that makes an object rotate in three dimensional space. For a simple rotation effect, you'll change the Y value, which makes the object rotate around its vertical axis.

When you put this rotation code in the **Update** function, the collectible will rotate a little bit each frame, making it appear to smoothly, continuously rotate.

Instructions

1. Open the **Collectible** script in the script editor.

2. Put your cursor inside the **Update()** method, between the sets of curly brackets.

It can be helpful to add a few new lines using the **Enter** (macOS: **Return**) key to give yourself a bit of space before you add your new code.

The extra space won't change the functionality of the code, but it can make your code easier to read.

3. Write the following line of code inside the **Update** function, between the opening and closing curly brackets:

```
transform.Rotate(0, 1, 0);
```

Important: Don't forget to add the semicolon at the end of the line of code.

This line instructs Unity to rotate the collectible object 1 degree around the Y-axis in each frame, creating a continuous rotation effect.

4. Save the script by pressing **Ctrl+S** (macOS: **Cmd+S**) or by selecting **File** > **Save** from the main menu of the script editor.

5. Back in the Unity Editor, enter Play mode to see the collectible object rotating continuously.

Tip: During **Play** mode, you can switch to the **Scene** view tab, which will allow you to get a close-up view of your collectible as the application is running.

6. Remember to exit **Play** mode when you're done testing!

Float Value (decimal #) in C#

Decimal numbers → "float values" (floating point number; data type that handles decimal numbers)

Float values (decimal #): require indication it's a float → put "f" at the end of the number

- *e.g: transform.Rotate(0, **0.5f**, 0);*

```
Event function
void Update()
{
    transform.Rotate(xAngle: 0, yAngle: 0.5f, zAngle: 0);
}
```

Data types: specify type of value and how computer integrates the value

- *E.g: integers for whole numbers; floats for decimal numbers; strings for text*

You might notice that the collectible rotates really quickly. However, if you try to reduce the value from **1** to **0.5** in your script, you'll get an error in your code, indicated by a red underline.

In C#, decimal numbers are treated as **float values.**
A float, short for "floating point number", is a data type that can handle decimals.

Data types in programming specify the type of a value and how the computer interprets this value. For instance, integers for whole numbers, floats for decimal numbers, and strings for text.

When you use float values in C#, you need to explicitly indicate it's a float by putting an "f" at the end of the number.

Instructions

1. Reopen the **Collectible** script in your script editor.

2. Modify the rotation line in the **Update()** method to **transform.Rotate(0, 0.5f, 0);**.

   This change reduces the rotation speed to half a degree per frame.

   When entering the decimal value **0.5,** make sure to add "f" at the end (0.5f). Try temporarily removing the "f" to see the error in your code.

3. Save the script and return to the Unity Editor. Play your scene to observe the slower rotation of the collectible.

declare **Variables: public float**

Add public variable to script

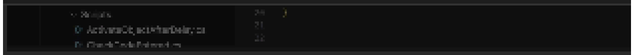
- To add variable data
    - Can be adjusted in inspector (without modifying script)

<u>E.g: Declare new rotation speed variable:</u>

- Declare new public float variable before **Start** method by adding line:
    - **public float** rotationSpeed;
    - E.g.:

```
public class Collectible : MonoBehaviour
{

    public float rotationSpeed;

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.Rotate(0, rotationSpeed, 0);
    }
}
```

Create a rotating collectible 8 / 9



In this step, you will enhance the flexibility of the rotation behavior by introducing a **public variable** to your script.

A variable in programming is just a named piece of data that can vary. When you declare a public variable for rotation speed, you'll allow yourself, and anyone else who works on this project in Unity, to easily adjust the rotation rate directly from the **Inspector** window without the need to modify the script.

This practice is an example of efficient game programming.

Instructions

1. Open the **Collectible** script in your script editor.

2. Above the **Start()** method, but after the first opening curly bracket of your script, use the **Enter** (macOS: **Return**) key to add a few empty lines to make some space for yourself to type.

Important: Pay close attention to the demo video to see where to put this code. Putting it in the wrong section of your script will cause errors.

3. Declare a new public float variable before the **Start** method by adding the line:

```
public float rotationSpeed;
```

This creates a public variable named **rotationSpeed** of type float. The word "public" (as opposed to "private") makes this value editable in the **Inspector** window.

Using a lowercase letter for the first word, but uppercase letters at the start of all subsequent words is known as **camelCase,** and is a common practice when naming variables. It's called camelCase because it kind of looks like a camel from the side, with humps at the center of the word.

4. Now, replace the hard-coded rotation value in the **Update()** method with this new variable you just created:

```
transform.Rotate(0, rotationSpeed, 0);
```

5. Save the script and switch back to the Unity Editor. In the **Inspector** window, you'll now see the **rotationSpeed** variable in the **Inspector** window when you select the collectible object, which is set to zero by default. You'll also notice that Unity has taken your camelCase **rotationSpeed** variable and changed it to **Rotation Speed** in the **Inspector** window for improved readability.

6. In the **Inspector** window, set the **Rotation Speed** value to a number like **0.5** to control the speed of rotation. Edit the value to the rotation speed you like the best.