

# RED BELT



**CODE NINJAS®**

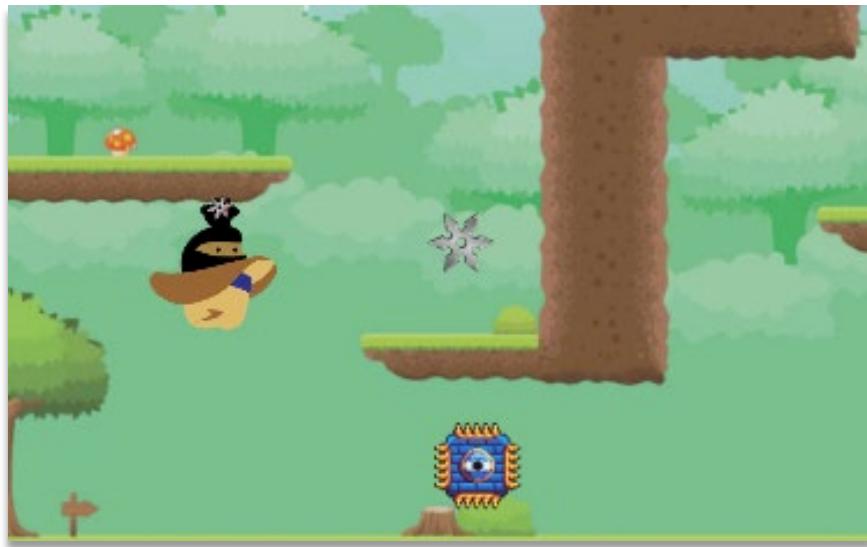
Gravity Trails .....	5
Plan and Design .....	6
Project Setup .....	7
A Walk in the Park .....	14
What Goes Up, Must Stay Up .....	27
You Shall Not Pass .....	35
Seeing Stars .....	39
Throwdown .....	51
Show Me the Shurikens .....	57
Take it up a Level .....	63
Creating a Start Screen.....	75
Sharing Your Unity Project .....	100
Project Submission and Reflection .....	110
Codey Raceway.....	112
Plan and Design .....	113
Project Setup .....	114
On Track .....	118
Shopping Spree.....	125
Moving Mountains .....	128
Exterior Decorating .....	130
The End of the Road .....	133
Eyes on the Road .....	135
Lap it Up .....	142
It's About Time .....	162
Hazardous Conditions .....	175
Feel the Power .....	195
Get Off My Lawn! .....	210
Victory Lap.....	214
Project Submission and Reflection .....	215

Sulky Slimes .....	217
Plan and Design .....	218
Project Setup .....	219
A Whole New World.....	221
Character Building.....	224
Mighty Mouse .....	227
To Infinity .....	235
Reach for the Star .....	241
A Slime's Life .....	251
A Change of Scenery .....	262
Project Submission and Reflection .....	267
Chef Codey .....	269
Plan and Design .....	270
Project Setup .....	271
A Room with a Ninja.....	274
Extreme Café Makeover .....	284
The Toast with the Most .....	287
Now We're Cooking .....	301
Order Up.....	316
How to Program an Egg .....	325
Time to Sizzle.....	337
Project Submission and Reflection .....	347



## Gravity Trails

Your goal is to plan, program, and playtest your very own platformer game. You will begin by using a premade background like you did in the Purple Belt's Scavenger Hunt activity. Then, you will use the Unity Asset store to find and build a unique custom level!



The game we will create together is influenced by a lot of different popular 2D platform games. As you plan and program your game, think about what you like about some of your favorite platform games!

## Plan and Design

Many platformers have a special mechanic that makes them stand out against all the others. In Gravity Trails, the player can't jump! Instead, the player can control the direction of gravity.



Hollow Knight by TeamCherry  
Built in Unity



Cuphead by StudioMDHR  
Built in Unity

Each level needs a background, a goal, and a challenge. For example, while there are enemies throughout the world, the player can collect items to help defeat them. The ability to change the direction of gravity affects only the player character. Part of game design is making the player feel like they are powerful and able to overcome the challenges they face.



### Ninja Planning Document

Take at least 5 but no more than 10 minutes and complete your Ninja Planning Document – Designing a Scene

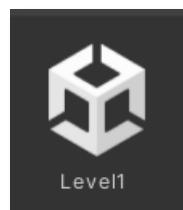
Take a look at the sample projects below for some inspiration!



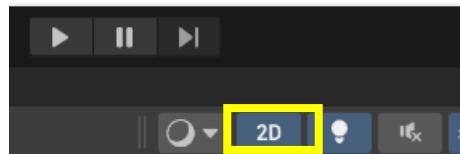
## Project Setup

---

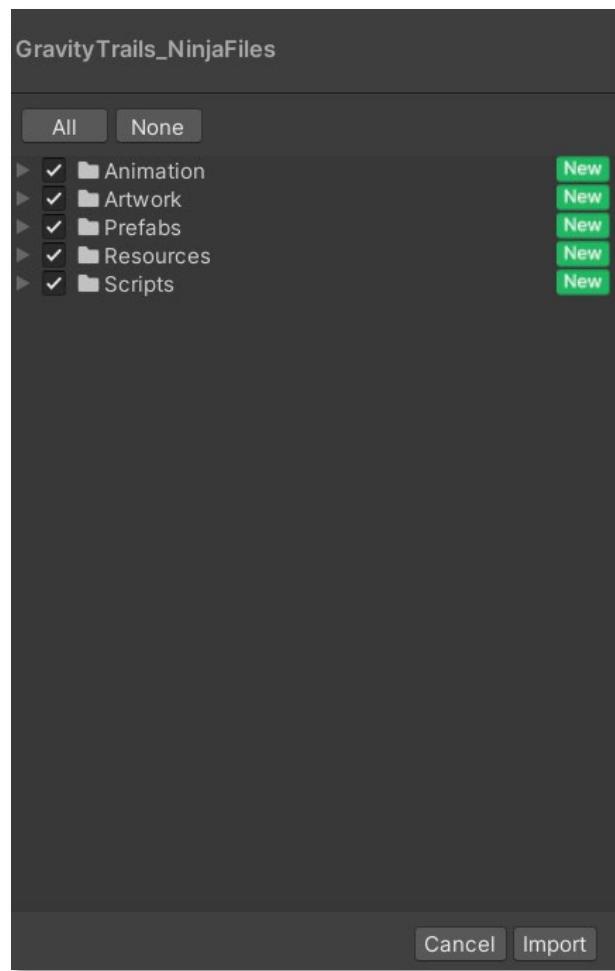
- 1** Start a new Unity Project and name it *YOUR INITIALS – Gravity Trails*.  
Select **3D template**.
- 2** After it loads, rename the Sample Scene to Level 1.



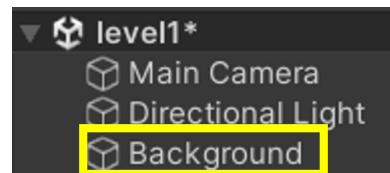
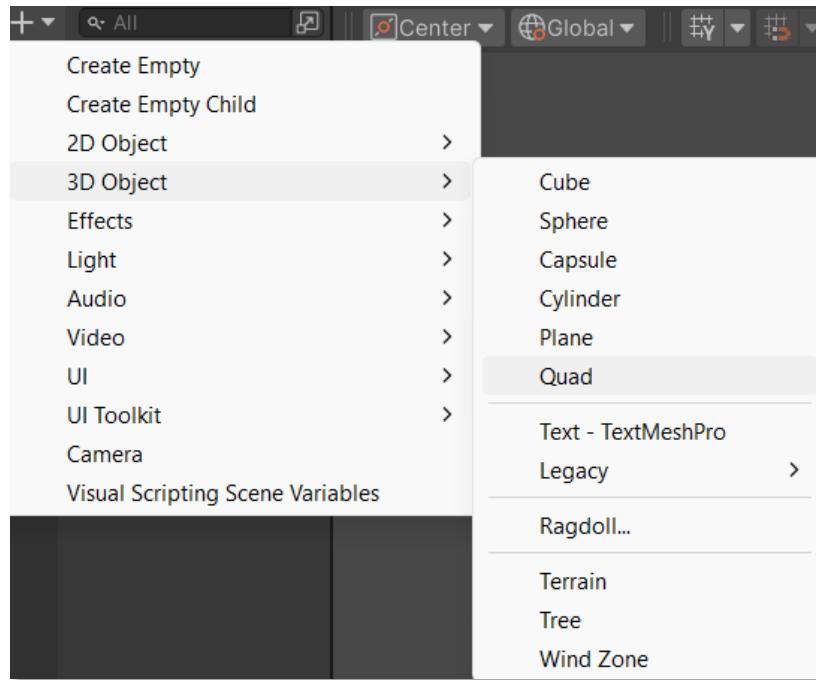
- 3** When your project first opens, it will have a 3D view. We can change that by clicking on the 2D button near the Play Button.



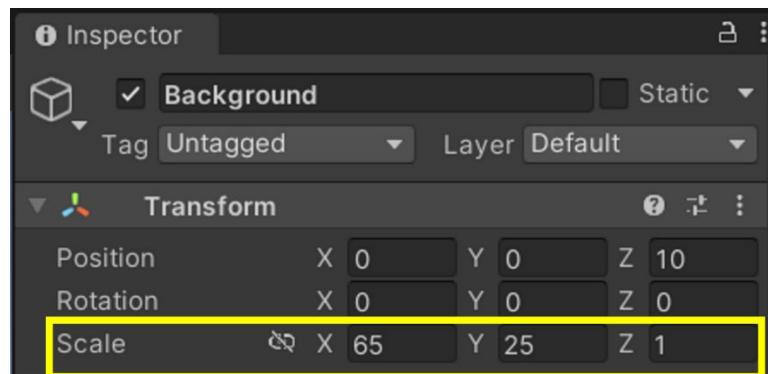
- 4** Next, import the **GravityTrails\_NinjaFiles.unitypackage** provided. This contains the background, the ninja, the enemies, and the other assets used to build level one. Make sure all the folders have a check next to it. Then click **import**.



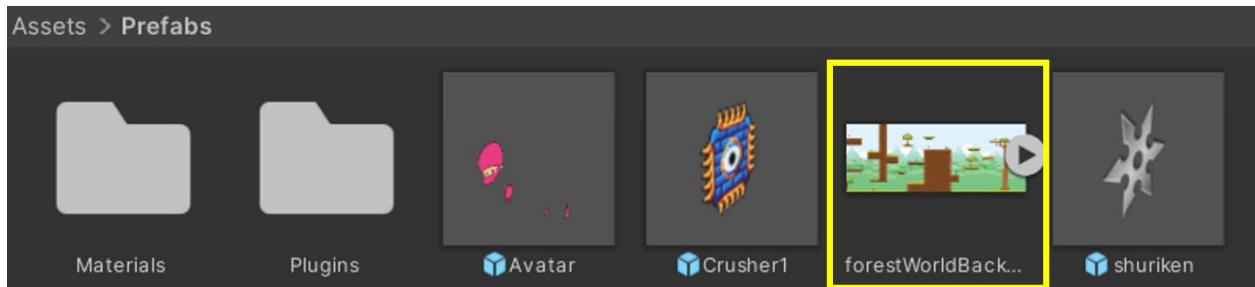
- 5** Create a **Quad** object in your scene. Rename it to **Background**.



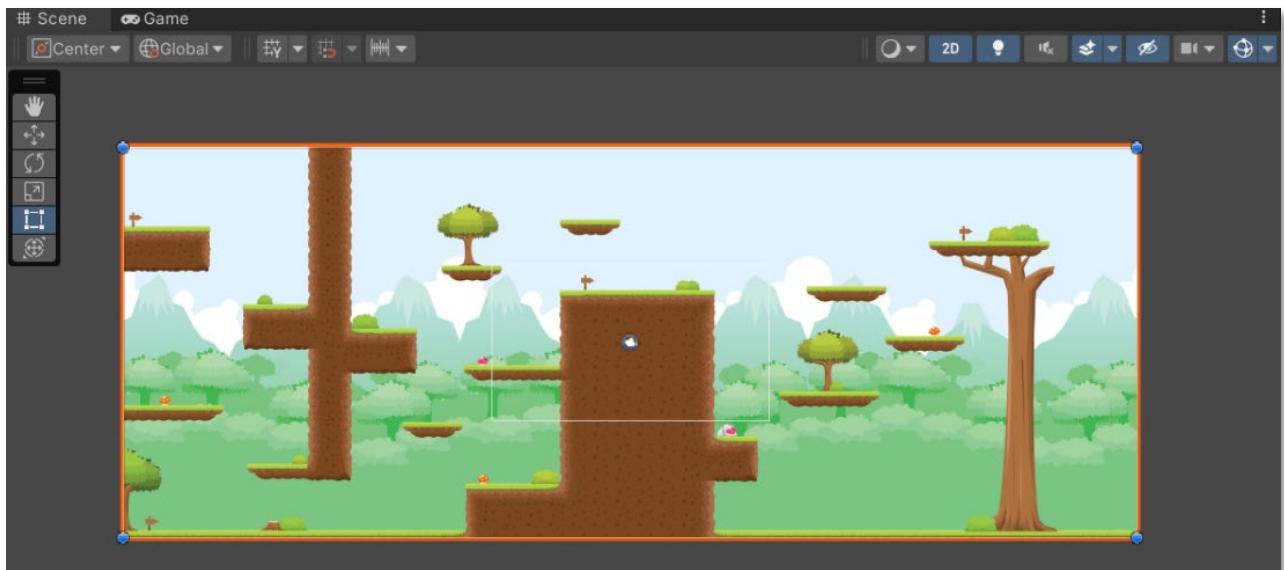
- 6** Adjust the Background's **Scale X** to **65** and the **Scale Y** to **25**. You may change these values later based on your playtesting.



- 7 Find the **forestWorldBackground** image in the **Prefabs** folder.



Drag this image onto the Background quad object in the scene.



### Sensei Stop

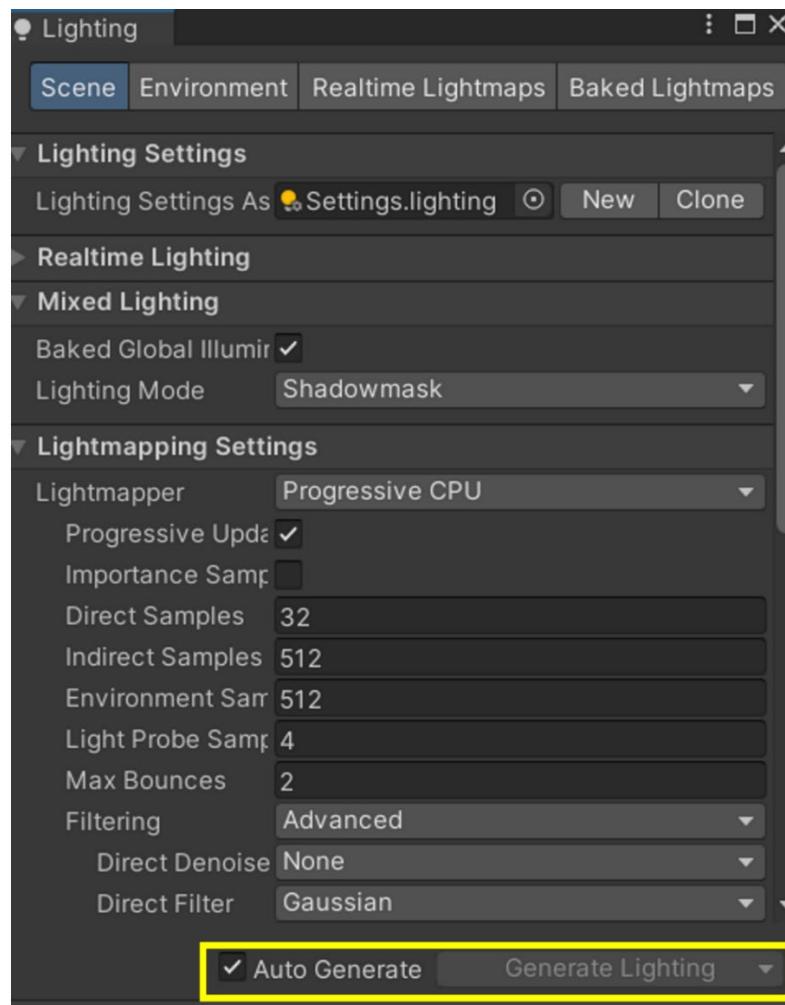
Demonstrate to your Code Sensei that you have created the Quad and attached the `forestWorldBackground` to it.

**8** If you notice that your background looks dark, look at the bottom right-hand corner and make sure that the Auto Generate Lighting is on.

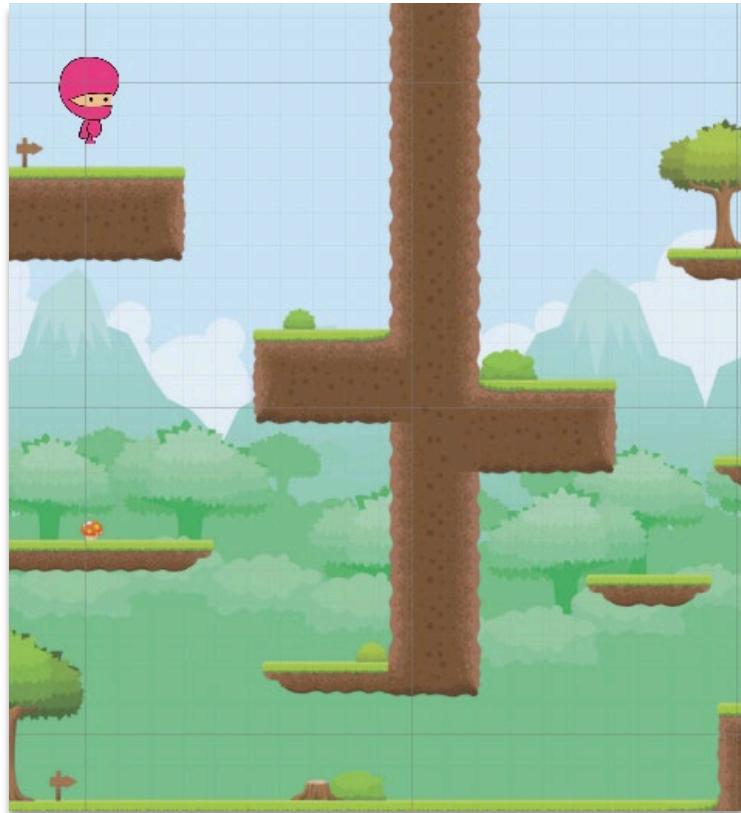
To turn it on, click on the “Auto Generate Lighting” button at the very bottom right of the Unity window.



Make sure **Auto Generate** is checked in the new Lighting window.



- 
- 9 Add the Avatar located in the **Prefabs** folder to the scene. Place the Avatar in the top left of the scene. You can change the position based on your playtests.



## 10 Take a closer look at the Avatar. Is it missing any body parts?



Ours seems to be missing an arm and leg. Remember we had this same problem in the Scavenger Hunt activity? What did we do to the background, so the Avatar can look normal?

If you need a quick reminder, go back to the Scavenger Hunt activity and check step 14!

### Sensei Stop

Describe what you must do for the Avatar to appear correctly. Make the changes and show your Sensei that all Avatar parts are showing.

Your Avatar should like the image below.



## A Walk in the Park

---

- 11** Playtest your game. What happens to the Avatar?

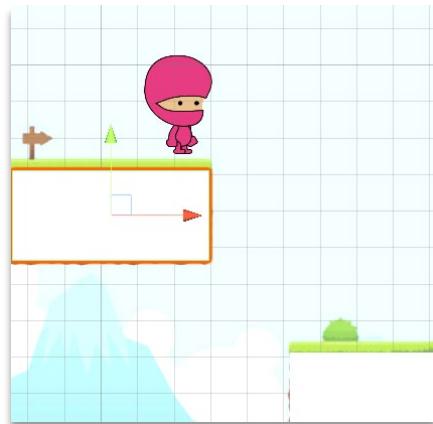


The Avatar is falling down through the scene!

- 12** Similarly, to the Scavenger Hunt activity, we need to create multiple **Quad** objects to have walking surfaces for our Avatar. To help us to keep things organized, first create an **Empty Game Object**, and rename it **Surfaces**. Then create the Quad objects inside the Surfaces object.



**13** Place your first two or three Quad pieces on the ledges near your Avatar. The Quad objects will look like white rectangles that will be used to detect collision in our scene.



**14** Playtest your game and try to go around the scene. Control the Avatar using either the **arrow keys** or **WASD**.

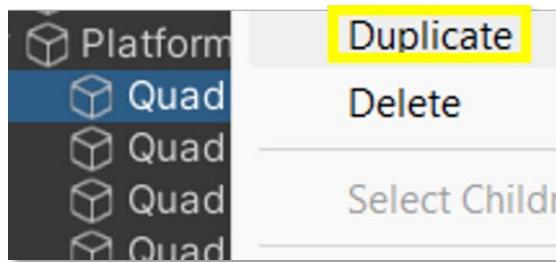
The Avatar still falling through the Quads. What can we add to stop our Avatar from falling?



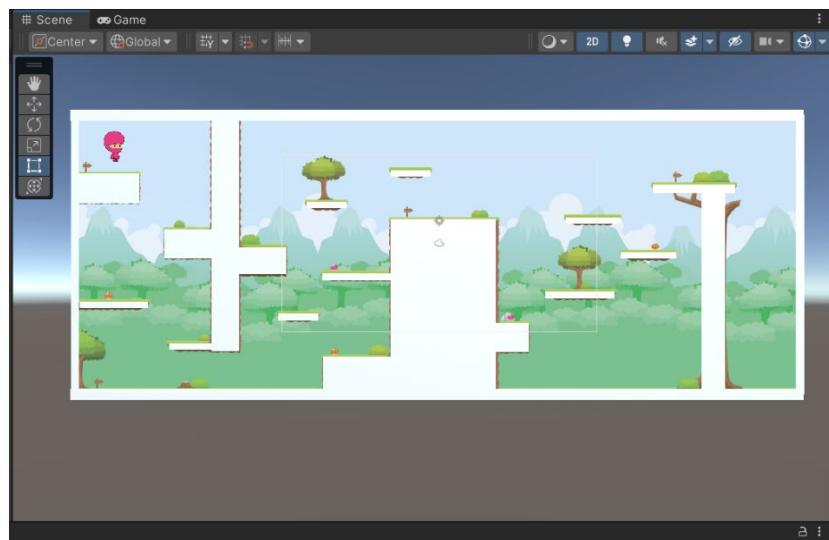
### Sensei Stop

Look at the Inspector for the Quad game objects. What non-2D component needs to be removed? What 2D collider needs to replace it? Tell your Sensei what changes you made.

**15** After you have modified your Quad objects to work with the Avatar, duplicate the original Quad by right clicking and selecting Duplicate or pressing Control + D.



Position and resize these new Quad objects on top of all the surfaces of your image. Remember to place Quad objects around the edges to make sure the Avatar cannot leave the scene.



**16** Playtest your game. Your Avatar will now be able to land on the Quad pieces and the Avatar will not be able to go through the brown dirt areas.



**17** We want to use the Cinemachine to have our camera follow the Avatar.



Select the **Window** tab then the **Package Manager**. Once the new window opens, find **Cinemachine** and click on **Install**

The screenshot shows the Unity Package Manager window. On the left, there is a list of packages under the 'Services' category. On the right, the details for the 'Cinemachine' package are displayed. The 'Install' button is highlighted with a yellow box.

Package	Version
Alembic	2.3.4
Analytics	5.1.0
Android Logcat	1.4.0
Animation Rigging	1.2.1
Apple ARKit XR Plugin	5.1.2
AR Foundation	5.1.2
Authentication	3.3.0
Build Automation	1.0.5
Burst	1.8.12
CCD Management	2.2.2
Cinemachine	2.9.7
Cloud Code	2.5.1
Cloud Diagnostics	1.0.6
Cloud Save	3.1.0
Code Coverage	1.2.5 ✓
Collections	2.1.4
Core RP Library	14.0.10
Deployment	1.3.0
Device Simulator Devices	1.0.0

**Cinemachine**  
2.9.7 · May 08, 2023 [Release]  
From Unity Registry by Unity Technologies Inc.  
`com.unity.cinemachine`  
[Documentation](#) | [Changelog](#) | [Licenses](#)

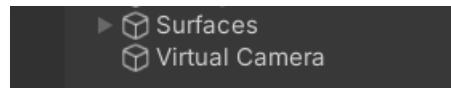
[Description](#) [Version History](#) [Dependencies](#)

Smart camera tools for passionate creators.

New starting from 2.7.1: Are you looking for the Cinemachine menu? It has moved to the GameObject menu.

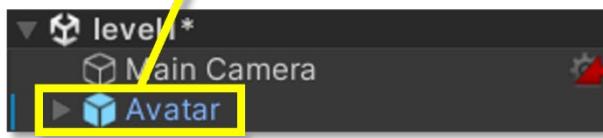
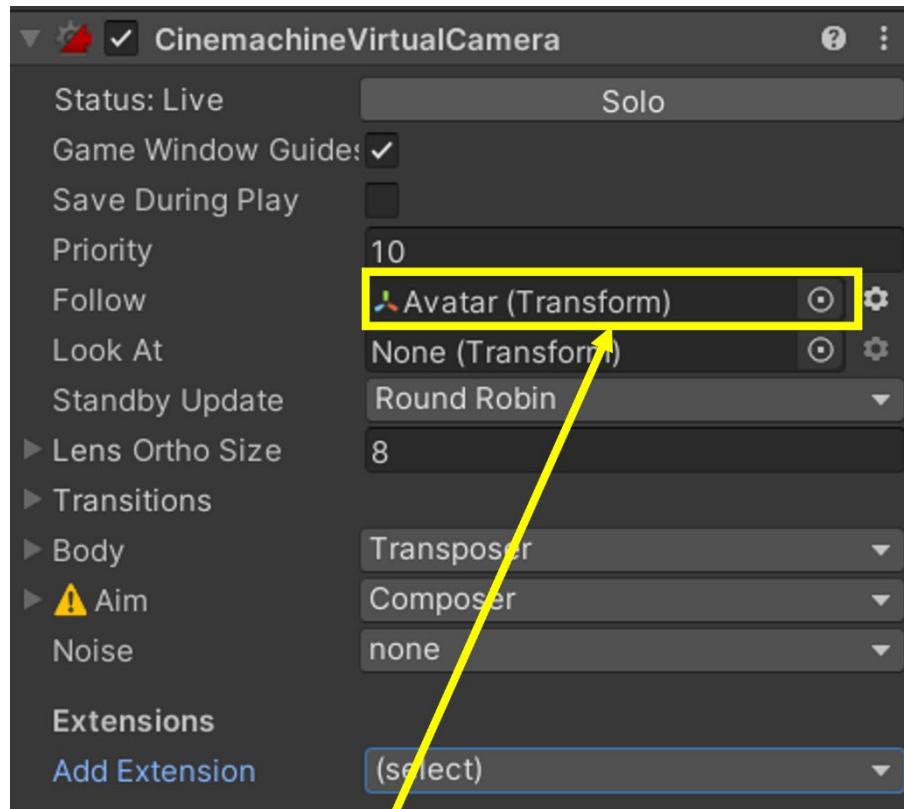
IMPORTANT NOTE: If you are upgrading from the legacy Asset Store version of Cinemachine, delete the Cinemachine asset from your project BEFORE installing this version from the Package Manager.

**18** In the **Inspector** you should see a new game object **Virtual Camera**.



If you do not see this object, then click on **Cinemachine** next to Window at the top and select **Create Virtual Camera**.

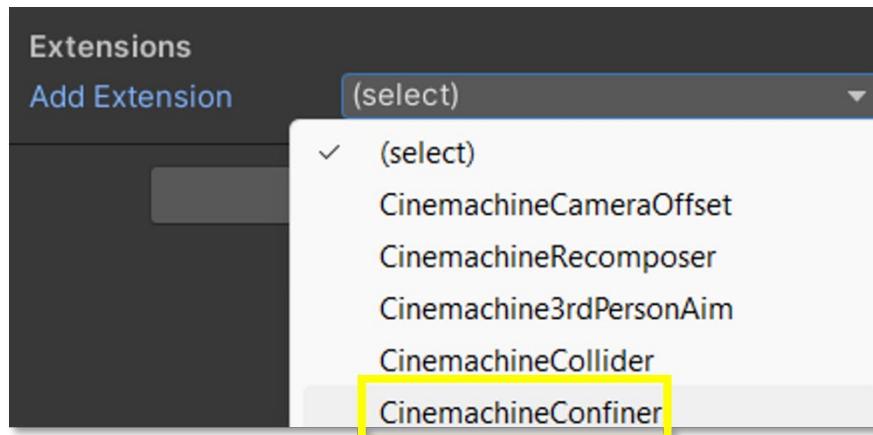
**19** Select the **Virtual Camera game object**. In the Inspector, drag the Avatar game object into the **Follow** property.



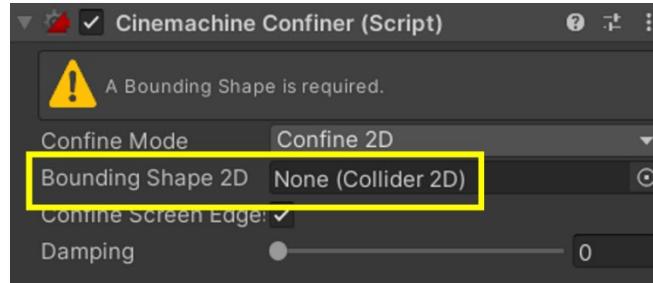
**20** Playtest your game. The camera is showing the empty scene behind the background image.



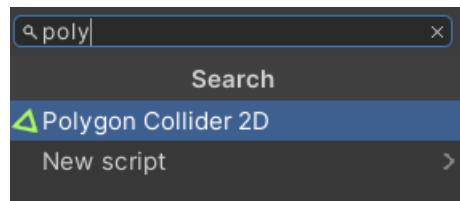
Select the **Virtual Camera** game object. Click **Add Extension** at the bottom of the CinemachineVirtualCamera component. Add the **Cinemachine Confiner** Extension.



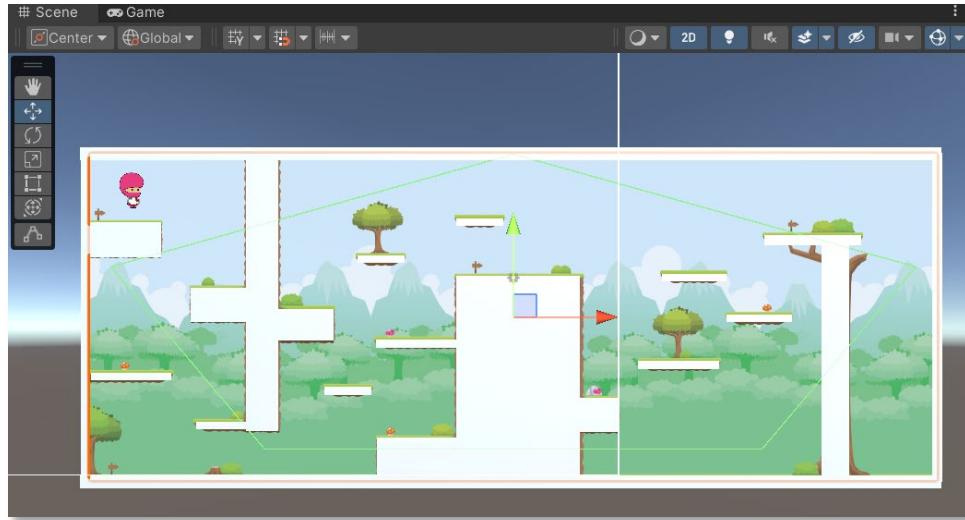
**21** This component needs a Collider to act as a box for our camera.



Select your Background game object and add a Polygon Collider 2D component.



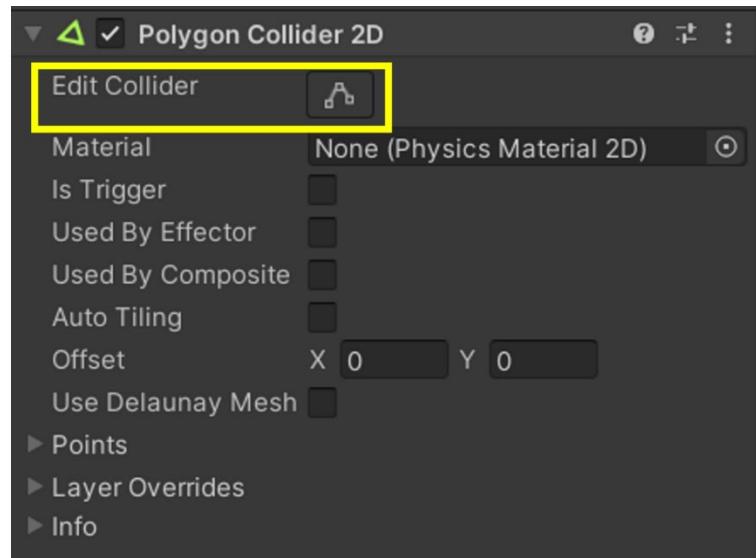
**22** There should now be a green shape in your scene.



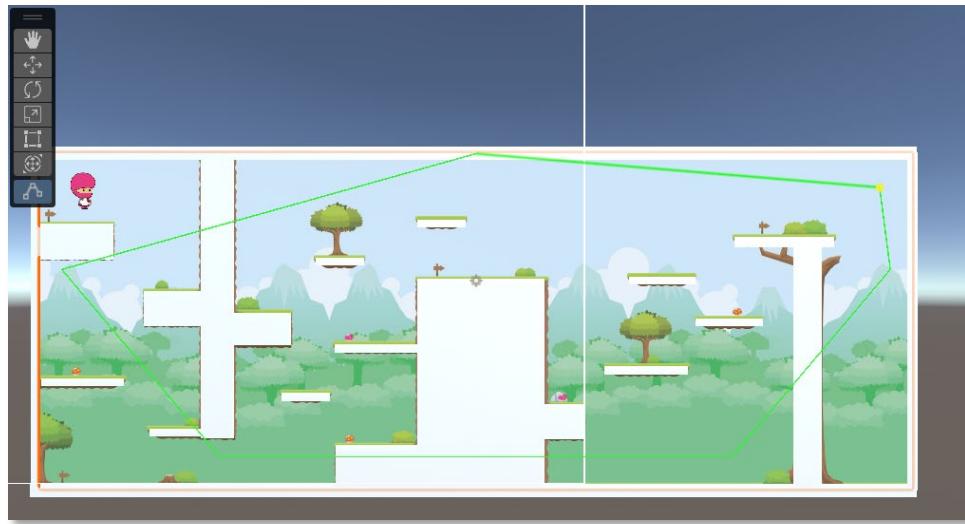
## Lighting

If your scene is too bright, open the lighting window and adjust the Environment Lighting Intensity Multiplier.

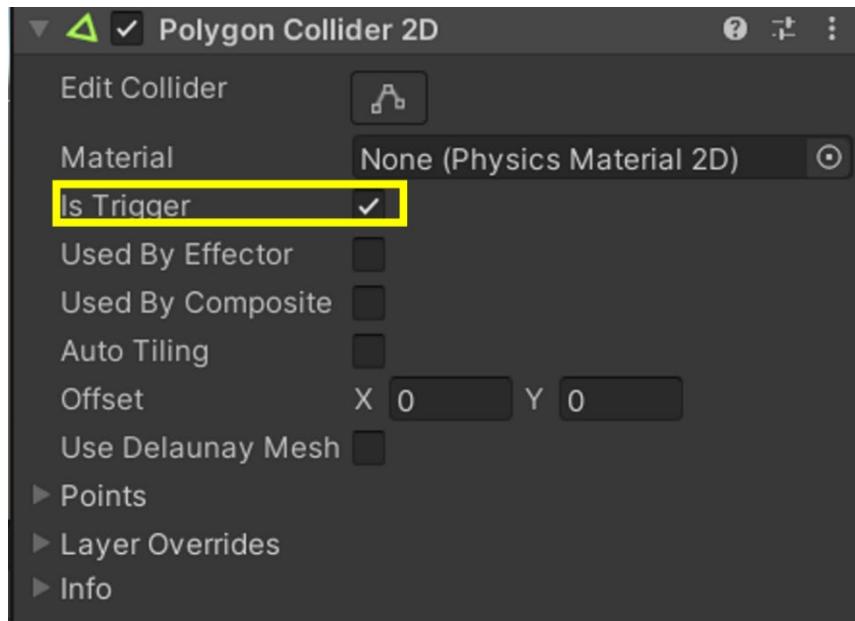
**23** Click the Polygon Collider 2D component's Edit Collider button.



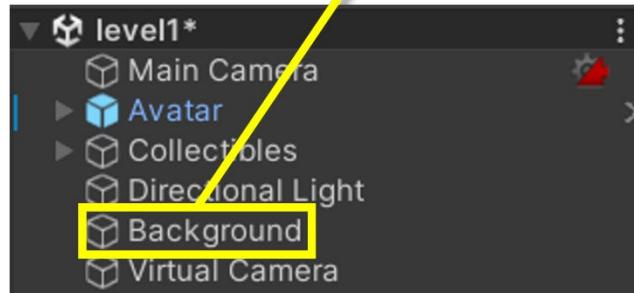
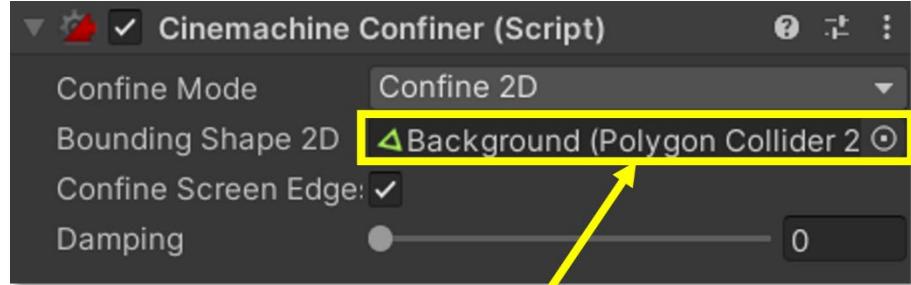
Drag the corners of the green polygon to the corners of the background image. Unity might draw thin green edges between the corners of your collider. When you are done, click the Edit Collider button again.



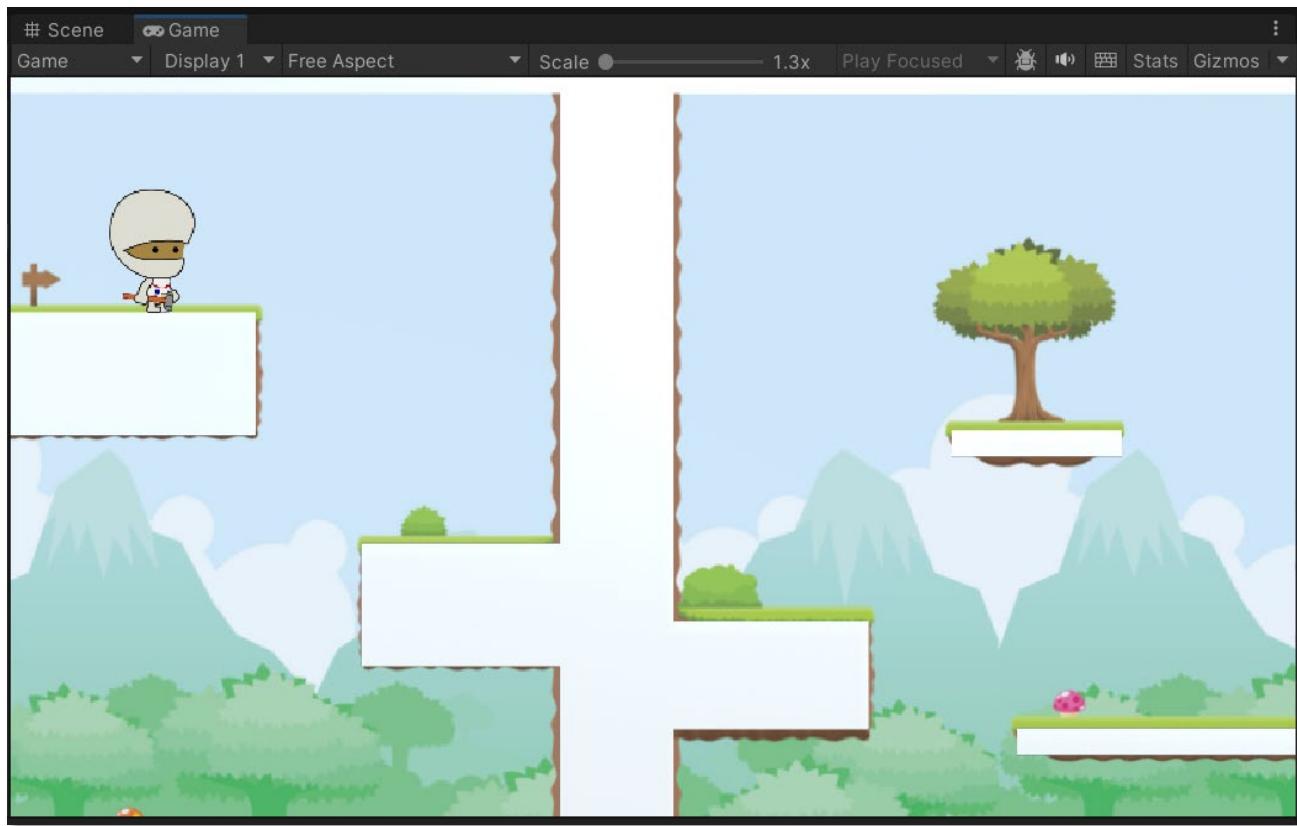
**24** Enable the Polygon Collider 2D component's Is Trigger property. If you do not do this, then your Avatar will be forced outside of the background image when you play your game.



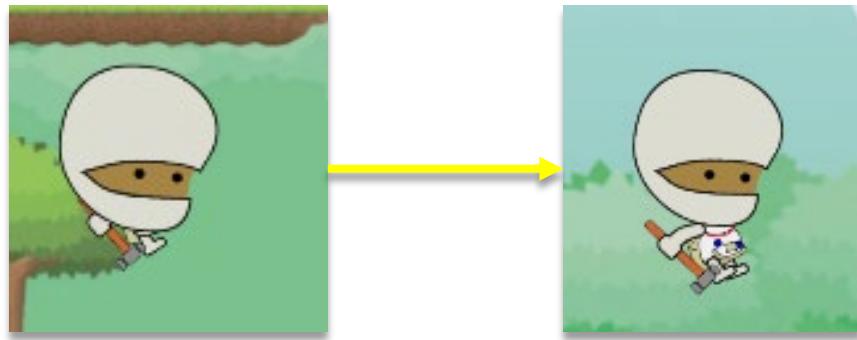
**25** Drag in the background game object to the **Polygon Collider 2D** component.



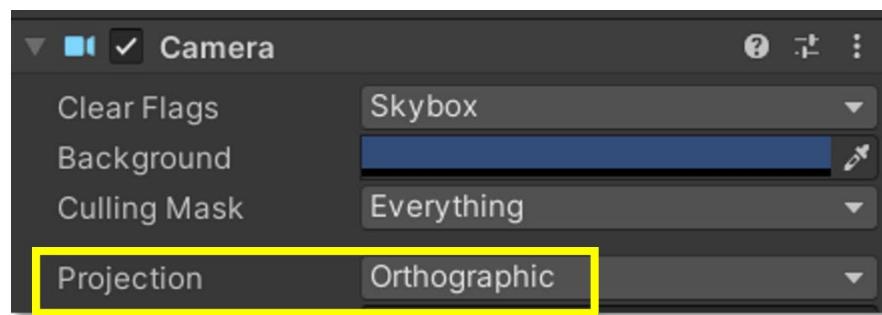
**26** Playtest your game. Now the camera will stay inside the collider.



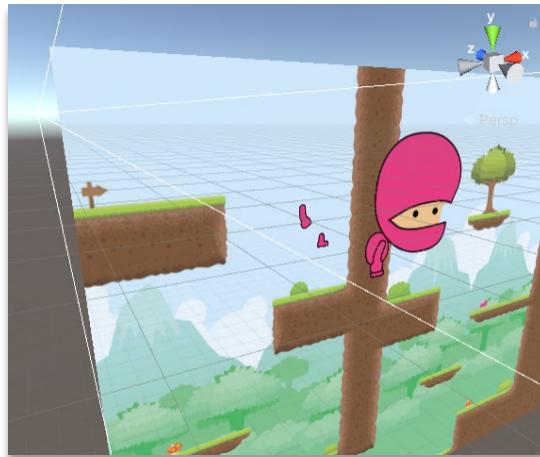
**27** Does your Avatar look a little smushed?



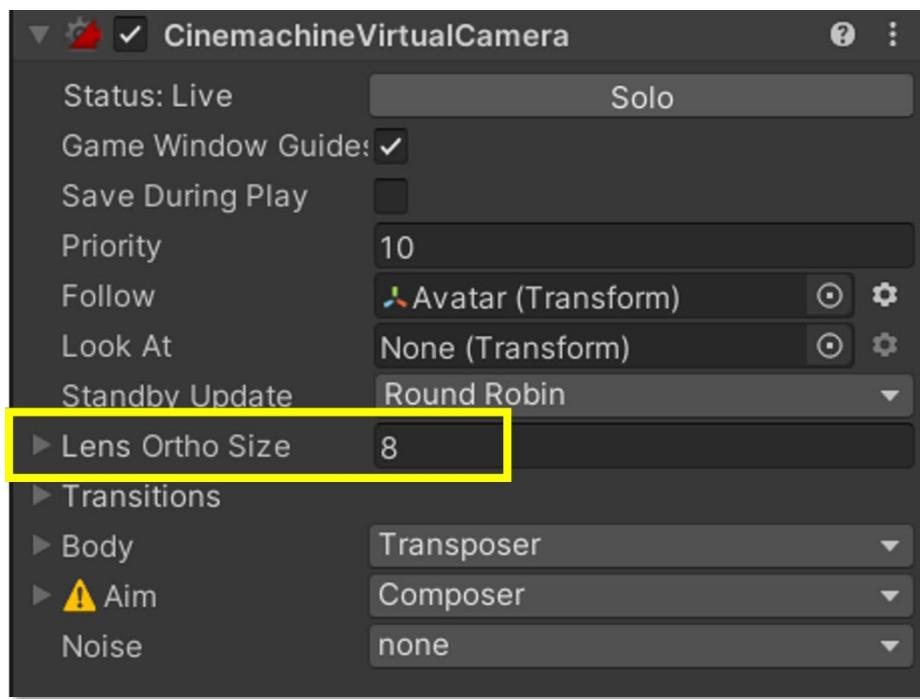
Click on the Main Camera game object and set Projection to Orthographic.



This tells Unity to ignore depth even though our 2D objects use the Z axis.



**28** Change the zoom of the camera by adjusting the **Virtual Camera** game object's Lens Ortho Size. Try values between 6 and 12 until you find a level of zoom that you like.



## What Goes Up, Must Stay Up

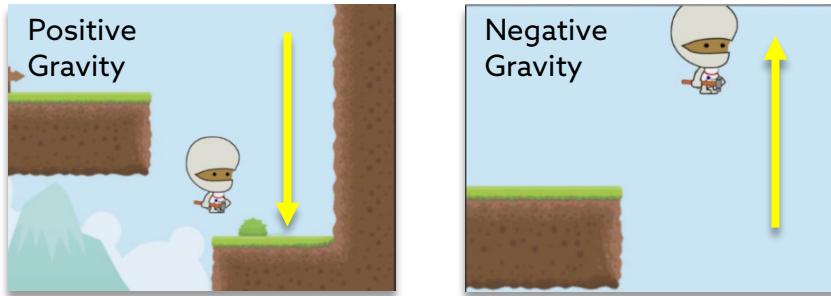
**29** Select the Avatar and in the Inspector. We want the player to control the direction of gravity so the Avatar can walk on the ceiling.



### Sensei Stop

Go through the Avatar's components in the Inspector and discuss with your Code Sensei what component allows us to control gravity? Change the value of the property to see how it affects the Avatar.

**30** Let's use this same logic in code so when we press the spacebar, gravity is either positive or negative.

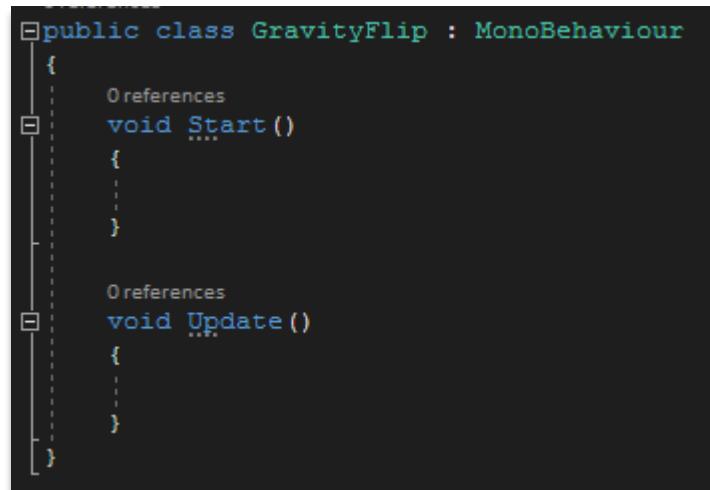


In your **Scripts** folder, create a new script named **GravityFlip**.



---

**31** Attach this script to the Avatar and open the **GravityFlip** script in Visual Studio.



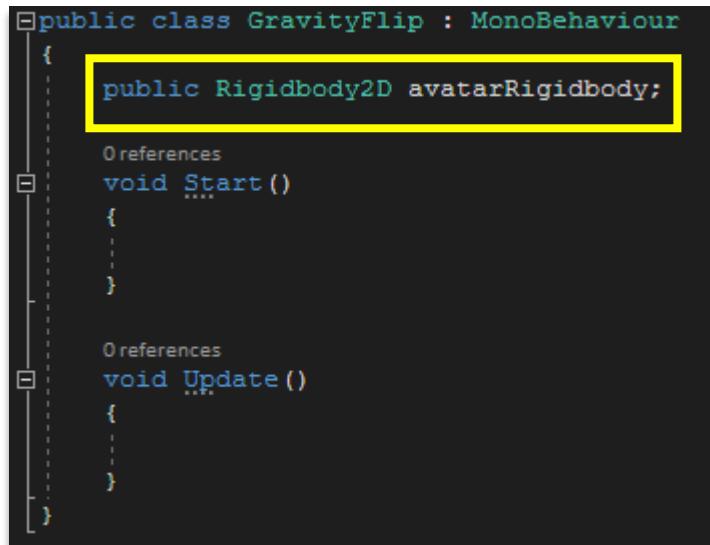
```
public class GravityFlip : MonoBehaviour
{
    void Start()
    {
    }

    void Update()
    {
    }
}
```

---

**32** Since the Rigidbody component controls the gravity, we need to access it through our code.

Create a public variable that allows us to access the **Rigidbody2D** component in our code.



```
public class GravityFlip : MonoBehaviour
{
    public Rigidbody2D avatarRigidbody;

    void Start()
    {
    }

    void Update()
    {
    }
}
```

- 33** Save your script and return to Unity. In the **GravityFlip** script component, fill the empty slot with the **Avatar (Rigidbody 2D)**.



Now we can make changes to all the different properties of the Avatar's Rigidbody through our code. For this project, we will only change the Gravity Scale property.

- 34** Return to the GravityFlip script. Write a condition in the Update function to check if the player pressed the spacebar.

```
public class GravityFlip : MonoBehaviour
{
    public Rigidbody2D avatarRigidbody;

    void Start()
    {

    }

    void Update()
    {
        if (Input.GetButtonDown("Jump"))
        {
        }
    }
}
```



### Input Names

To see all of input names and keys go to Edit, then Project Settings, then Input Manager.

---

**35** When the player presses the **spacebar**, flip the sign of the **Rigidbody's gravity scale** by multiplying it by -1.

```
void Update()
{
    if (Input.GetButtonDown ("Jump"))
    {
        avatarRigidbody.gravityScale *= -1;
    }
}
```

---

**36** Save your **script** and return to Unity. Playtest your game and press the spacebar. Is your Avatar now floating or falling on the ground?

Select your Avatar and in the **Inspector** find the Rigidbody 2D's **Gravity Scale** property.

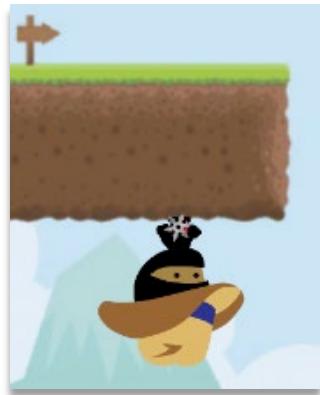
Continue pressing on the spacebar. Do you now see how the Gravity Scale automatically turns into either 2 or -2?

Gravity Scale      2      Gravity Scale      -2

Instead of us manually changing the values like we did in the Sensei Stop, we can now change gravity by pressing on the spacebar!

---

**37** We are now able to go through the entire course, but we don't want the Avatar to look like it is floating.



---

**38** Playtest your game and select the Avatar. Begin by making sure your Avatar's **Gravity Scale** is **negative**. Your Avatar should look like it is floating.

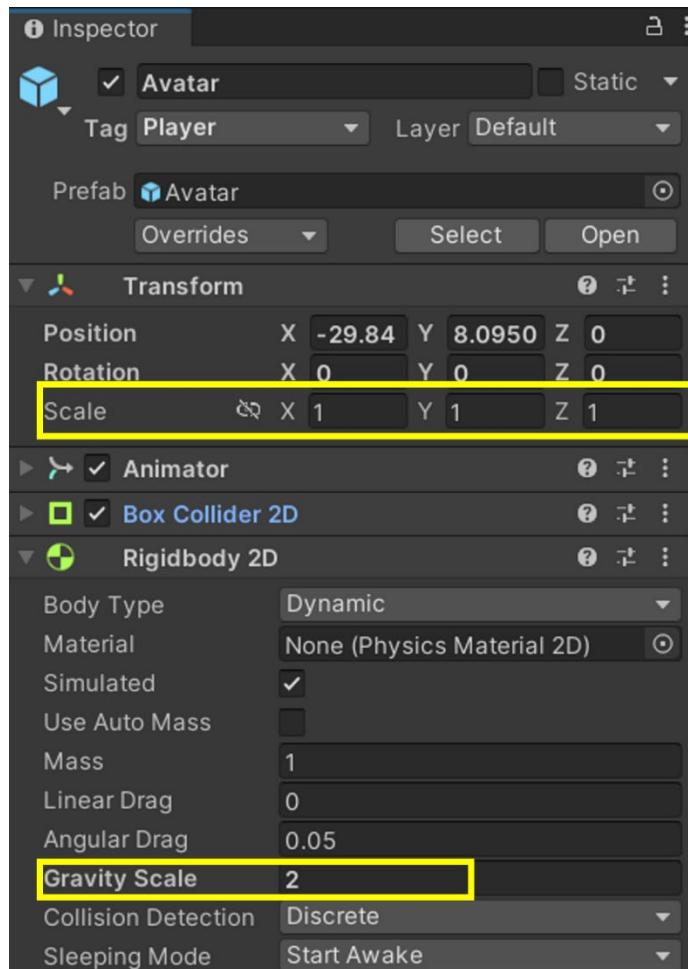


**39** In the Transform component, look at the **Scale values** in the Inspector for your Avatar. What happens when you make the X, Y, or Z **scales** negative?



Describe what happens to the Avatar when you make each scale negative. Then tell your Sensei what scale you think you should make negative when the gravity is flipped so that the avatar looks like its walking upside down.

**40** Before moving on make sure your Scale and Gravity Scale components are back to the original values as shown below:



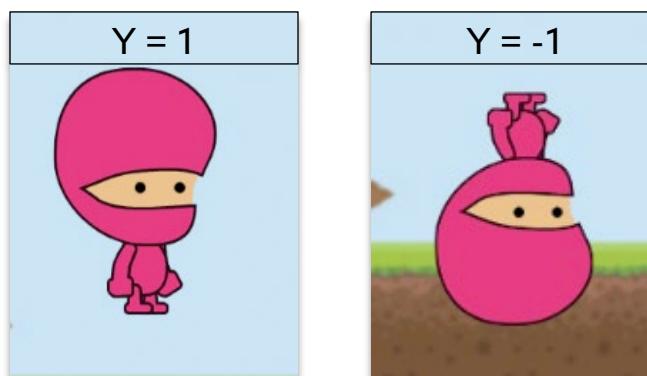
- 41** Open the **GravityFlip** script. Inside the Update function's if condition, store the current Scale values of our Avatar in a **Vector3** variable. Name the Vector3 variable **newDirection**. We are going to use it to flip the **Y Scale** value.

```
void Update()
{
    if (Input.GetButtonDown("Jump"))
    {
        avatarRigidbody.gravityScale *= -1f;
        Vector3 newDirection = transform.localScale;
    }
}
```

- 42** When we experimented with the negative values earlier, we learned that it can flip the direction the Avatar is facing. Multiply the **newDirection's y property** with **-1**.

```
void Update()
{
    if (Input.GetButtonDown("Jump"))
    {
        avatarRigidbody.gravityScale *= -1f;
        Vector3 newDirection = transform.localScale;
        newDirection.y *= -1;
    }
}
```

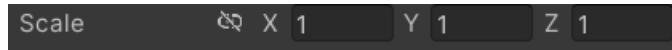
The Avatar's Y scale is 1. When the spacebar is pressed, it is multiplied by -1, and the newDirection variable will either have a value of 1 or -1.



**43** Save your script and return to Unity. Playtest your game. What happens when we press the spacebar?

The Avatar still is not flipping! Why is that?

Look at the Avatar's inspector. Is the Y Scale getting updated like we think it is when we press the spacebar?

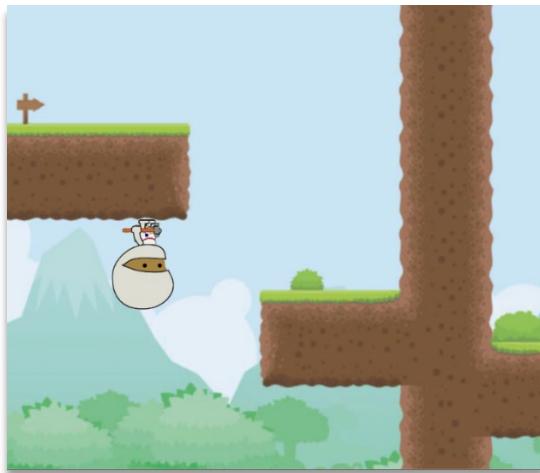


Open the GravityFlip script.

 **Sensei Stop**

Look at the code we have so far. Discuss with your Sensei what is missing and then implement the change necessary to have our Avatar flip when the space bar is pressed.

**44** After you change your code, save your script and return to Unity. Playtest your game. Your Avatar will now flip based on the direction of gravity.



## You Shall Not Pass

---

**45** Now that the player can navigate through the entire level, we can add enemies to keep the game challenging.

For our enemies, we are going to use the Crushers from Robomania. If you don't want to use the crushers, use the Unity Asset Store to find the enemies you want to use. They can be anything you want, but make sure that the enemies are 2D.



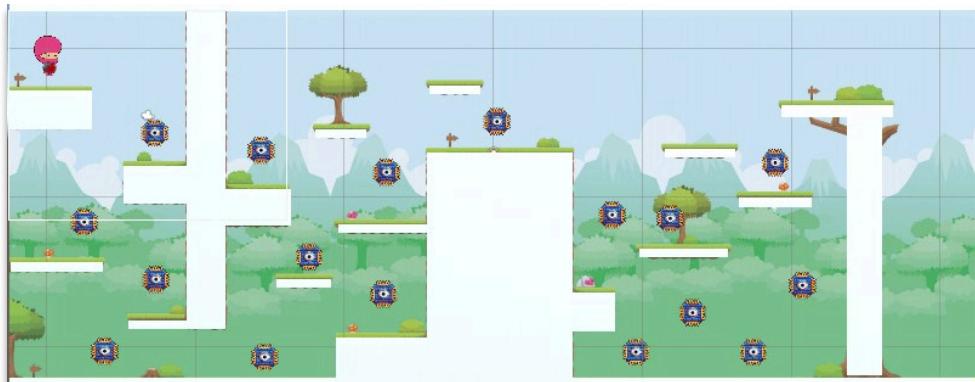
In the Robomania activity, you programmed the Crushers to bounce back and forth between the scene's boundaries. In this scene we want to add at least **4 enemies**, and they will have **different** boundaries.

---

**46** Drag your enemies from the prefab folder into the scene. Adjust the scale to make sure they fit in the environment. Make sure that your enemies have the BoxCollider2D and Rigidbody components.



A good game designer does not want to add a ton of enemies to the world because the level becomes too hard to complete. We need a good balance of enemies so the player has a chance to get to the end.



As you can tell from the picture above, it is almost impossible to get through the scene without crashing into an enemy.

It may look and feel funny to the person creating the game, but this will bring the game experience down for a player. Playtest your game a few times to find a number of enemies that isn't too easy or too hard.

**47** Using what you have learned and the resources from previous activities, you will program your enemies to bounce back and forth in your scene. If the Avatar collides with the enemies, the level will restart using the **Scene Manager**.

Let's work on our enemy movement first. On your own, program the enemies to move like they do in Robomania. Use steps **33a-44b** of the Brown Belt Robomania activity.

Create a public variable for the maximum and minimum bounds. This way you can have different values for the bounds for all the different enemies.

```
public int maximumXPosition;  
public int minimumXPosition;
```



### Sensei Stop

Demonstrate to your Sensei that your enemies are moving around the scene.

**48** Now that our enemies can bounce around the scene, create a new script in the **Scripts** folder and rename it **EnemyCollision** and attach it to the Avatar. This script will check if the player collides with an enemy.



**49** In this script we only want to check when the Avatar collides with the enemies. If it does, then we want to restart the scene.

On your own create the logic that checks when these two objects collide. You will have to:

- assign your enemies the **Enemy** tag,
- create an **OnCollisionEnter2D** function,
- check to see if the avatar collides with an Enemy, and
- restart the scene.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    |
}
```



Demonstrate and describe to your Code Sensei what happens when the Avatar collides with the enemy.

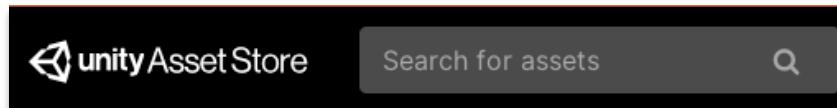
## Seeing Stars

**50** What can we do to help our Avatar defend themselves from the enemies? In this section we are going to Instantiate, or clone, an object that our Avatar can throw to defeat the enemies.

Game design involves making the player feel like they can overcome daunting challenges! Let's place multiple collectables for the Avatar to collect and use to clear enemies!



**51** You can use the Unity Asset store to find a game object you would like to use as an offensive attack.



In this example project we are using a shuriken. If you want to use the shuriken, you can find it in the **Artwork** folder.

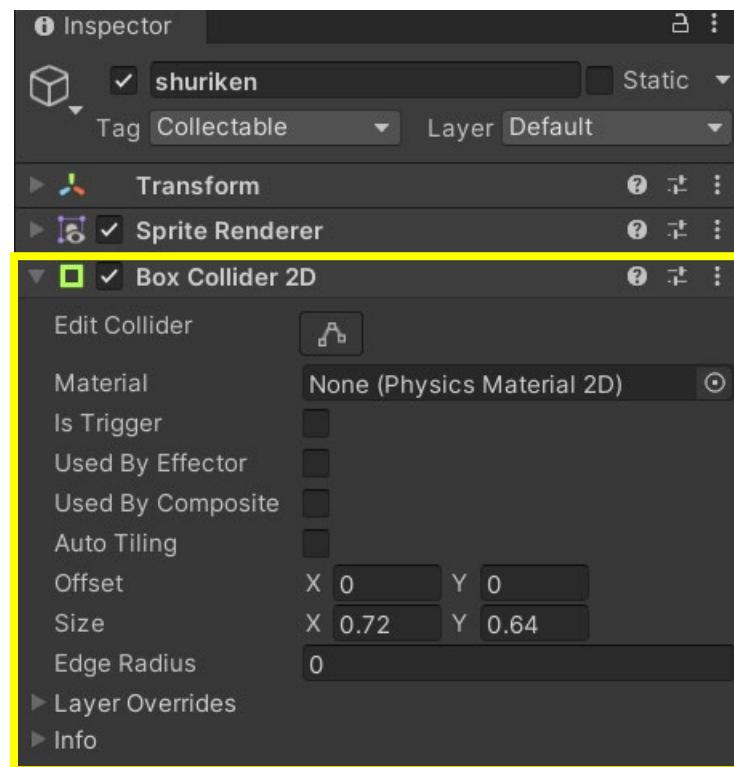


Click to expand the arrow and drag the shuriken on the right onto the scene.

**52** Select your object and assign it the **Collectable** tag.



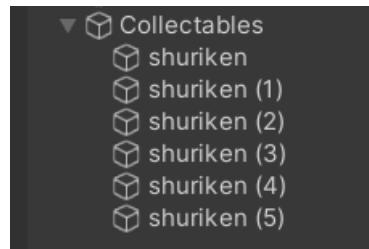
**53** Give the object you are using a **Box Collider**.



**54** Place multiple collectables in your scene. We recommend having more collectables than enemies. This way, we can ensure that the player has enough chances to defeat the enemies.



**55** To keep things organized, create an **empty game object** and rename it **collectables**. Add your collectable game objects to keep things organized in the Hierarchy.



---

**56** Before our Avatar can throw an object, the player needs to collect one. When designing your game, you do not want to give the player everything all at once. You want them to work for it.

Create a new script in the Scripts folder and name it **Throwable**. Attach the script to the Avatar game object.



---

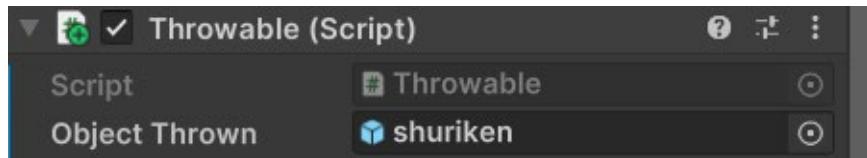
**57** Open the script in Visual Studio. We are going to Instantiate the game object that we want to throw.

Create a **public GameObject** with the variable name **objectThrown**.

```
public GameObject objectThrown;
```

---

**58** Save your script and return to Unity. Drag the object you want to use in the **objectThrown** property so it will know what prefab we want to clone.

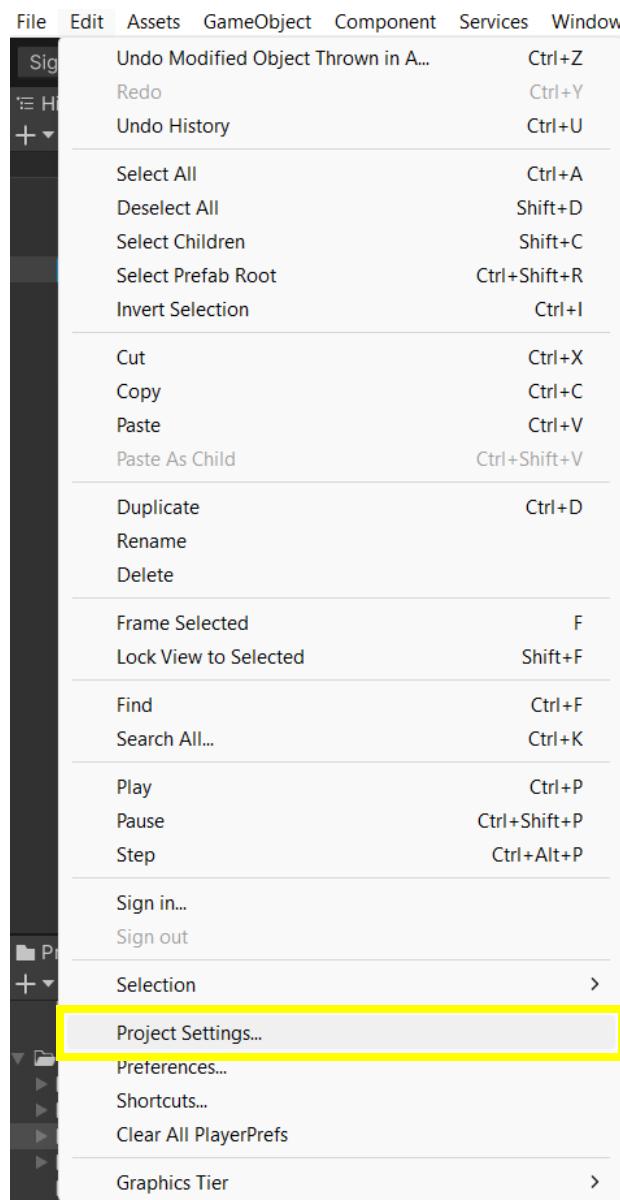


**59** Now that the Avatar knows what object it will be using, we need to instantiate it where the Avatar is positioned.

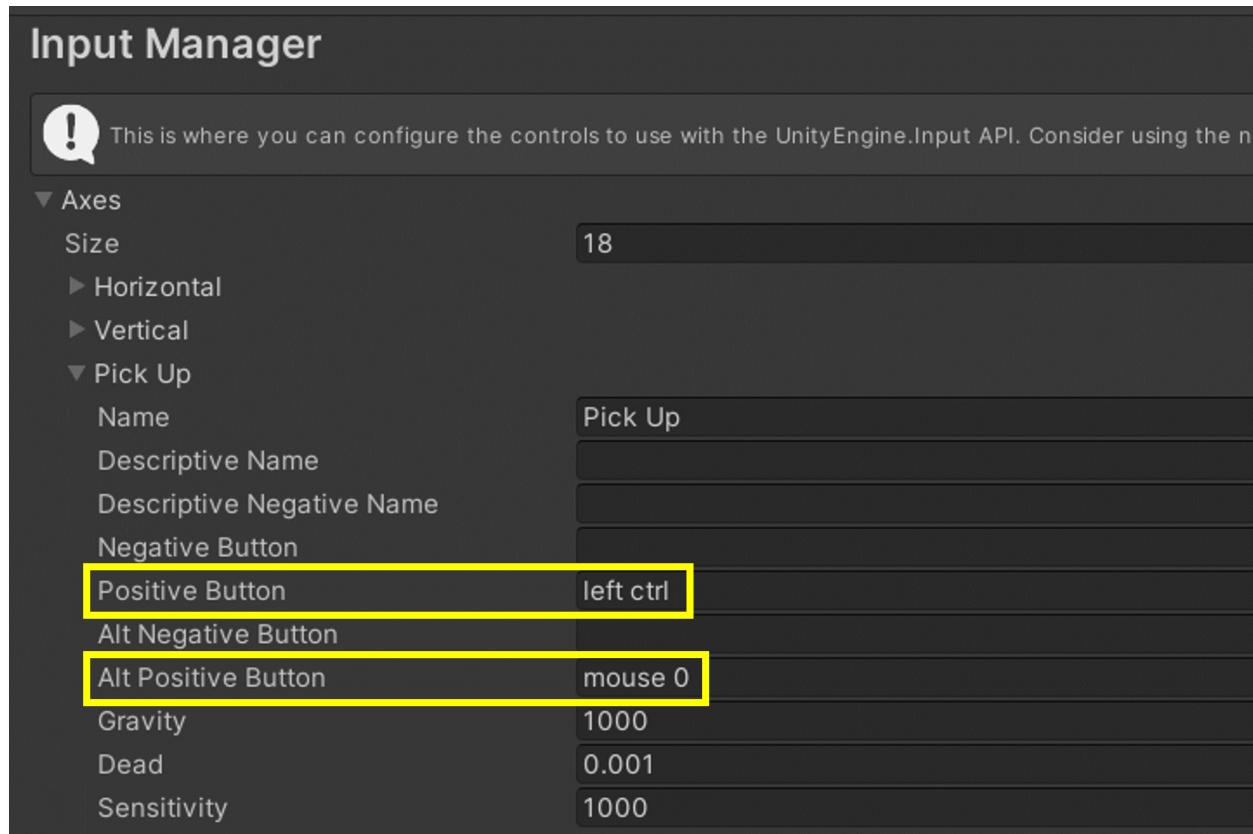
Open the **Throwable** script. In the **Update** function we want to place the object in front of our Avatar.

```
void Update()
{
    if (Input.GetButtonDown("Fire1"))
    {
        Instantiate(objectThrown, transform.position, transform.rotation);
    }
}
```

**60** Save your script and return to Unity. To check which key press triggers the **Fire1**, in the tabs above go into Edit then Project Settings.



**61** Select Input Manager, click the triangle to unfurl the Axes menu, and find Fire1.



To throw the collectable, you will have to press the control key that is on the left side of the keyboard. You can also left click on your mouse, as there is an alternative positive button!

---

**62** If you don't want to use the left control key as a way to throw your object, you can delete "left ctrl" and replace it with the key you want. To see the exact key value you need to type, click on the question mark button at the top right of the Project Settings.

This will take you to the Unity Documentation. Scroll down until you find **Mapping virtual axes to controls**.

Under **Key Family** you see the keyboard keys it is referencing. Under **Naming convention**, it tells you the name you need to add in the **Positive Button** component so that Unity understands what key you are pressing.



No matter what key you want to use, make sure you do have a specific key in the Positive Button component!

---

**63** Exit the settings menu and playtest your game. What happens when you try to clone the object?



The object appears behind the avatar and pushes the avatar forward!

We must reposition the object slightly in front of our Avatar or it will continue to push our Avatar forward.

---

---

**64** Open the **Throwable** script. Create a new **Vector3** variable and name it **offset**. We will use this variable to calculate what side we clone our object, either left or right.

```
public Vector3 offset;
```

---

**65** Right before we **Instantiate**, in the **Update()** function and inside the **if** condition, we want to set **offset** to equal a **new Vector3** that positions the object one position to the side of the Avatar.

```
offset = new Vector3(1, 0, 0);
```

The **X value** moves the object from left or right.

The **Y value** moves the object up or down.

The **Z value** moves the object forward or back a layer.

**66** We want to position the throwable in the direction that our Avatar is facing. To do this, multiply the new **Vector3** by the Avatar's **transform.localScale.x**.

```
offset = transform.localScale.x * new Vector3(1, 0, 0);
```

We want to use the Avatar's **localScale** because it will give us a value of 1 or -1 depending the X direction it is facing. You can see which direction the avatar is facing in the inspector while the game is running.

If our Avatar is facing to the right, Scale X is equal to 1.



If our Avatar is facing to the left, Scale X is equal to -1.



**67** So far, we have coded the throwable to appear one place in front of the Avatar, but we haven't told it where we want it to instantiate.

Inside the Update function's if condition, create a **Vector3** variable with the name **throwablePosition**. Set it equal to our Avatar's **transform.position + offset**. This way, the shuriken is always placed one unit in front of the Avatar in the X direction.

```
Vector3 throwablePosition = transform.position + offset;
```

- 68** The last step is to Instantiate, or clone, the throwable using the **throwablePosition**. Update the code from before by replacing transform.position with throwablePosition.

```
Instantiate(objectThrown, throwablePosition, transform.rotation);
```

- 69** Save your script and return to Unity. Playtest your game and test creating a throwable object when the Avatar is facing right and left.



- 70** There is one problem. We do not want our Avatar to be able to clone an infinite number of objects. We only want to be able to clone when we have collected an object from the game world. Stop your game return to the **Throwable** script.

**71** We want to track the number of objects we have collected. Create a new **int** variable and name it **throwableCounter**.

```
public int throwableCounter;
```

**72** On your own, add to the **Throwables** script so that the Avatar only clones an object if it has collected objects.

Create an **OnCollisionEnter2D** function that:

- checks if the collided item has the Collectable tag,
- increments **throwableCounter** by 1, and
- destroys the collected object.

Modify the “Fire1” conditional statement so it:

- clones an object only if **throwableCounter** is greater than zero, and
- subtracts one from **throwableCounter** when a throwable game object is Instantiated.

#### Sensei Stop

Work with your Sensei to answer the following questions based on the above pseudocode. How does the Avatar pick up an item? How many items can the Avatar throw? If the Avatar has not collected an item, can it throw an object?

**73** Save your script and return to Unity. Playtest your game. If you select the Avatar while the game is running, you should see the game object disappear and the throwable counter value increase when your Avatar collides with a collectable.

After you collect one, press the “fire1” key or left click. What happens? Our throwables appear in front of the avatar, but they don’t move!

## Throwdown

**74** Right now, our object spawns in front of the Avatar, but that doesn't help the player defeat the enemies. We need to program the logic to make our object remove the enemies!



In the scripts folder, create a new script and name it **Projectile**. Attach this script to the object you are cloning. In our example, the shuriken object is in the **Prefabs** folder.



**75**

As soon as our object is instantiated, we want it to move in the direction the Avatar is facing.

We first need a variable that knows the direction the throwable should move in. Luckily, we created the offset variable in the Thrower script that tells us the direction the Avatar is facing.

To access that variable in the Projectile script, create a **public Thrower** variable with the name **direction**.

```
public Thrower direction;
```

**76**

Since we are modifying the prefab object, we cannot use the Inspector to attach game objects from the scene.

As soon as the throwable is created, we want to be able to access the variables in the Thrower script. In the **Start()** function, set the value of direction to the Player object's Thrower component.

```
void Start()
{
    direction = GameObject.FindGameObjectWithTag("Player").GetComponent<Thrower>();
```

**77**

In the **Update()** function, increment the throwables **transform.position** by **direction.offset**.

```
void Update()
{
    transform.position += direction.offset;
```

**78** Save your script and playtest your game. Our throwable moves so fast! Can you think of a way we can slow it down? Open the **Projectile** script.

We can slow it down by multiplying `direction.offset` with `Time.deltaTime`.

```
transform.position += direction.offset * Time.deltaTime;
```

**79** Save your script and return to Unity. Playtest your game again. The throwable is really slow now! Can you think of how we can adjust the speed?

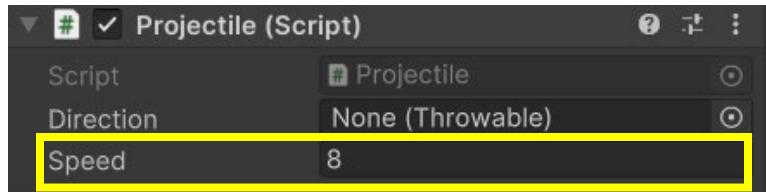
Stop the game and open the **Projectile** script. One way we can speed up how fast the throwable moves is by creating a **public float** variable named **speed**.

```
public float speed;
```

**80** Multiply `Time.deltaTime` by **speed**.

```
void Update()
{
    transform.position += direction.offset * Time.deltaTime * speed;
}
```

**81** Save your script and return to Unity. Click on the throwable object prefab you are using. In the **Inspector** find the **Projectile** script. Give the **Speed** variable a value. We recommend a value between 6-10.



**82** Playtest your game. The throwable should now be moving at a good speed for the player – not too fast or slow! It will also move in the direction the Avatar is facing!

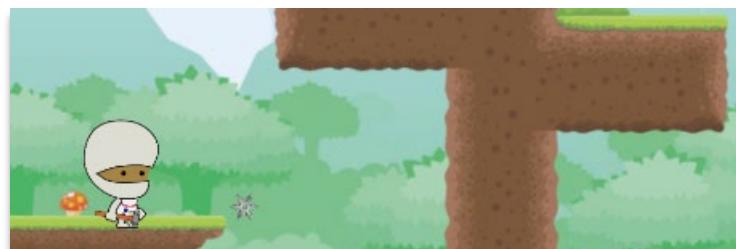


**83** Notice that the object you instantiate never gets removed from the scene.

We need to destroy the enemy and the object the Avatar is throwing if they collide.



We also need to destroy the object the Avatar throws even if it does not collide with an enemy.

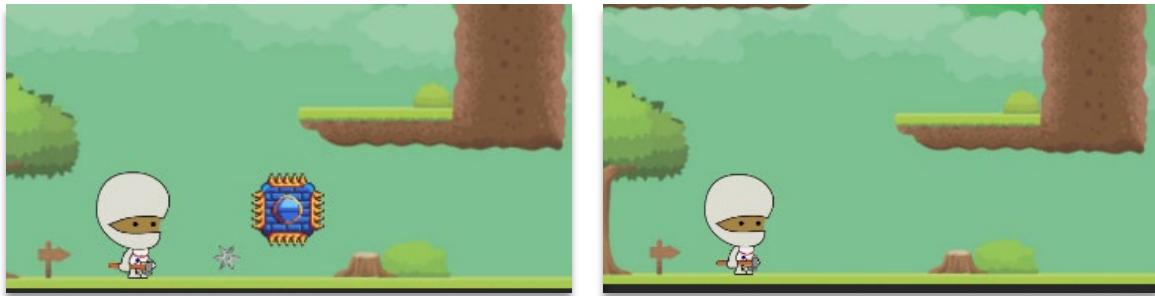


**84**

In order to destroy the enemies, you must:

- give the throwable a unique tag,
- use the **OnCollisionEnter2D** function, and
- destroy the enemy and the throwable.

We have already created the **OnCollisionEnter2D** function in the **EnemyMovement** script. You can use this function to hold the logic that checks when the throwable collides with the enemy.



### Sensei Stop

Demonstrate to your Code Sensei that you have successfully implemented the pseudocode that describes the interaction between throwables and enemies.

Remember to save your script before returning to Unity. Playtest your game to ensure the enemies and throwable are removed when they collide.

**85** Next, we need to destroy the throwable game object if it doesn't collide with an enemy after a few seconds.

Open the **Projectile** script and create a private void function called **DestroyThrowable()**.

```
private void DestroyThrowable()
{
}
```

**86** This function will only be used to destroy our throwable game object. Inside the function, add **Destroy(gameObject)**.

```
private void ...DestroyThrowable()
{
    Destroy(gameObject);
}
```

**87** On your own, **Invoke** the `DestroyThrowable` function so that it destroys our game object after a few seconds.

If you need a refresher on how to use the `Invoke` function, look back at the **Brown Belt** activity **World of Color** activity.



### Sensei Stop

Demonstrate to your Code Sensei that your throwable is destroyed after a few seconds. If you are not able to see the throwable vanishing, use the scene view and zoom out so you can see it disappear.

## Show Me the Shurikens

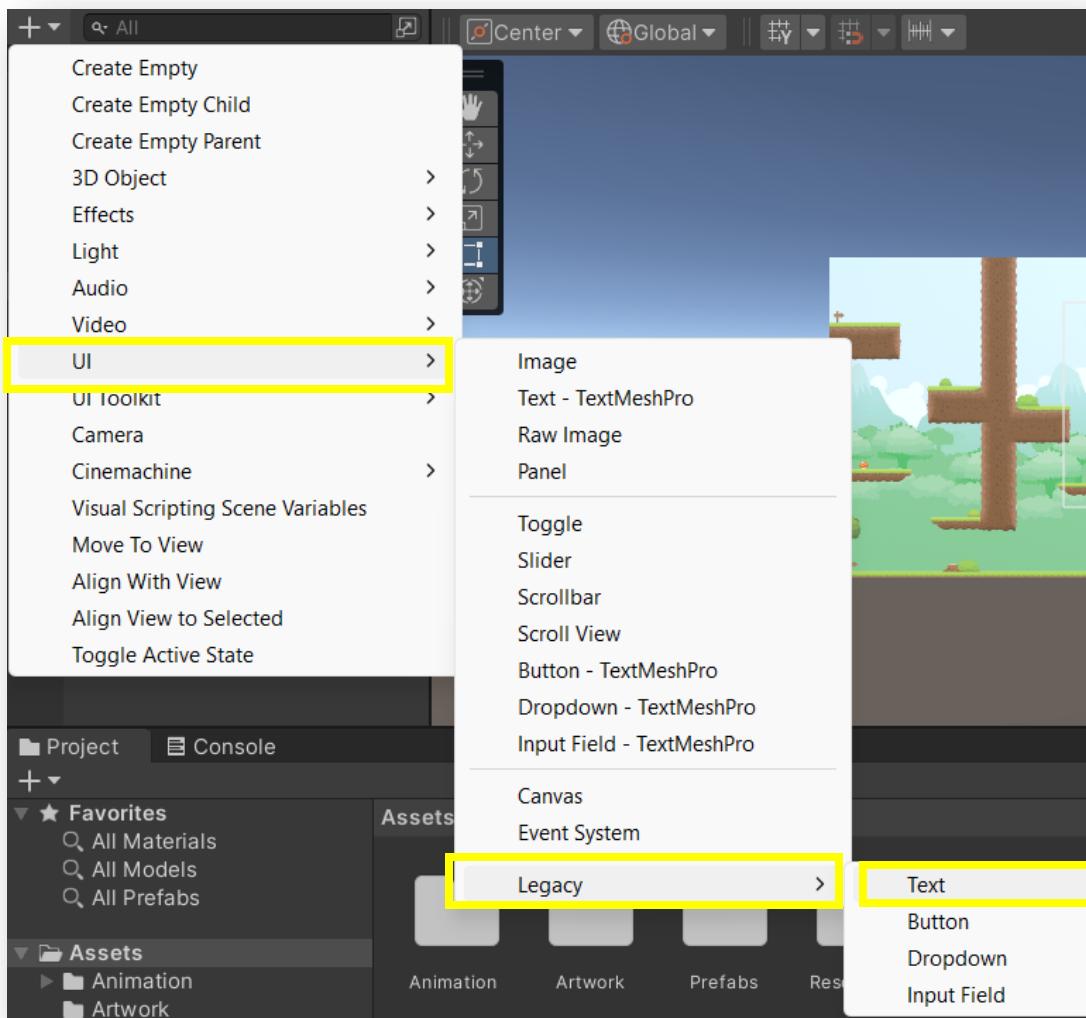
---

**88** Our Avatar is now able to collect items and throw them at enemies to get past them! However, the person playing our game won't be able to click the Inspector to see how many throwable items they have collected.

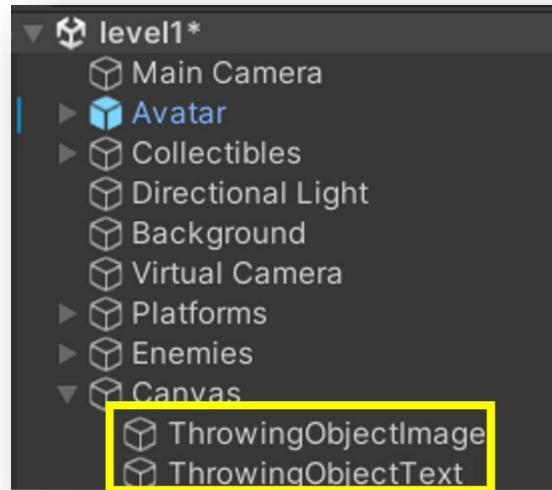
Let's create a way for our player to know exactly how many objects it can throw at the enemies.



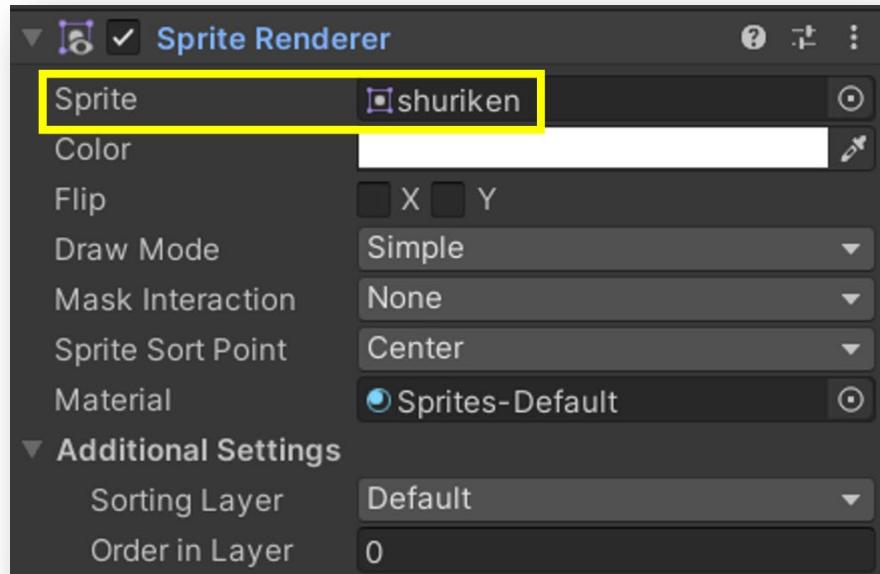
## 89 Create a UI Legacy Text and UI Image in the Hierarchy.



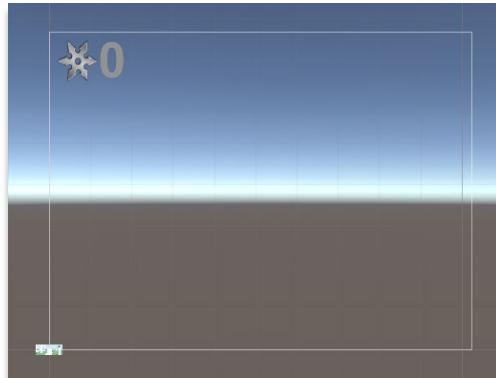
**90** Rename your image to ThrowingObjectImage and the Text to ThrowingObjectText.



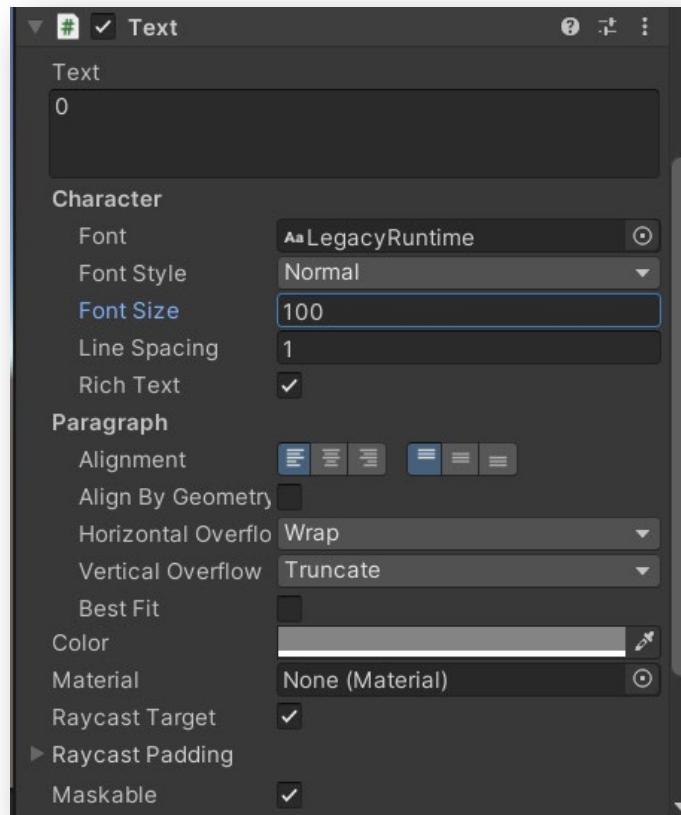
**91** Select the **ThrowingObjectImage**. In the Inspector find the **Image** component, we are going to change the **Source Image** to the throwable you are using. Click on the circle and choose your throwable.



**92** Position the **ThrowingObjectImage** anywhere on the Canvas and the **ThrowingObjectText** next to it. We suggest somewhere at the top right or left so we are sure the player can see the UI.



**93** Resize and customize the **ThrowingObjectText** to your liking by changing the **Text** components.



**94** We can update the text value in our **Throwable** script. Open the script. Add **UnityEngine.UI** to the top of the script.

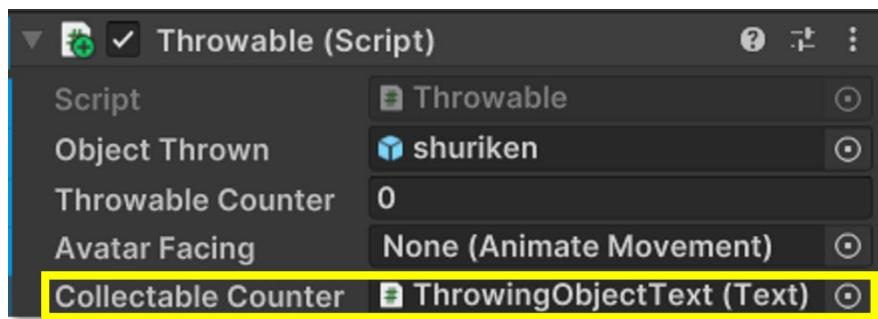
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
```

**95** We need to create a variable in order to update the text object. Create a **public Text** variable named **collectableCounter**.

```
public Text collectableCounter;
```

Save your script and return to Unity.

**96** Drag in the **ThrowingObjectText** into the **CollectableCounter** component in the Inspector.



**97** Open the **Throwable** script again. We already have the `throwableCounter` variable, which checks how many objects the Avatar can throw.

On your own, update the **ThrowingObjectText** according to the **throwableCounter** value.

If our Avatar collides with a collectable:

- add one to `throwableCounter`, and
- set `collectableCounter.text` equal to `throwableCounter.ToString()`.

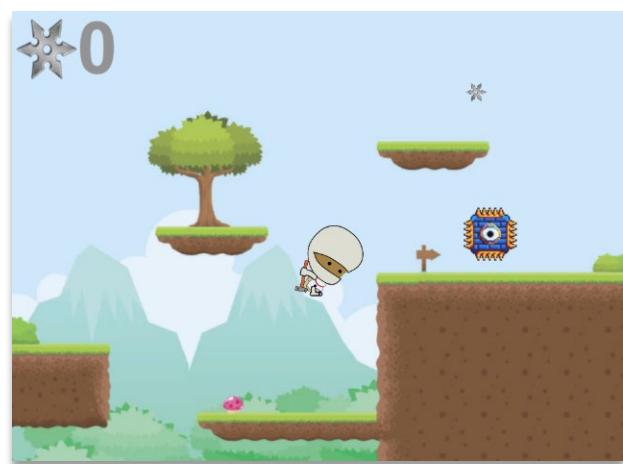
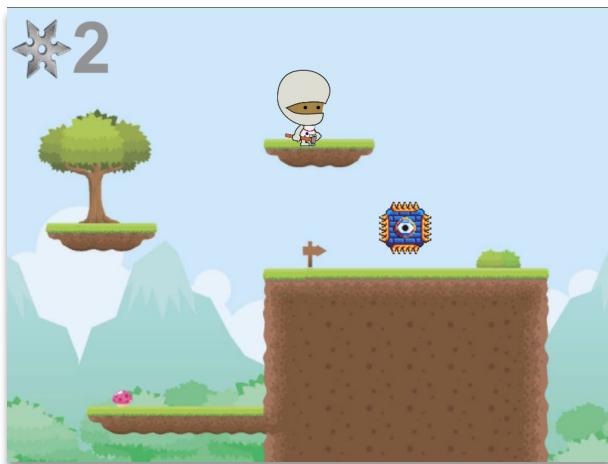
When the player throws a collectable:

- subtract one from **throwableCounter**, and
- set `collectableCounter.text` equal to `throwableCounter.ToString()`.

 **Sensei Stop**

Demonstrate to your Code Sensei that the `offenseText` can be seen in the game and that it updates properly as items are collected.

Your game should update every time you collect or throw an object.

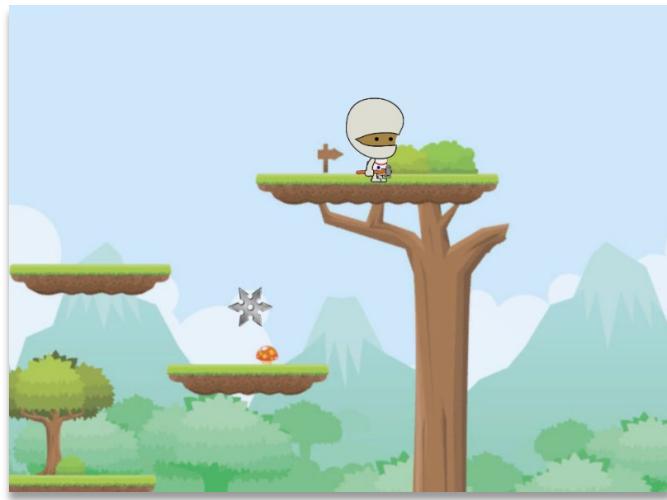


## Take it up a Level

---

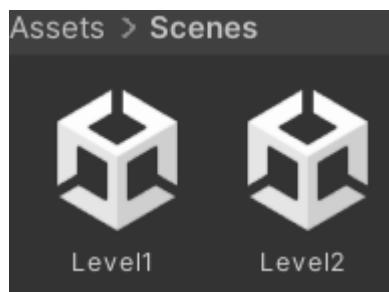
**98** Our Gravity Game now works! There are enemies, throwable objects, and a user interface. Our goal in this game is to eliminate all the enemies and get to the end of the scene to move on to level two.

For the forest background, the player needs to reach the top right of the scene. If you used a different background image, your goal area may be different.

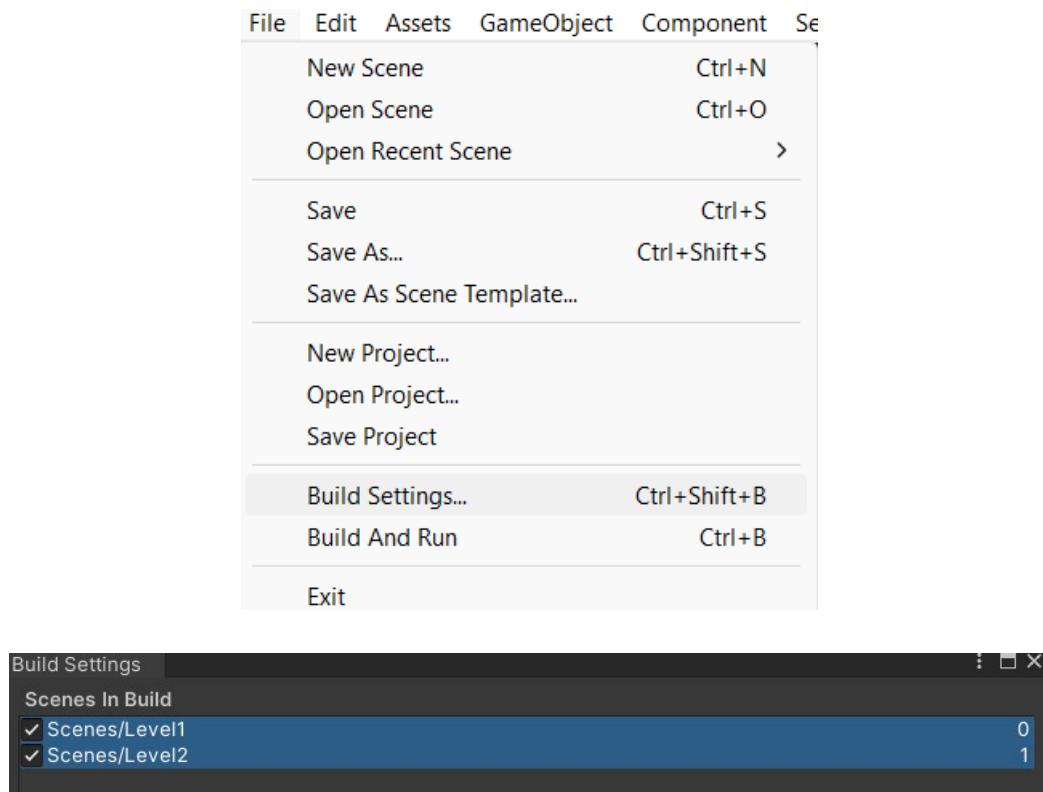


---

**99** In the Scenes folder, duplicate your level1 scene and rename it **level2**.

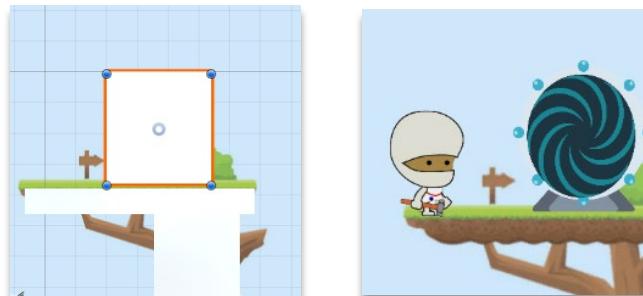


**100** Add this new scene to the **Scenes in Build** list located in **Build Settings**.



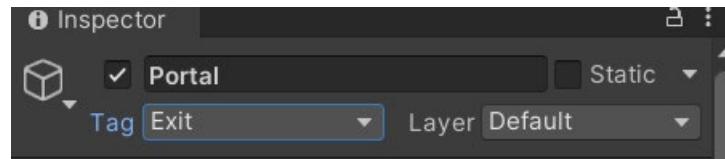
**101** Open the Level 1 Scene. Create a Quad object and rename it portal. This object will be used to teleport our Avatar to level 2 only if all the enemies have been destroyed. You can replace the Quad object with any other game object you like. In our example project we found a portal and are going to use it instead of the Quad.

Position the Portal somewhere at the end of your level. We are placing our portal at the top right of our scene.



---

**102** Create a new tag named **Exit** and assign it to the Portal game object.

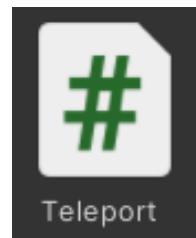


---

**103** Add a **Box Collider 2D** to the object, and make sure it has no other collider components.

---

**104** Create a new script in the Scripts folder named **Teleport**. Attach this script to the Portal game object.



---

**105** Open the Teleport script in Visual Studio.

We first need to find out how many enemies are in the scene. Since they all have the tag **Enemy**, it is easy to find out how many there are. Create a **public int**, name it **enemyCount**.

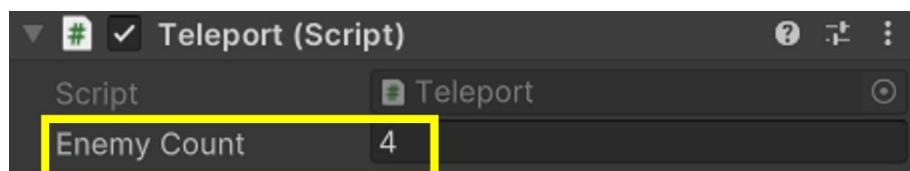
```
public int enemyCount;
```

**106** In the Teleport script's **Start()** function we want to use the **FindGameObjectsWithTag** function and **.Length** to get the total number of enemies.

```
void Start()
{
    enemyCount = GameObject.FindGameObjectsWithTag("Enemy").Length;
}
```

**107** Save your script and return to Unity.

Playtest your game. Select the Portal game object, what number do you see in the **enemyCount** variable?



In our example, Enemy Count equals four because we have 4 enemies in the scene.

**108** On your own, add code to the **EnemyMovement** script that subtracts one from enemy count whenever our avatar removes an enemy.

The conditional logic is already in the **EnemyMovement** script, we just need to connect it to our **Teleport** script.

 **Sensei Stop**

Demonstrate to your Sensei that the **enemyCount** variable is decreasing every time you destroy an enemy.

**109** Return to the **Teleport** script. To move on to the **level2** scene, the Avatar will have to collide with the portal after the player has destroyed all the enemies in the scene.

We will create a **OnCollisionEnter2D** function with a conditional that will check for 2 things. First, it will check to see if the Avatar collided with the portal by verifying that the colliding object has the "Player" tag. Second, it will check to see if **enemyCount** is equal to zero.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.gameObject.tag == "Player" && enemyCount == 0)
    {
        // Teleport code here
    }
}
```

- 
- 110** If both conditions are true, to the game will load the level2 scene. To use the SceneManager, we need to add **using UnityEngine.SceneManagement** at the top of our script.

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

- 
- 111** Back in the conditional statement, use the SceneManager's **LoadScene** function to load the scene that is in build index 1.

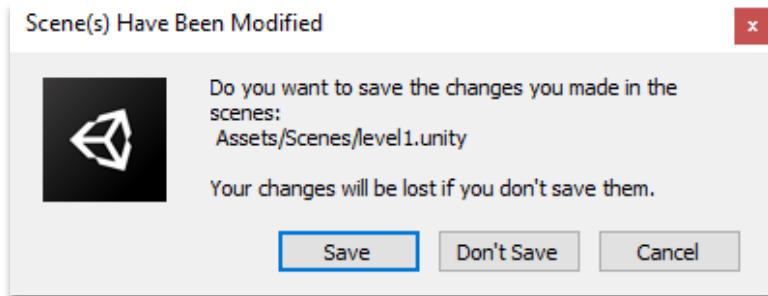


```
private void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.gameObject.tag == "Player" && enemyCount == 0)
    {
        SceneManager.LoadScene(1);
    }
}
```

---

**112** Save your script and return to Unity. Playtest your game! Destroy the enemies and walk into the portal. Do you move on to the level2 scene?

Stop your game. In the Scenes folder, open up level2. If Unity asks you to save your changes, select **Save**. If you don't, your work will be deleted.



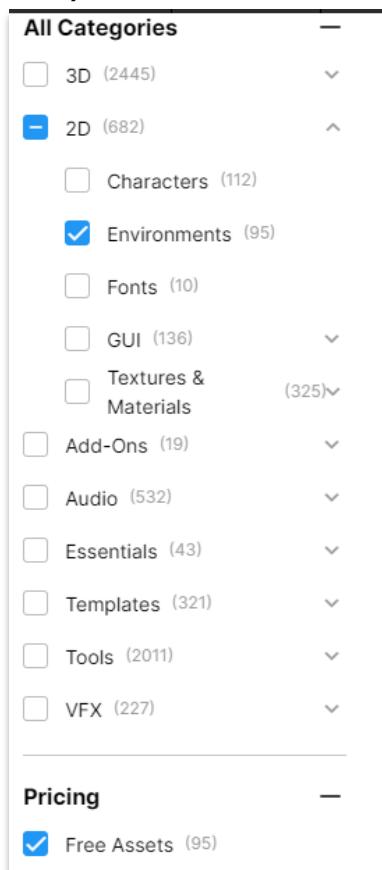
---

**113** You might be wondering what we are going to do in level 2. It is now your turn to create your very own 2D scene using the Unity Asset store!

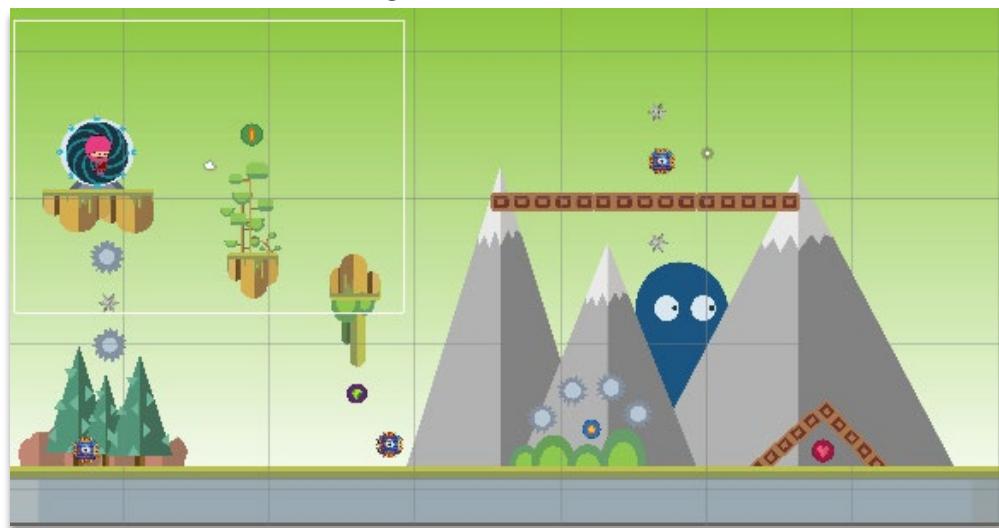
For level 1, we provided 2D backgrounds that need Quad objects to create floors and walls. For level 2, you can download and import Artwork and Prefabs to create your own complete scene.

---

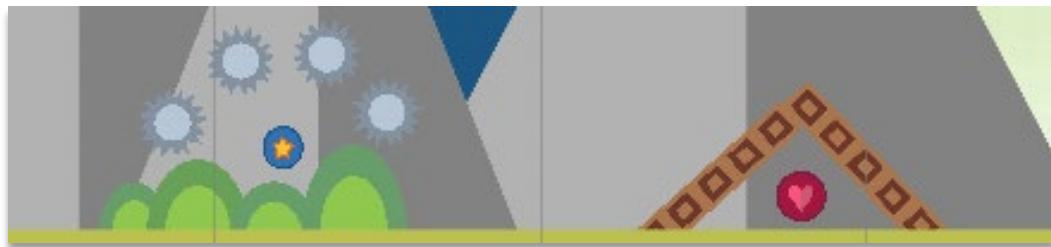
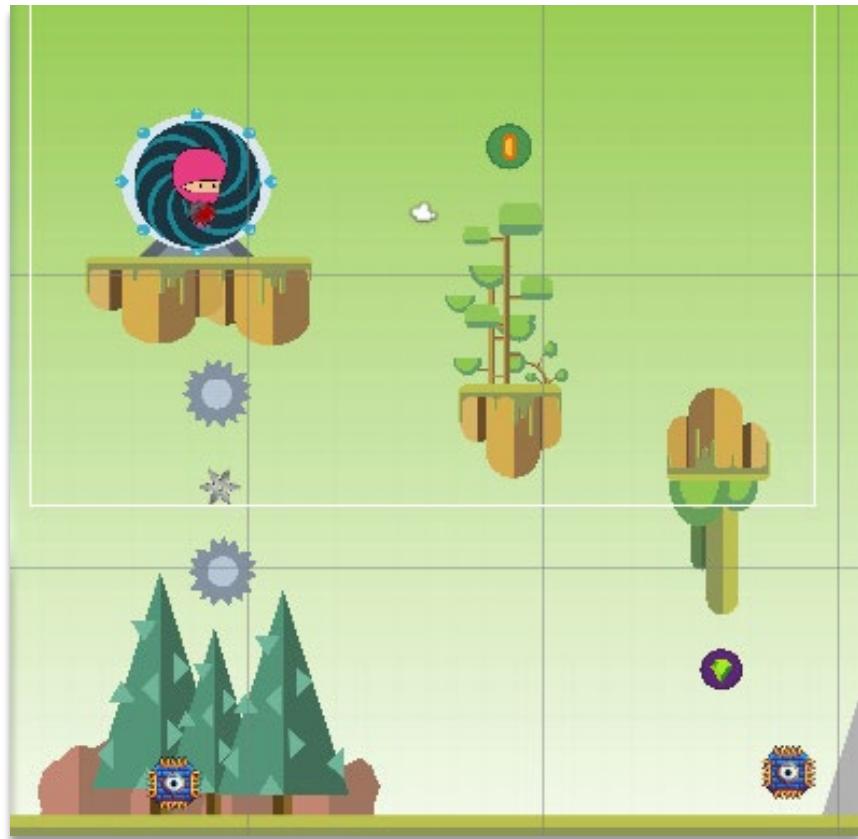
**114** Go into the Unity Asset Store tab. In the Categories, apply the filters we have below so you can find Artwork you like.



**115** Go through the different artwork and import it into your project. Look at what we have made below using different downloaded art!

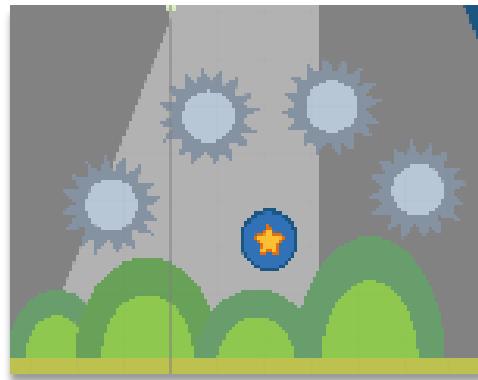
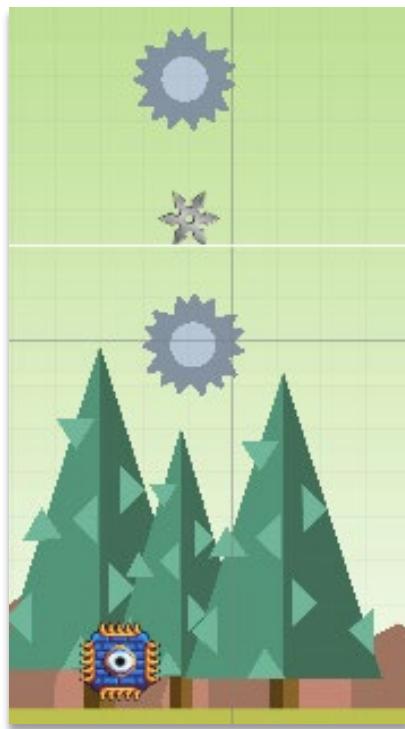


**116** In this level, you have the option to add more enemies or more collectables to make your level more challenging and fun. We have added additional collectables all around the scene that the Avatar must collect to pass level2.



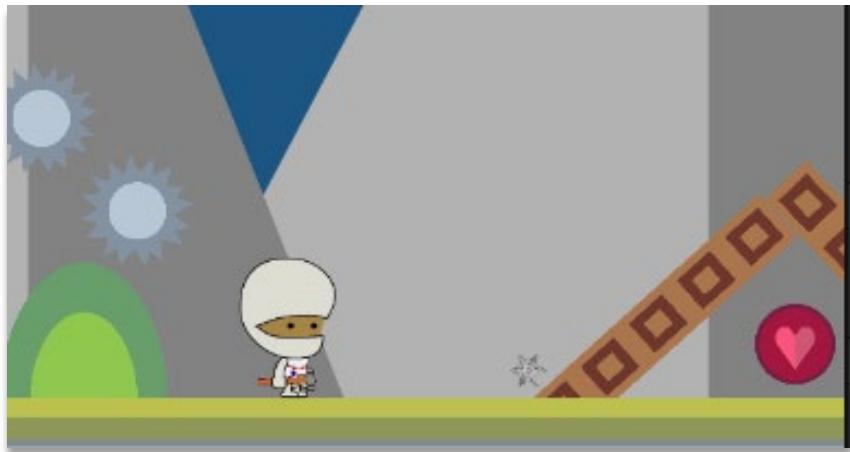
**117**

The crushers are still our enemies, but now there are additional traps! There are rotating spikes that resets Avatar if it gets too close to them.

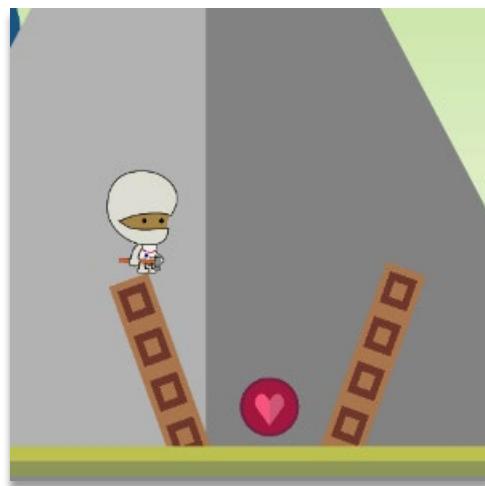


**118** You can even block off a collectable and program a way for the player to unlock it!

Can the Avatar throw an object and break the barriers to unlock it?



Or is there a switch that opens the gates for the Avatar?



---

**119** We have also modified level1 in order to align better with level2. In game design, it is always ok to go back and make changes to your game so things can match up together.

Instead of keeping the Quad object as our Portal, we replaced it with a Portal object we found in the Unity Asset store!

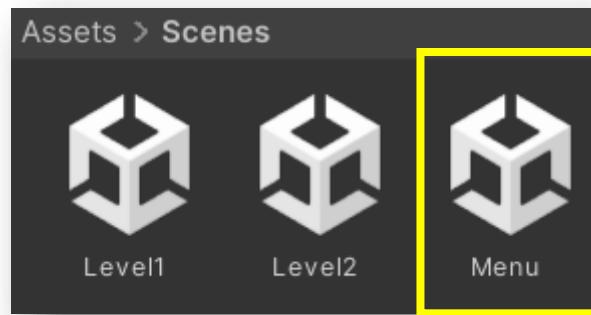


---

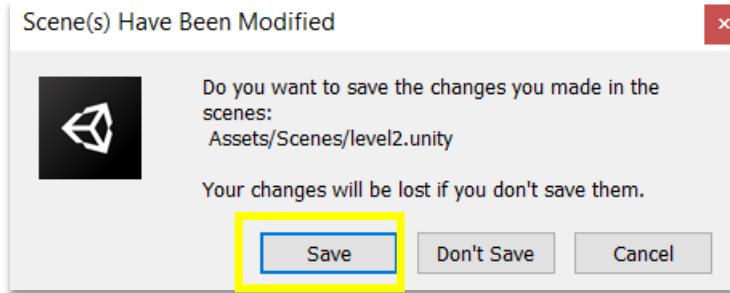
**120** Refer to your Gravity Trails Planning Document to build out the scene you planned!

## Creating a Start Screen

- 121 In the **Scenes** folder, create a new scene with the name **Menu**.



- 122 Open the scene! If you are prompted to save your current scene, make sure to click **Save**. This way you do not lose your work.



**123**

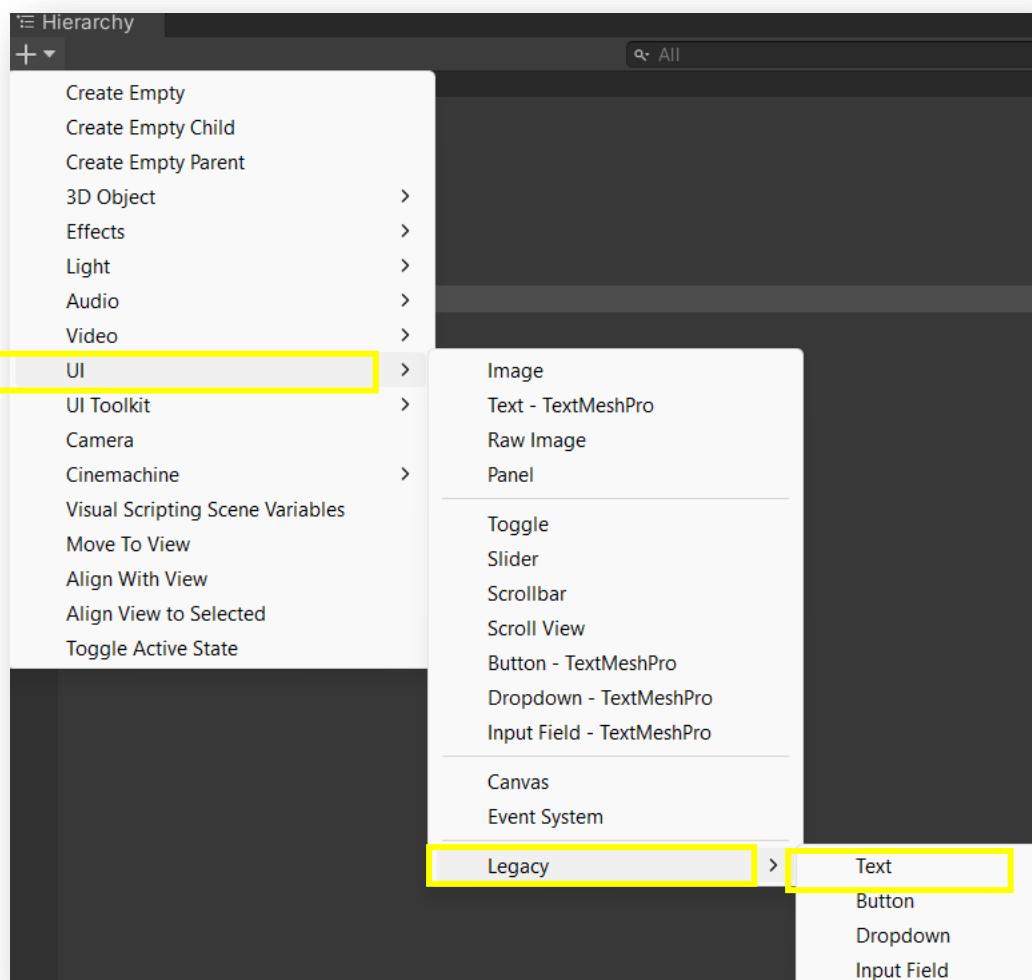
Our game menu will display four things:

the title of the game,  
a background,  
a start game button, and  
the controls the player needs to know to play your game.

Open both the **Scene** and **Game** tabs so it is easier to see what the Canvas looks like when the game has started.

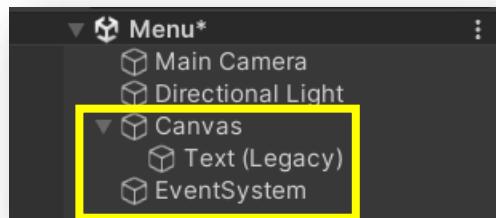
**124**

In the Hierarchy, create a new **UI Legacy Text** object. This object will display the title of our game.



---

This will create a **Canvas** with your **Text** inside of it and an **EventSystem** object.



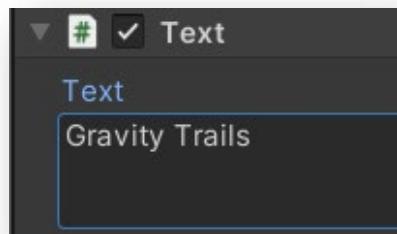
---

**125** Rename the Text object to **Title**.



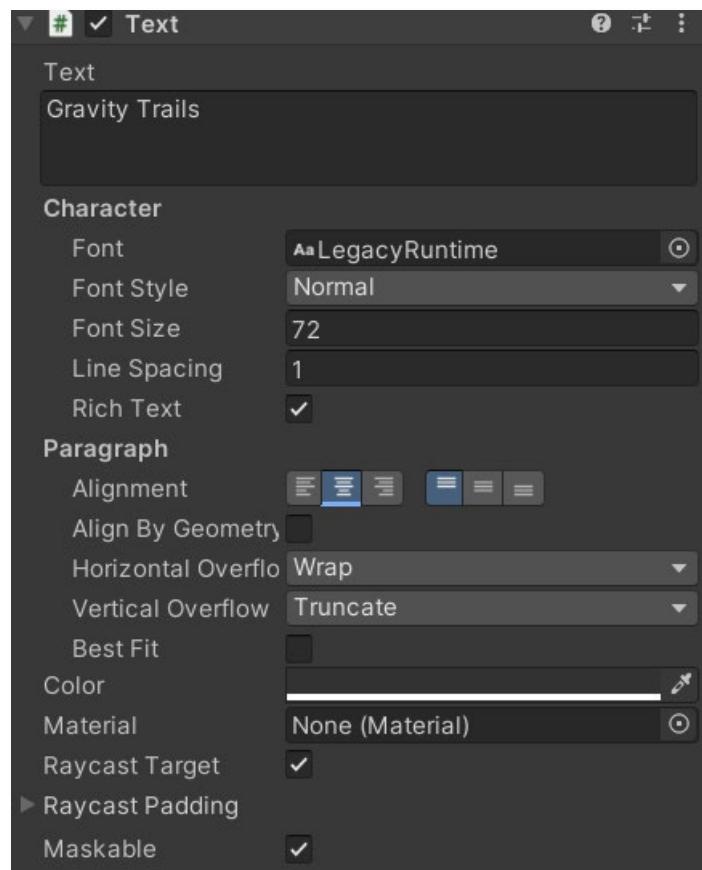
---

**126** Click on the Tile object and in the **Inspector**. Change the value of the **Text** property to the title of the game.



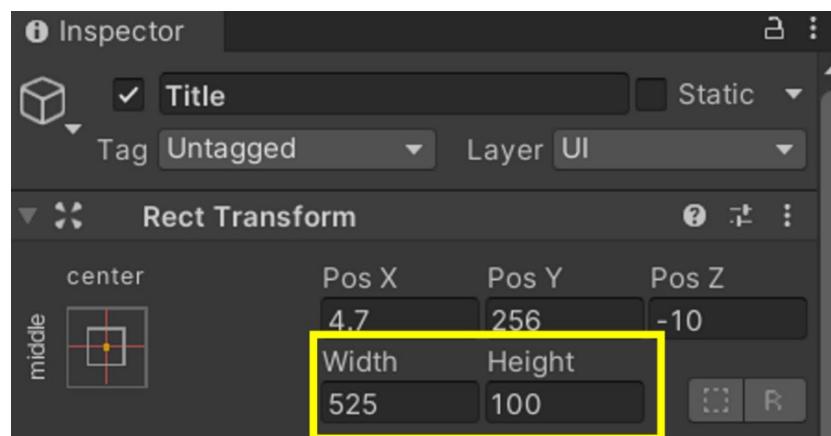
**127**

Change the other properties of the Text component to customize your title.

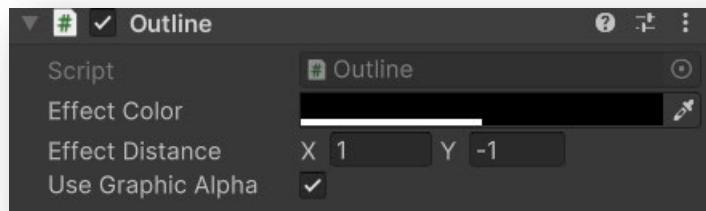


**128**

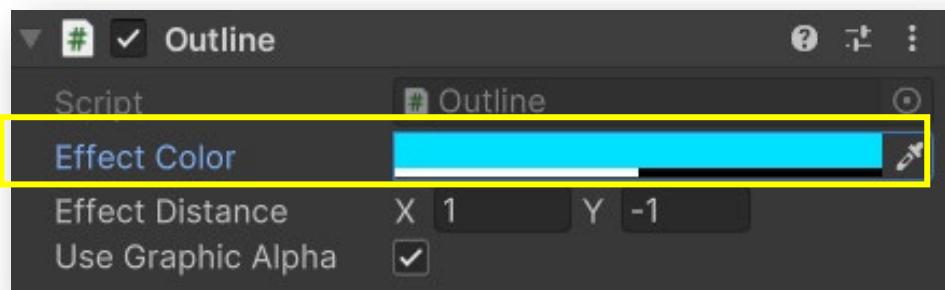
If you are unable to see the changes you are making, you might have to adjust the **Width** and **Height** of your text box.



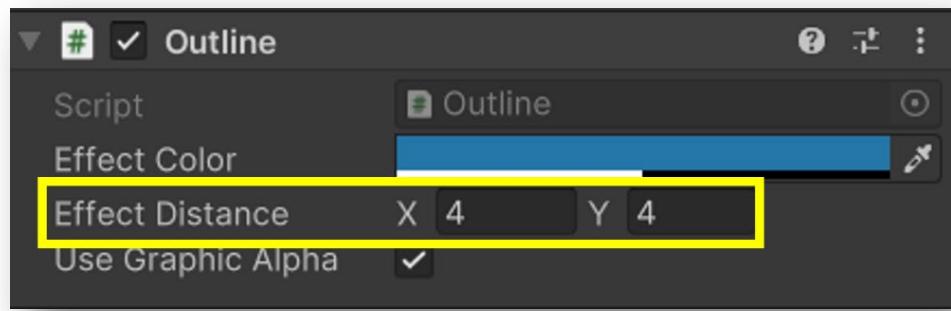
**129** Another way we can customize the Title is by adding the **Outline** component. Click on **Add Component** in the Inspector, search for **Outline**, and click on it.



**130** Change the **Effect Color** to add a border to the text.

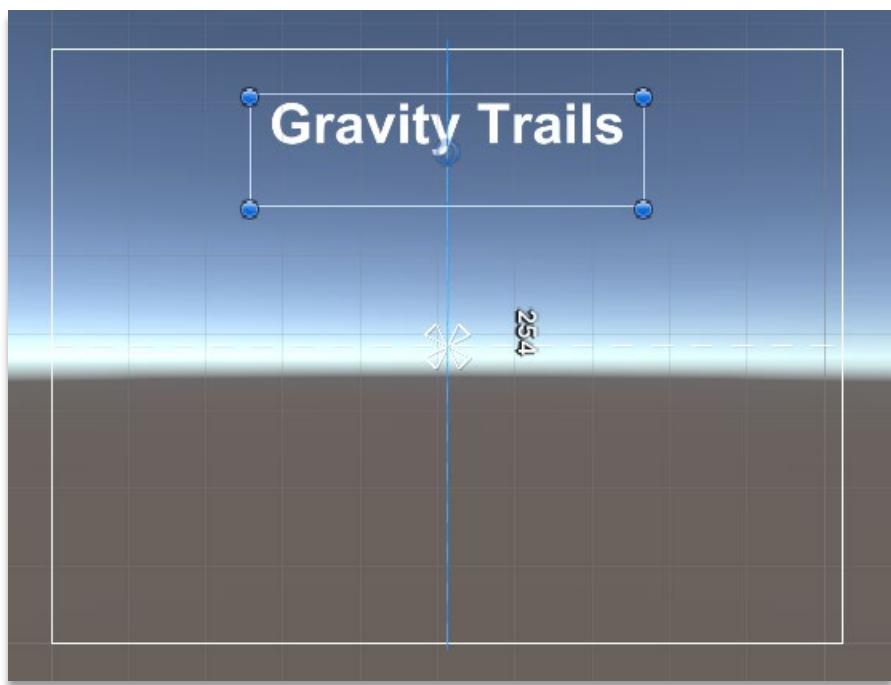


**131** Change the **Effect Distance** to make the outline smaller or larger.



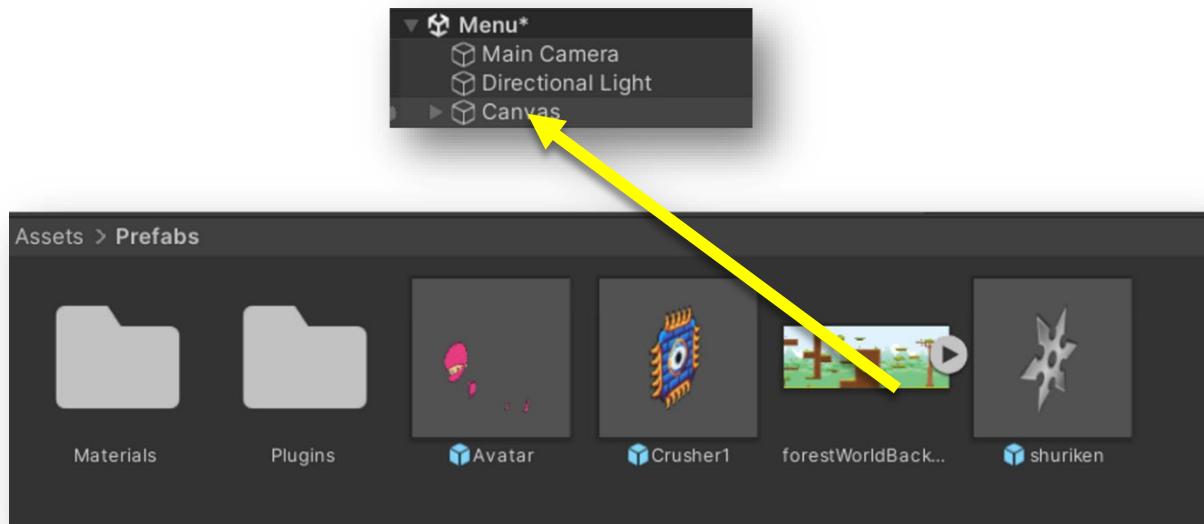
**132** Use the **Rect Tool** to center the Title on the Canvas. You will know it is centered when a Vertical Blue line appears in the screen. You may need to resize the textbox to a larger size to see the blue line.



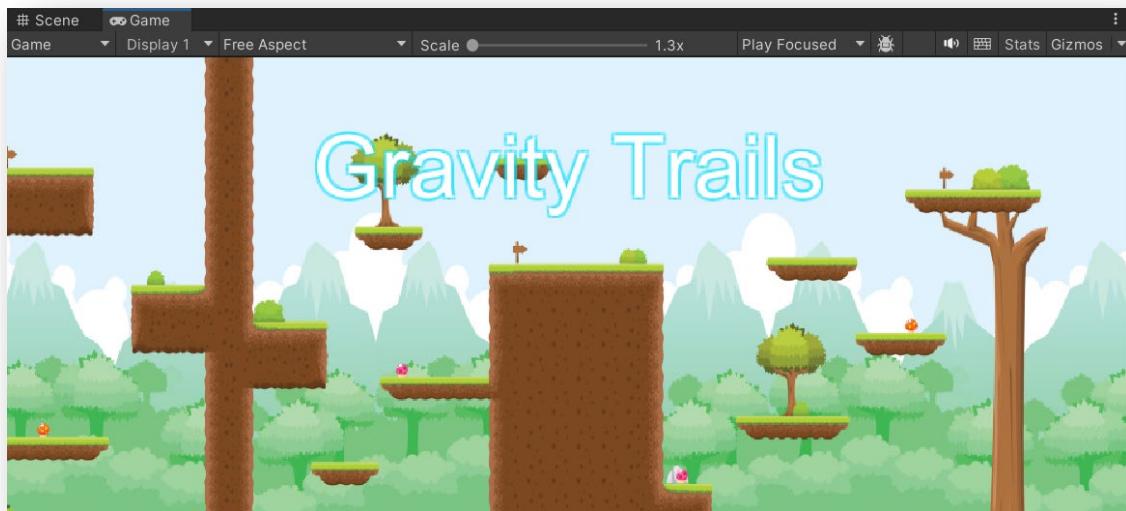


**133** The Start Screen is not quite complete! Let's work on the background. You can use the **forestWorldBackground** that we used previously or find a new background in the Unity Asset store.

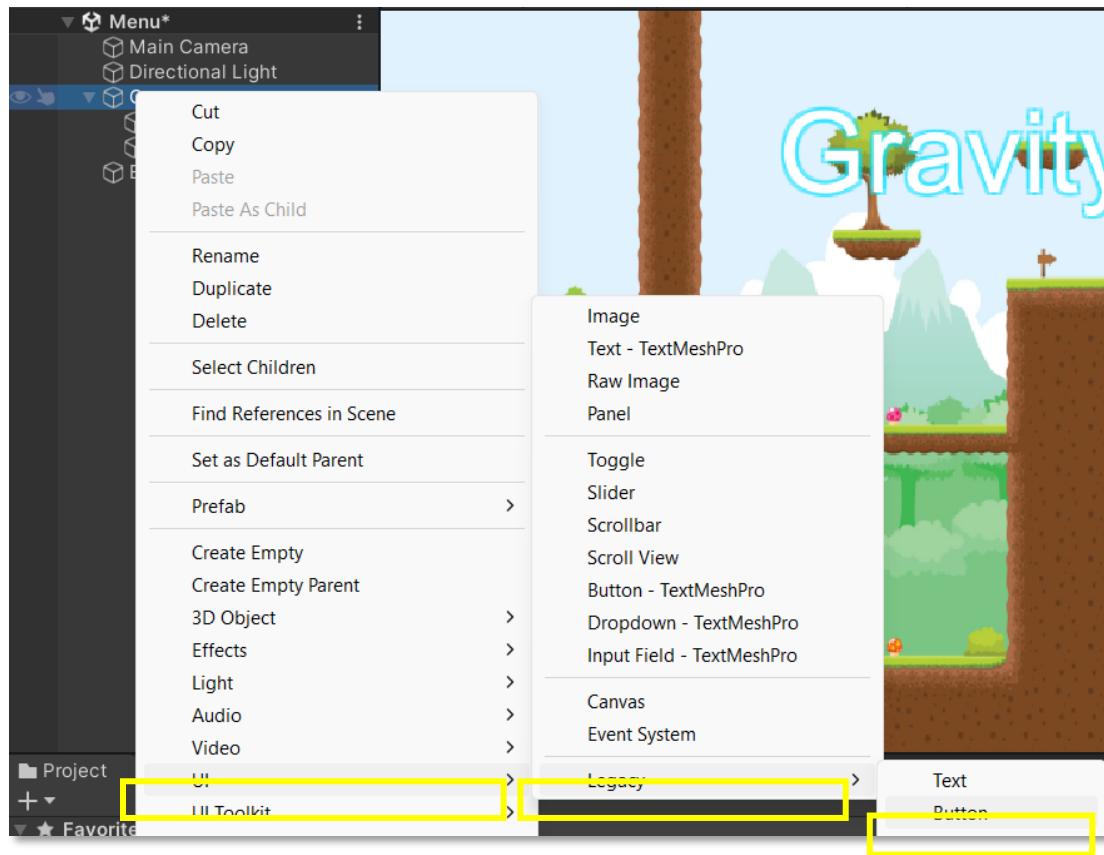
Once you have the background you want, we are going to add it to the scene. Drag the background you chose into the Canvas in the hierarchy. If you don't see the background in the game view, make sure that the background image is positioned where your camera is.



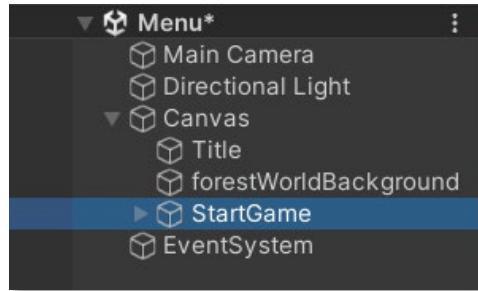
**134** Your background image should now be visible on the game screen. Use the **Move Tool** to adjust the position of your image.



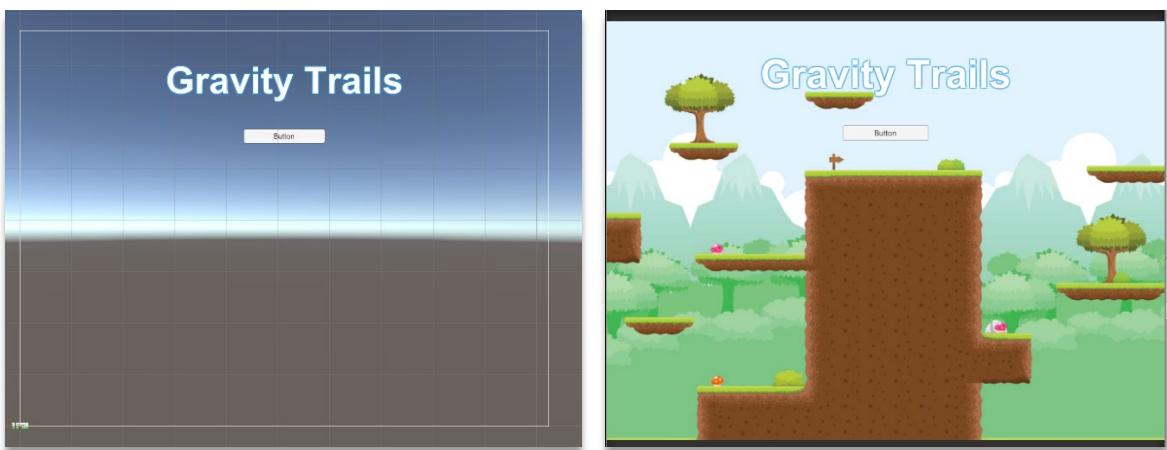
**135** Now that we have our background setup, let's move on to creating our **Start Game** button. In the Canvas object, right click and go to **UI** then **Legacy** then **Button**.



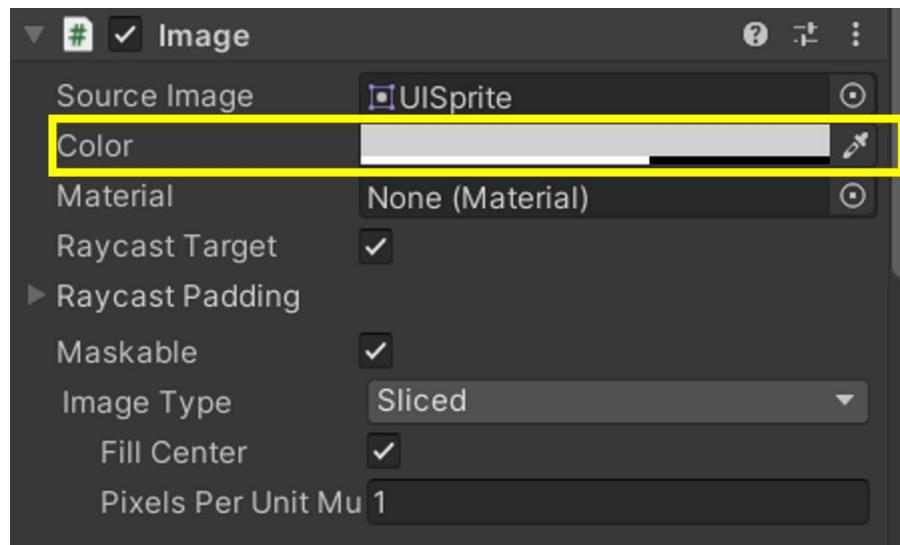
**136** Rename the button to **StartGame**.



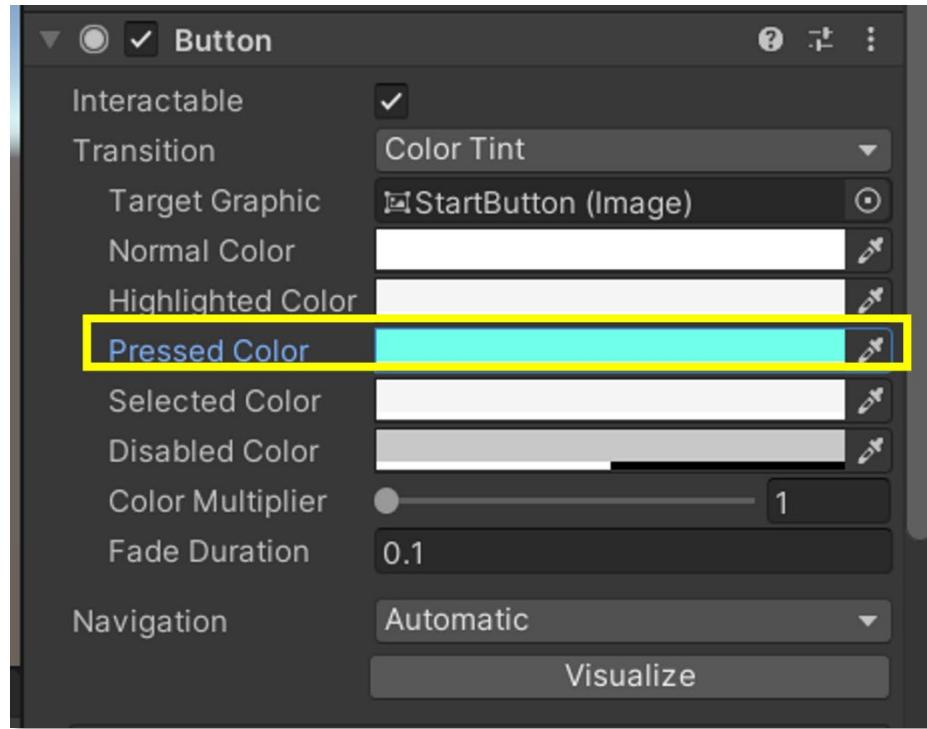
**137** You should see a button created in the middle of the scene. Feel free to position it anywhere that is easy for the player to see it.



**138** We can customize the buttons appearance by making changes to the **Image** and **Button** component in the Inspector. Try changing the **Color** of your **button!**



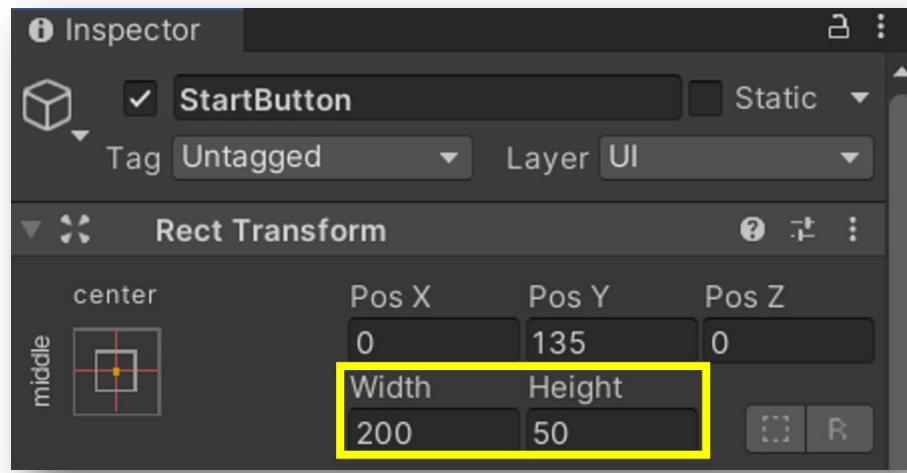
**139** In the **Button** component, you can change the **Pressed Color** so when a player wants to start the game, they are given feedback that the button has been pressed.



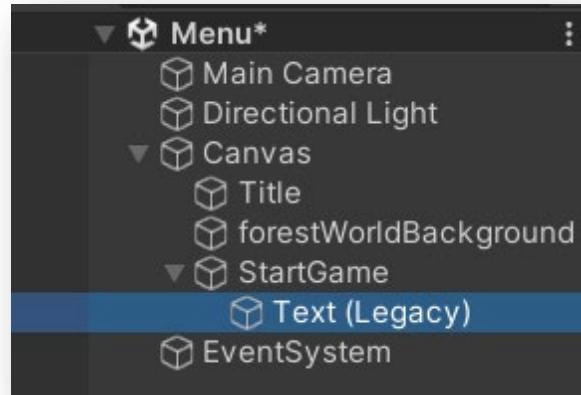
**140** Playtest the scene and click on the Button. Notice how it changes colors when clicked.



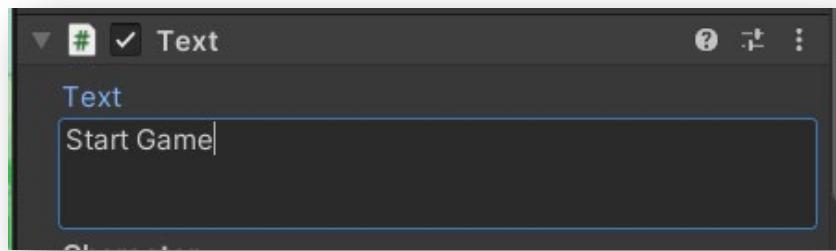
**141** Stop your game. If you feel your button is to small adjust the **Width** and **Height** to make it larger.



**142** The last part of our button that we want to customize is the **Text**. Expand the StartGame object and select the **Text**.

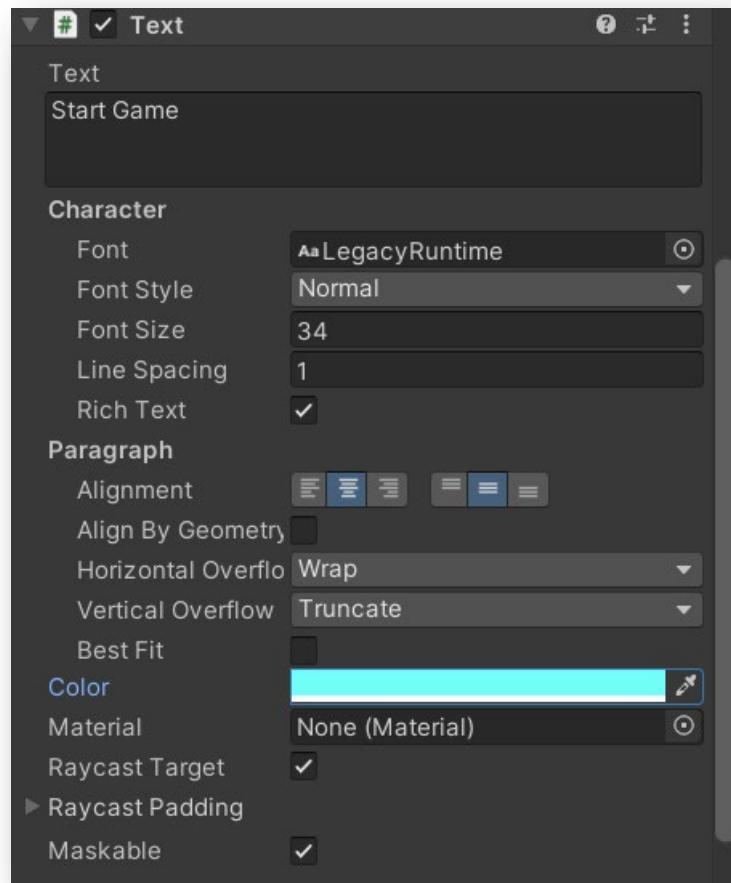


**143** In the Inspector change the **Text** value to **Start Game**.



**144** Customize the text to your liking by using the other options in the **Text components**.

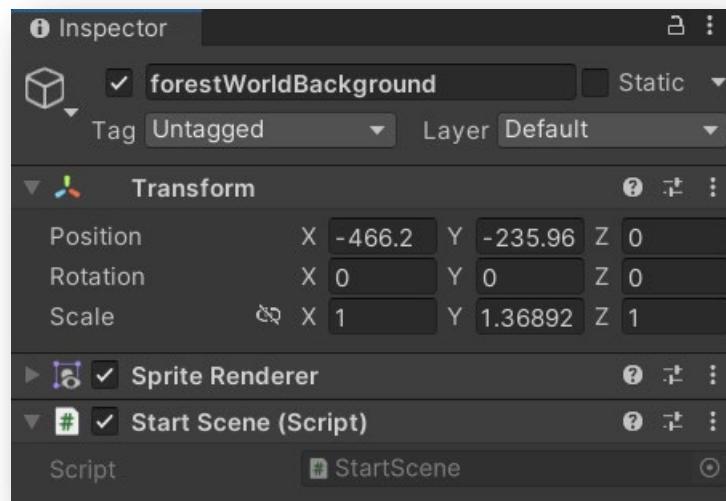




**145** We need to add logic that takes the player to the first level when the button is clicked. In the **Scripts** folder create a new script and name it **StartScene**.



**146** Attach this script to the background you are using. In our project, we are attaching the script to the forestWorldBackground.



**147** Open the **StartScene** script. We are going to use the **OnMouseDown()** function to load the level we want when the player clicks on the StartGame button. Create a **public void OnMouseDown()** function.

```
public void OnMouseDown()
{
}
```

---

**148** In order for us to change the scene we are on, we must include **using UnityEngine.SceneManagement** at the top.

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

---

**149** In OnMouseDown() function, load the level one scene.

```
public void OnMouseDown()
{
    SceneManager.LoadScene(1);
}
```

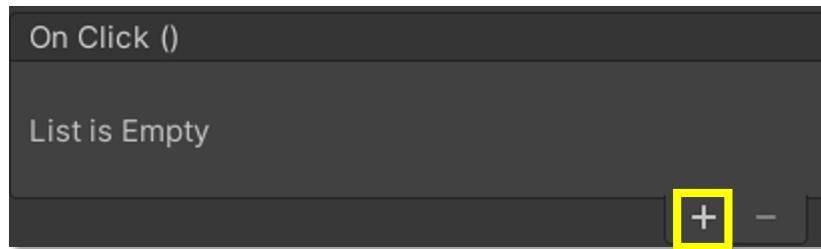
Double check your **Built Settings** to make sure that you call the correct scene that you want to load!

---

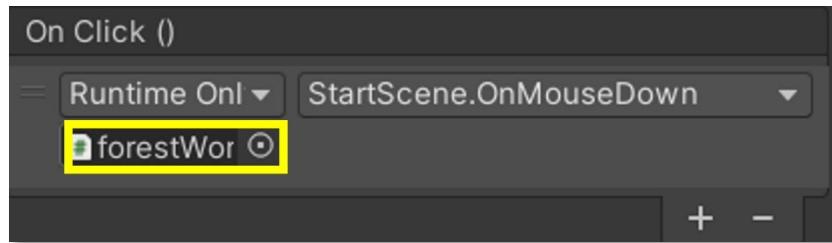
**150** Save your script and return to Unity. Playtest your game and try clicking on the Start game. The button still doesn't change the scene!

We need to connect our code to the StartGame button. Select the **StartGame** object in the Hierarchy.

In the Inspector scroll down until you find the Button component, and then the **OnClick()** component. Click on the **+** so that we can connect our **StartScene** script to this button.

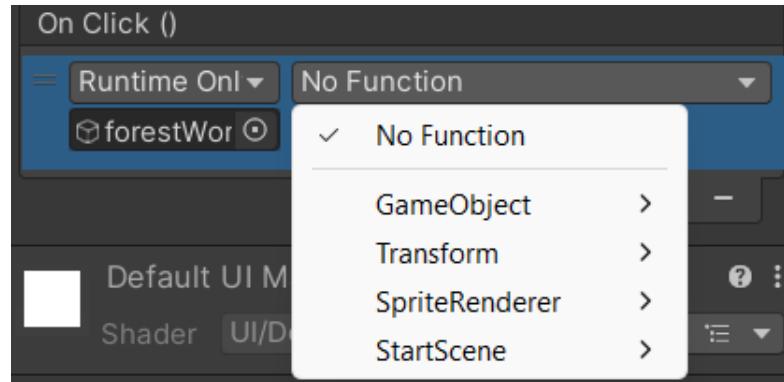


**151** You should now see three slots, we first need to fill the one that says "None" with the object that has the StartScene script. Drag in the background you are using into that component.

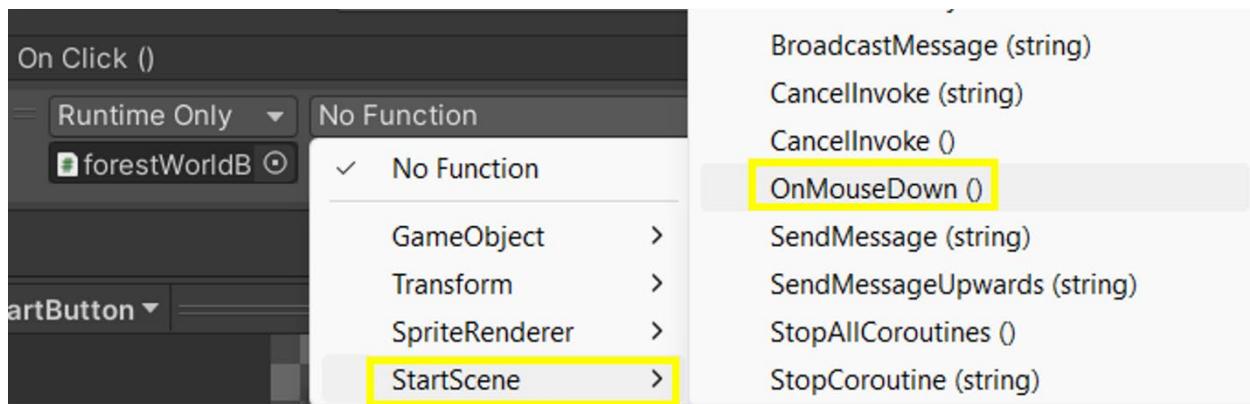


Notice how we can now change the property on the right that says **No Function**.

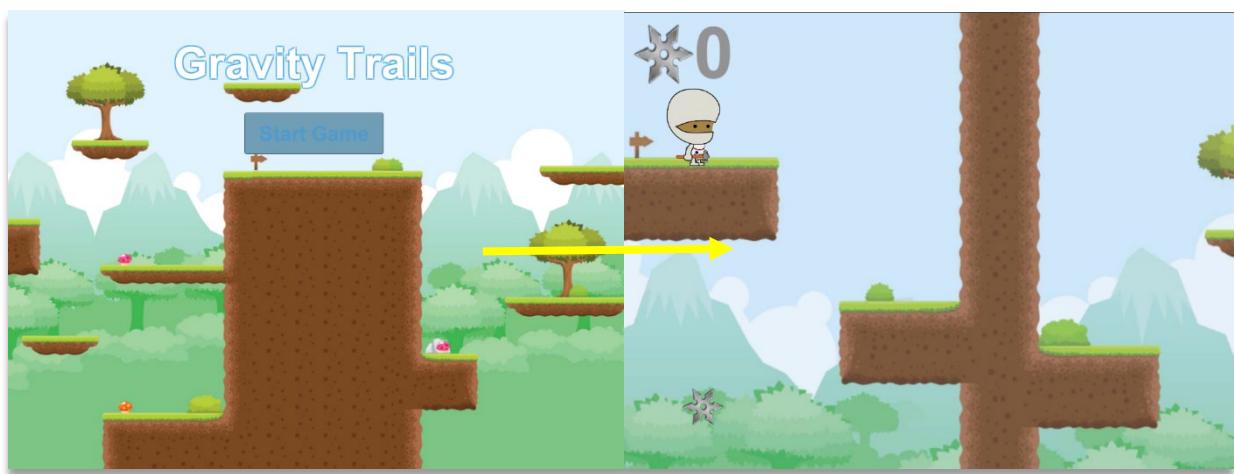
**152** The component on the right is how we tell the button what script to look at and the function to run. Click on the arrow next to **No Function** to see the options we have.



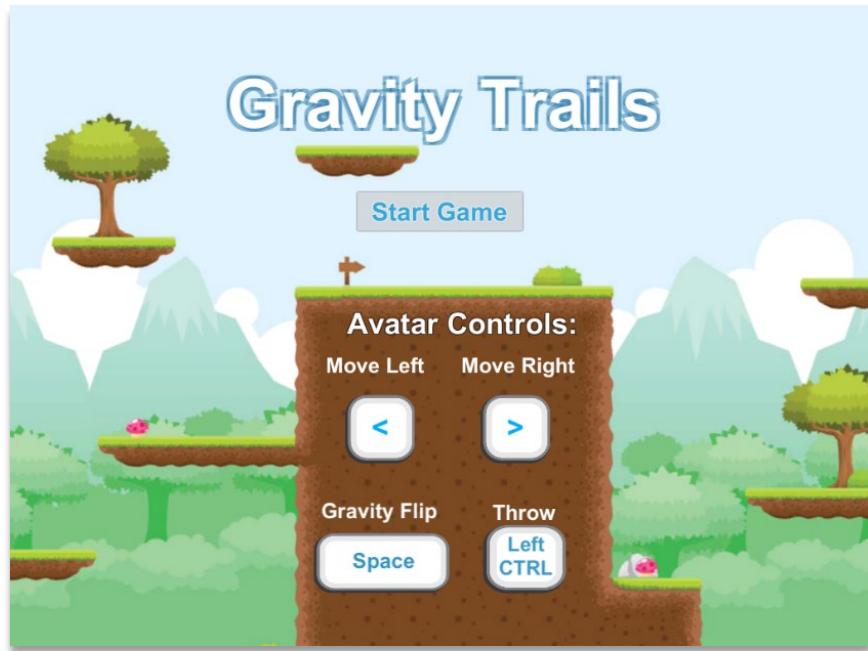
**153** We want to access the function found in the **StartScene** script. Click on **StartScene** then **OnMouseDown()**.



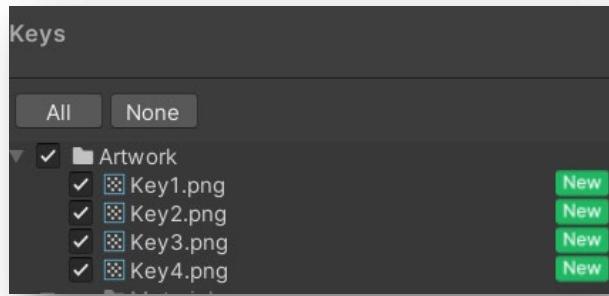
**154** We programmed our **OnMouseDown()** function so that the game takes the player to level1. We then connected the logic in our script to our button. Playtest your scene, when you click the StartGame button does it load level1?



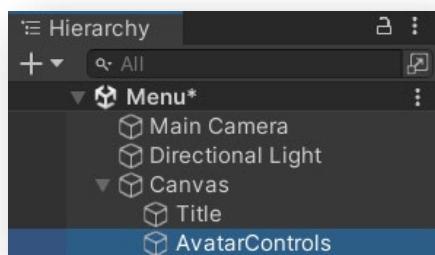
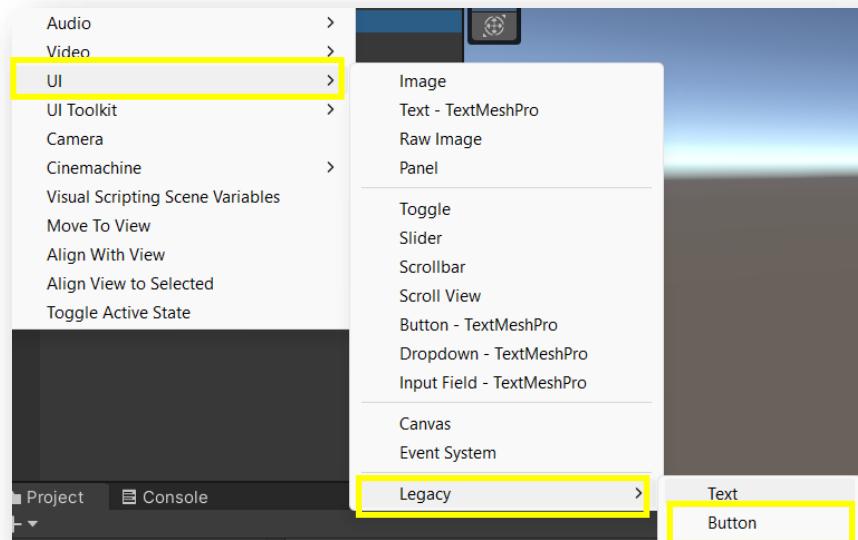
**155** Finally, we will add the controls the player needs to know to play the game! It will look something like the image below.



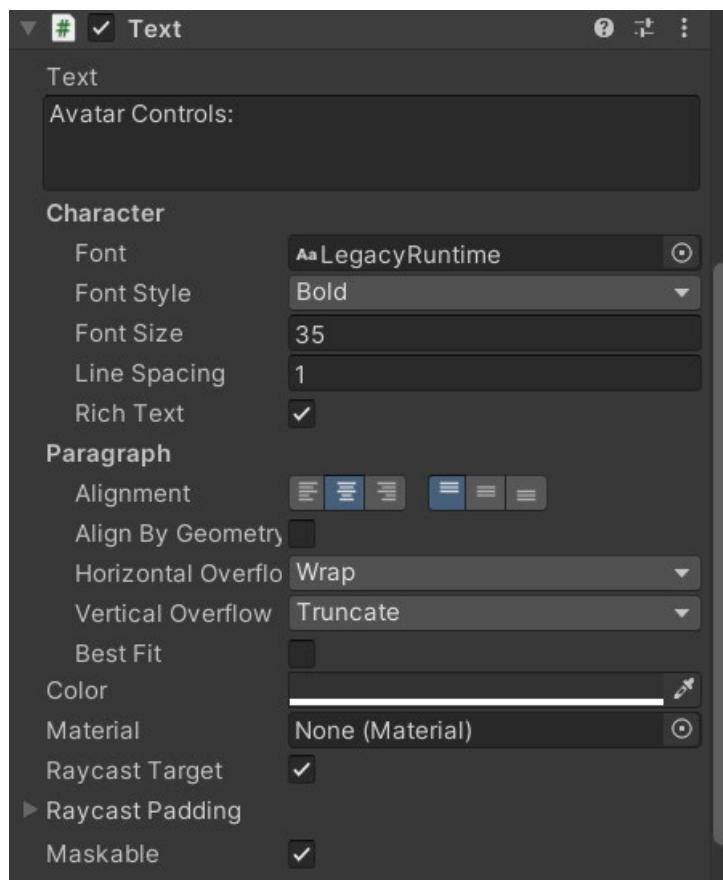
**156** Import the **keys.unitypackage**. This package contains the keyboard images we will use to make the keys.



**157** Create a **UI Legacy Text** object and rename it **AvatarControls**.



**158** In the Inspector, change **Text** so that it says **Avatar Controls**. Customize and position the text object to your liking. Look at the example below for some ideas.



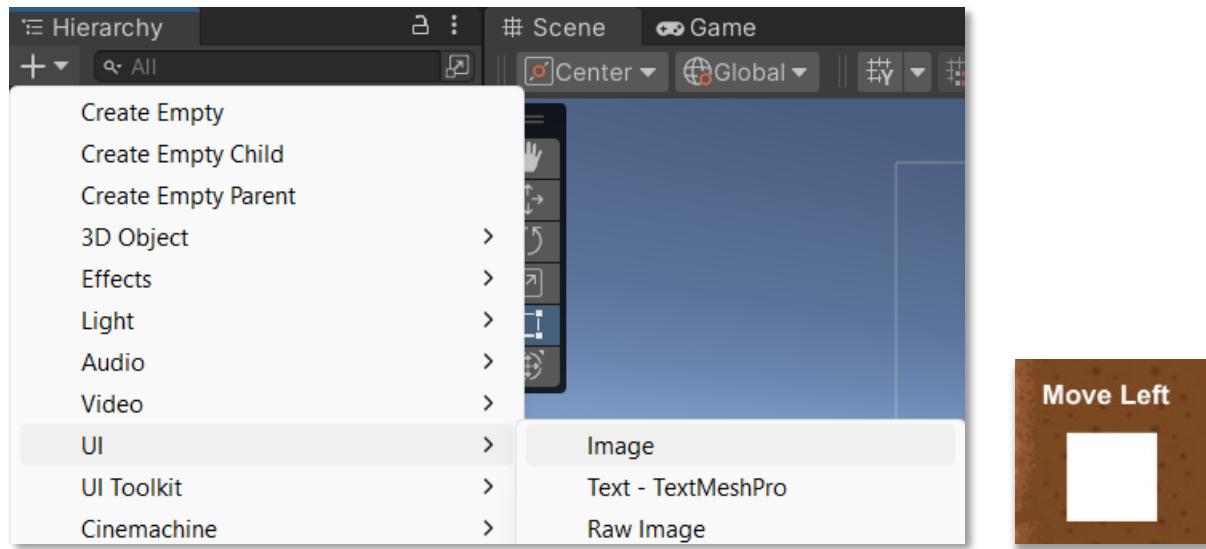
In this activity our main character is the Avatar, but in other games the main player might have a different name. You should change the name before the word "Controls" for other game menus that you create.

**159** We want to include all the controls the player needs to know to control their character. In this activity we need to show the player how to **Move Left**, **Move Right**, **Gravity Flip**, and **Throw**.

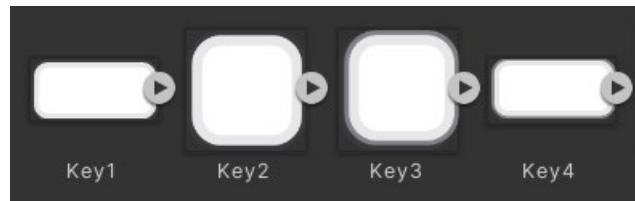
Create another text object and rename it **MoveLeft**. In the Inspector change the **Text** to “**Move Left**”. Customize the text and position it to your liking! Look at our example below:



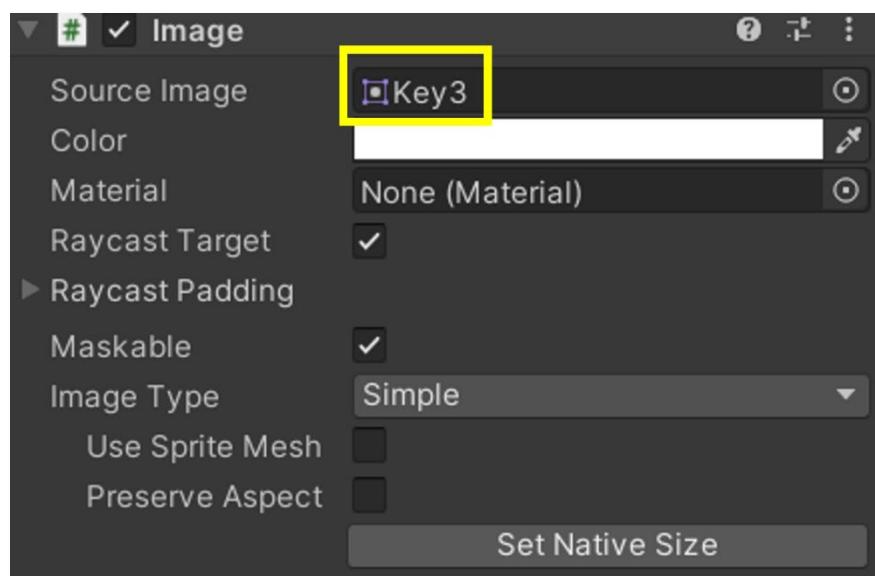
**160** Here we tell the player that they can move left, but using what keyboard key? Below this let's add a key image! Create **UI Image** object and rename it **key**. Position it below the text.



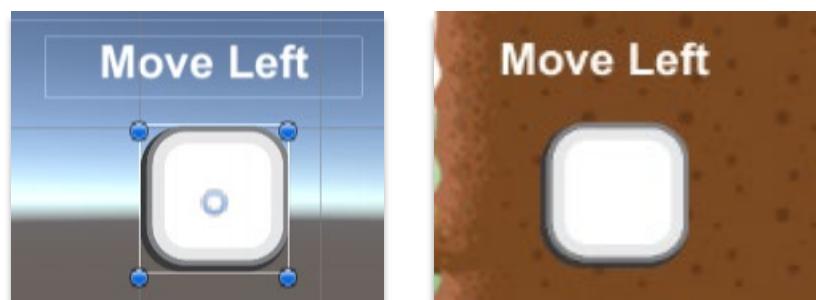
**161** We have made a few key images that you can use. In the **Artwork** folder you can find 4 different keys.



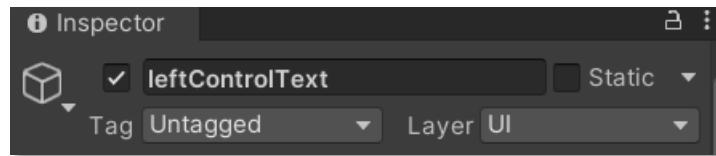
**162** Make sure you have the **key** object selected, and in the Inspector drag the key sprite that you like into the **Source Image** component.



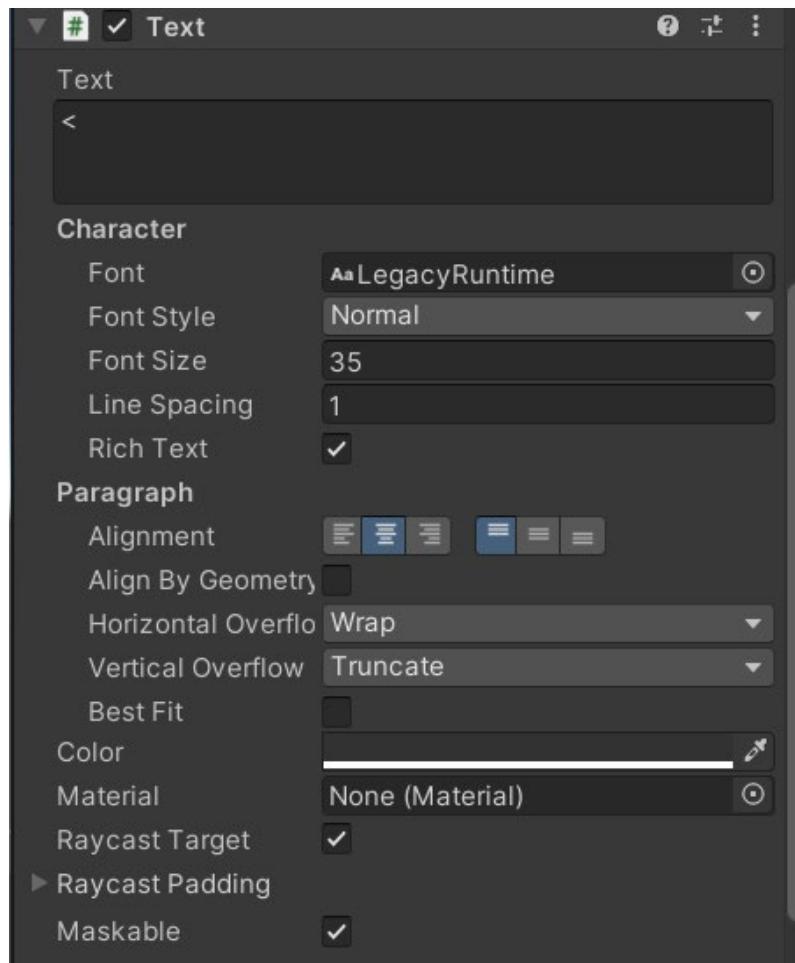
We used the **Key3** sprite. You should see the change in both the **Game** and **Scene** view.



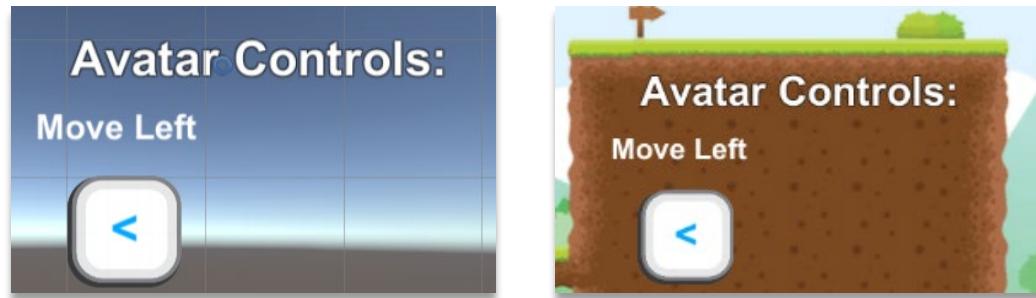
**163** To complete the left control image, we want to place a text object inside of our image. Create another **UI Legacy Text** object. Rename it **leftControlText**.



**164** In the Inspector change the text to a left facing arrow < and customize it to your liking!

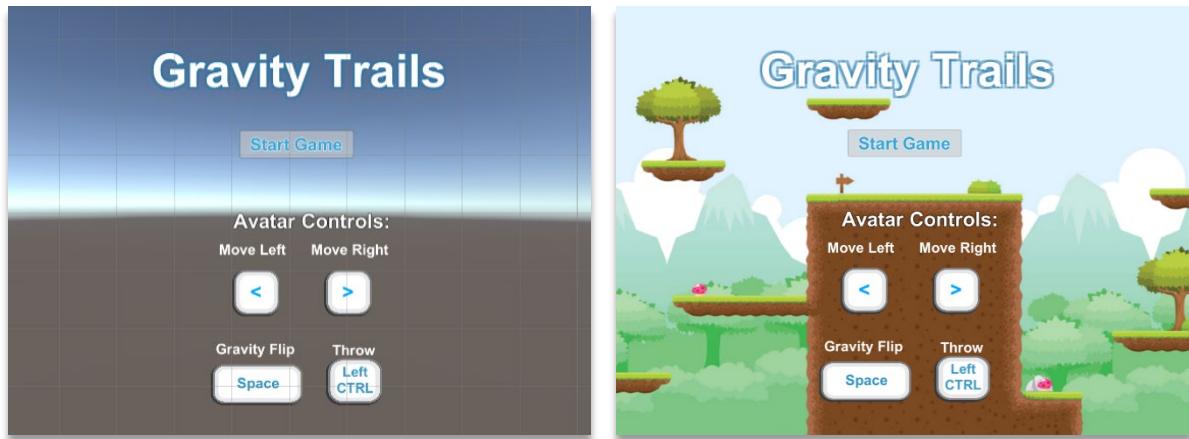


**165** Your key that tells the player how to move the Avatar to the left is complete.



**166** Like we mentioned earlier, the Avatar has three more controls in this game. Repeat steps 159 through 165 to create the rest of the controls.

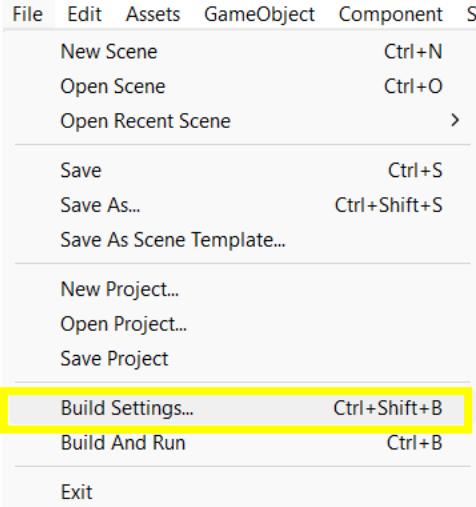
At the end you should have four total controls like the example below!



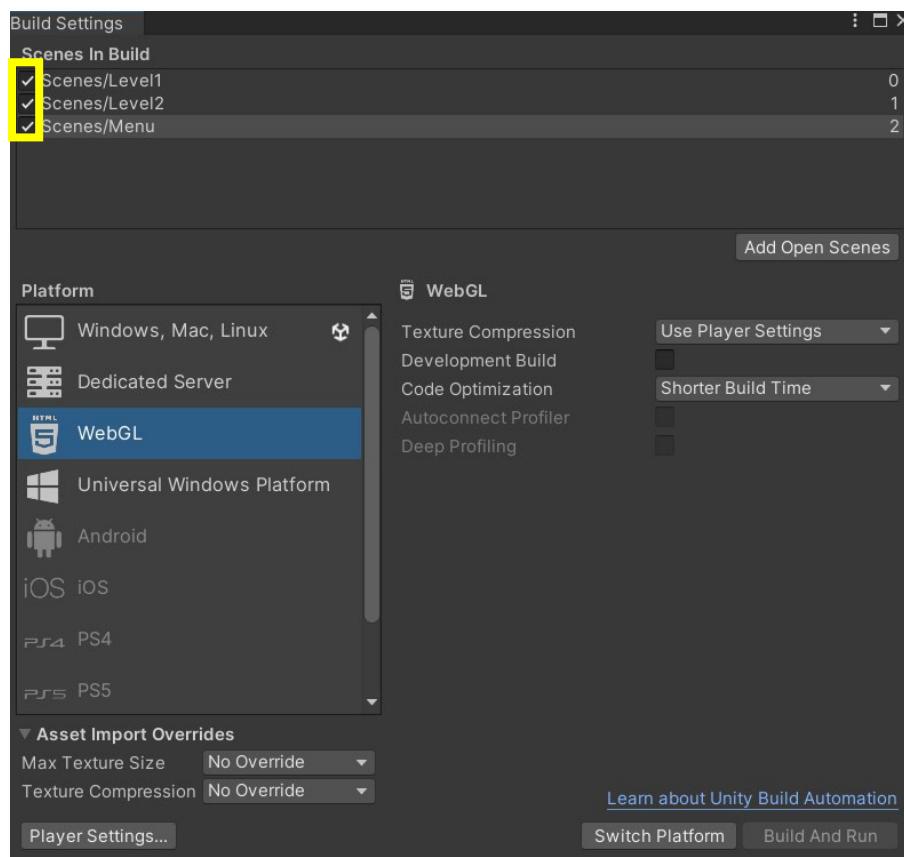
**167** Playtest your Menu and have fun on level 1!

## Sharing Your Unity Project

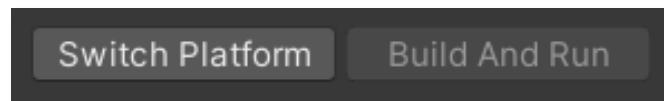
**168** In Unity, click File, then Build Settings.



**169** Make sure all scenes are checked. Under Platform, click WebGL.



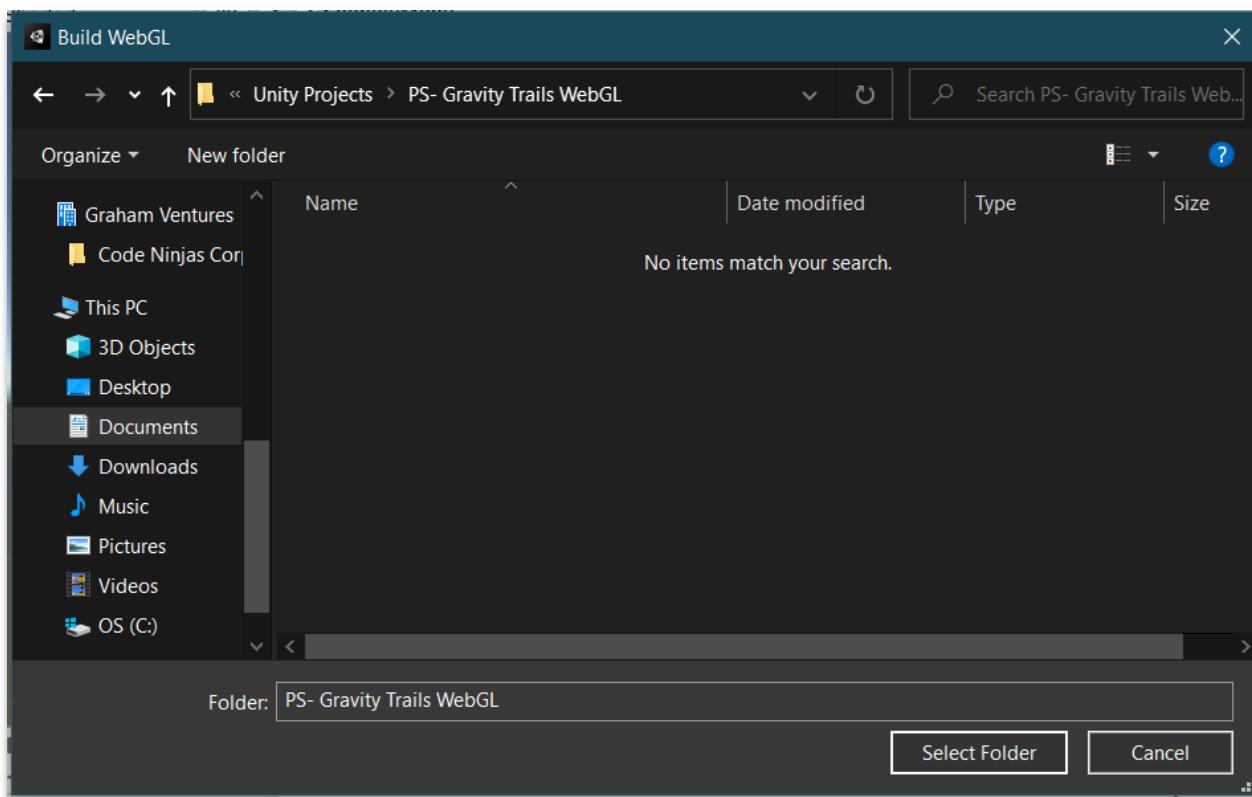
**170** Click “Switch Platform.” This may take a few minutes.



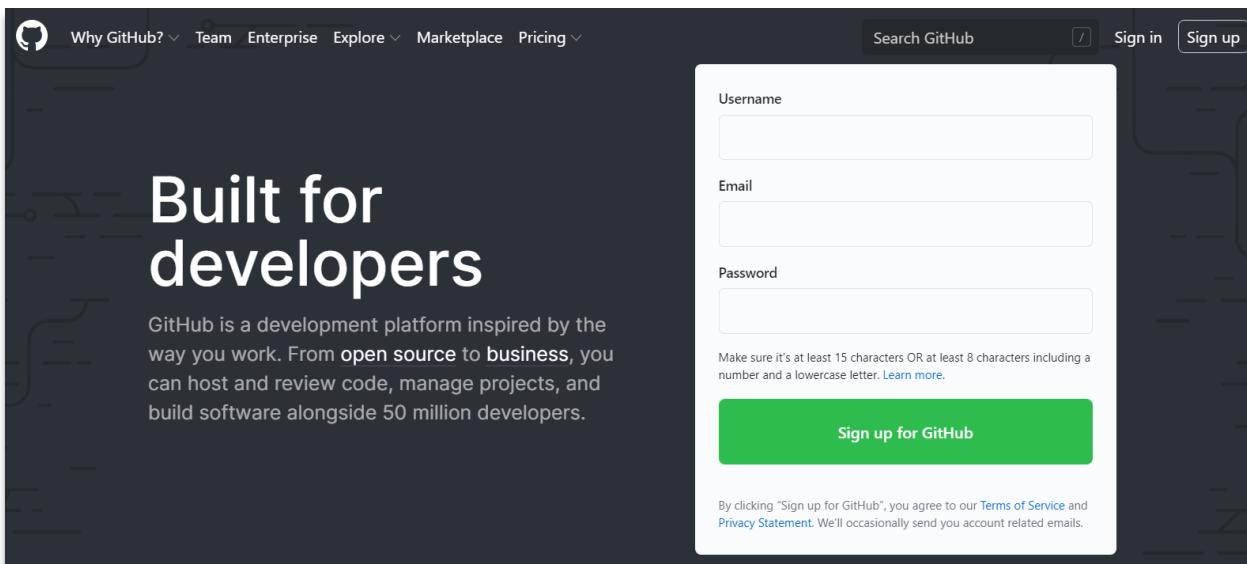
**171** Click Build And Run.



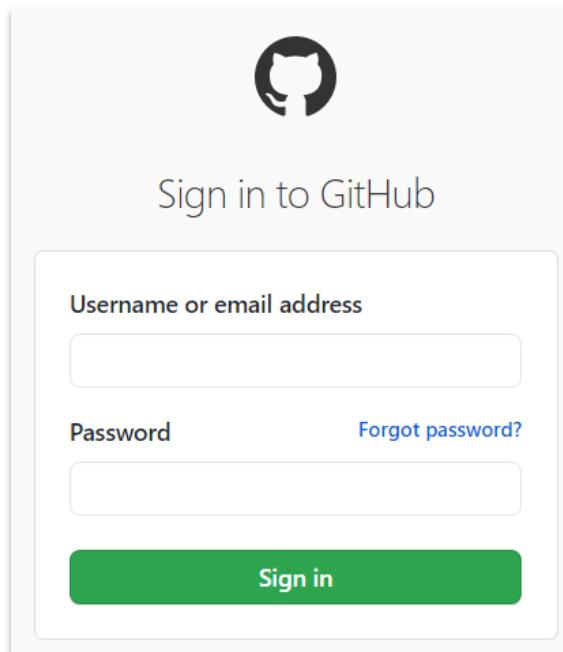
**172** Create a folder and save the WebGL files in a location you will remember.



**173** Go to GitHub.com and sign in. If you do not already have an account, create a free account.



**174** Sign into GitHub.



**175** In your dashboard, click the “+ New Repository” button.

Repositories

New

**176** Create a new repository.

### Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*

pollysmith ▾

Repository name \*

Name your repository,  
most likely something that  
reminds you of your game.

Great repository names are short and memorable. Need inspiration? How about [potential-system](#)?

Description (optional)

 Public

Anyone on the internet can see this repository. You choose who can commit.

 Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you’re importing an existing repository.

Add a README file

This is where you can write a long description for your project. [Learn more.](#)

Check the box that says  
Add a README file.

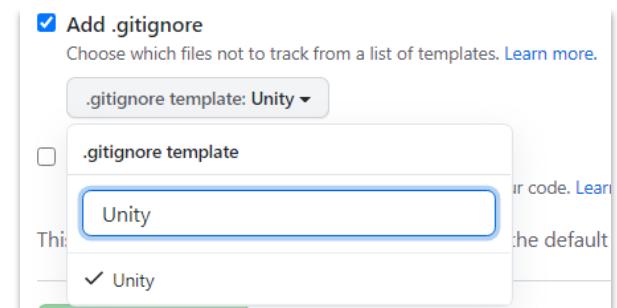
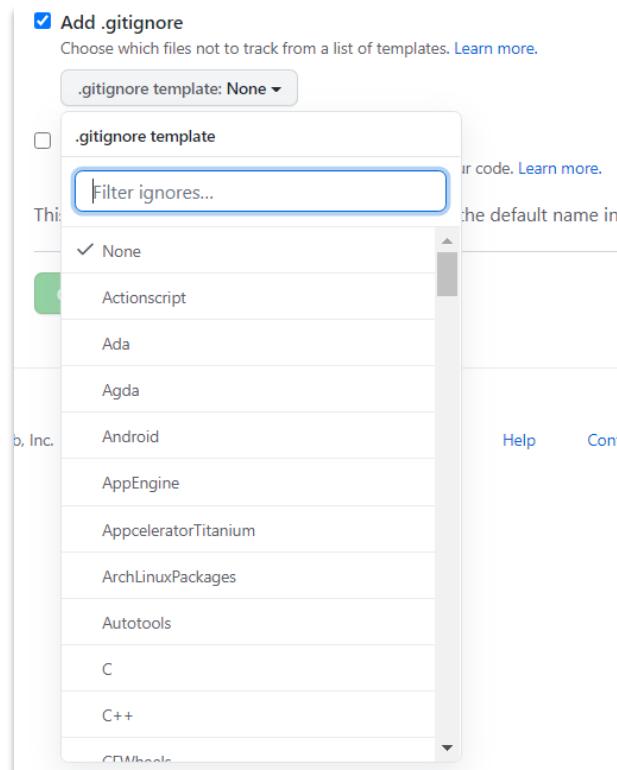
Add .gitignore

Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: None ▾

Click the “Add  
.gitignore” box.

**177** Click the .gitignore template: None dropdown menu, and type “Unity”.

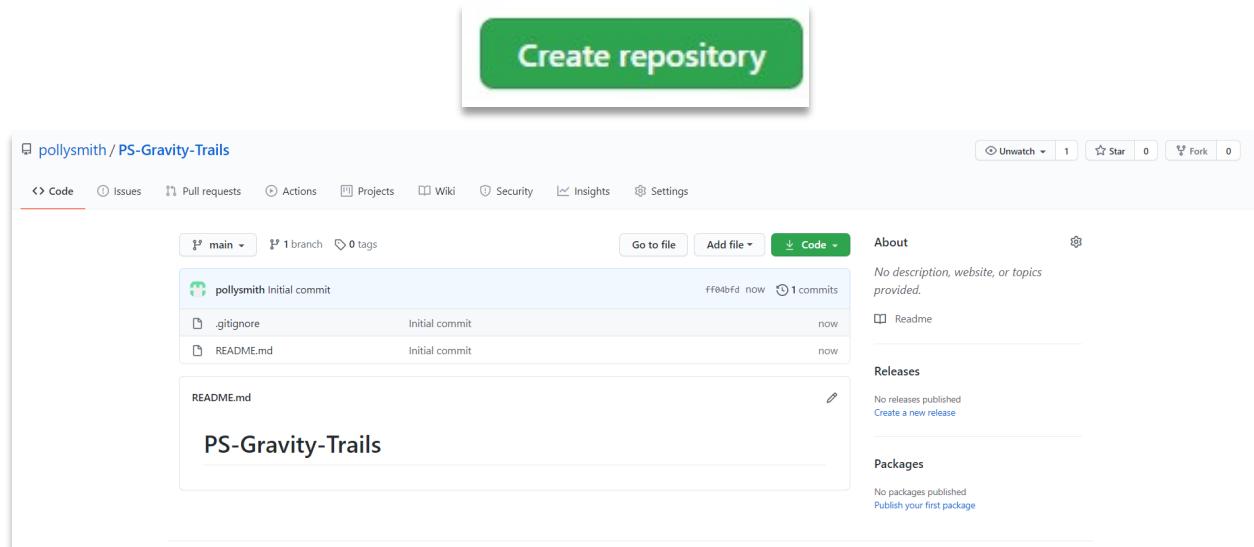


**178** Leave the “Add a license” box alone unless you know what kind of license you want to use.

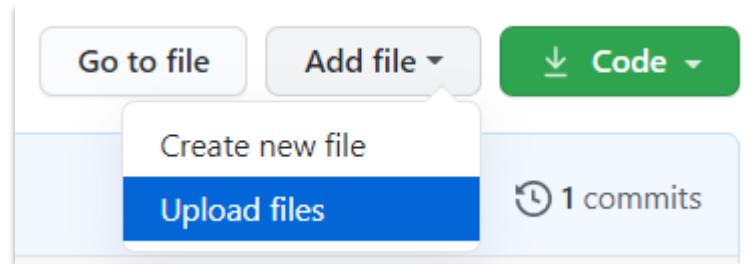
**Choose a license**  
A license tells others what they can and can't do with your code. [Learn more.](#)

This will set  `main` as the default branch. Change the default name in your [settings](#).

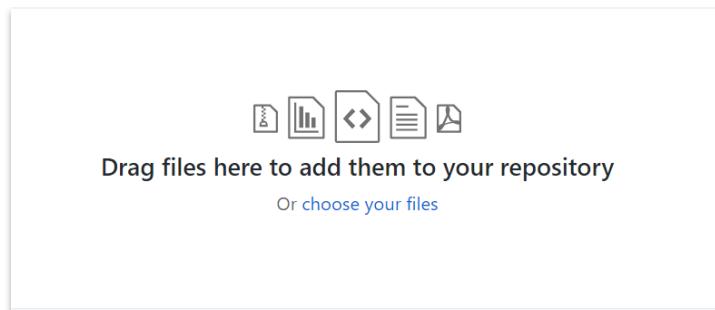
**179** Click the Create repository button.



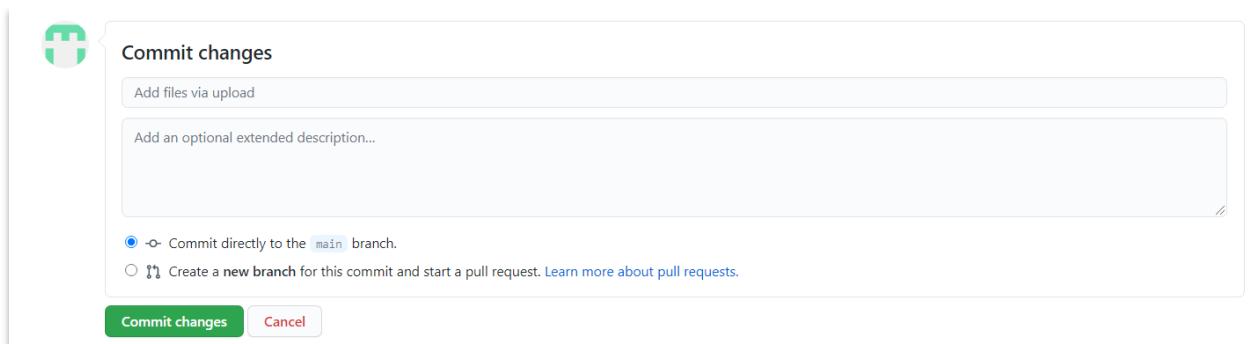
**180** Click Add file, then Upload files.



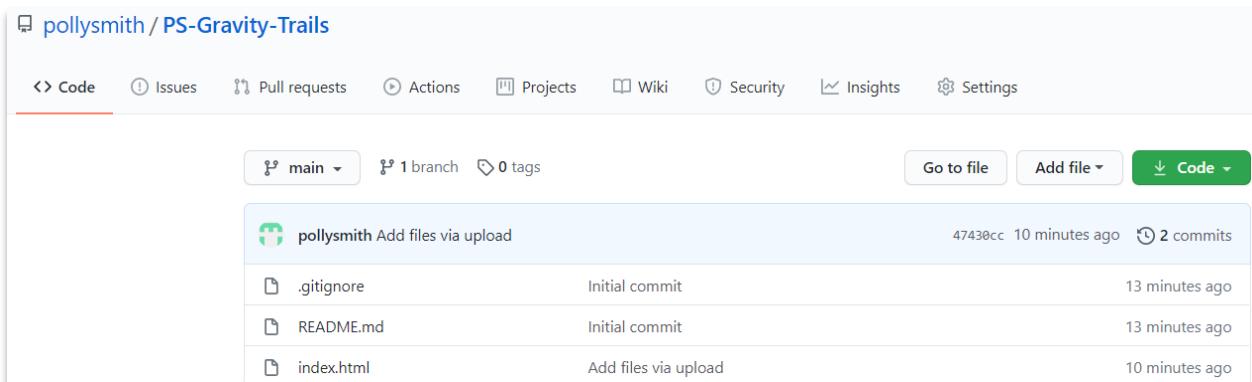
**181** Click “choose your files,” then locate the folder where you saved your game’s WebGL files. Upload *index.html*, *Build/*, and *TemplateData/*



**182** Click Commit changes.



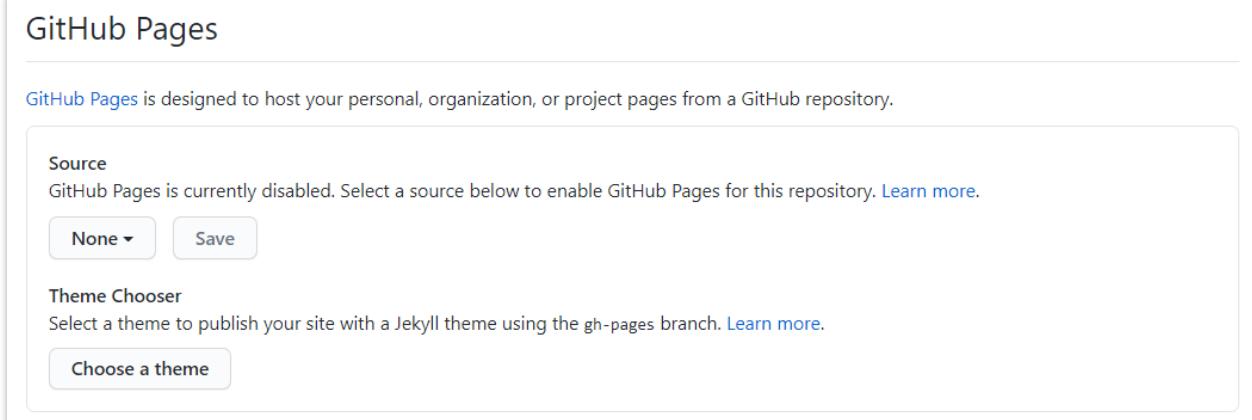
**183** Once your files are uploaded, click on “Settings”.



The screenshot shows a GitHub repository page for 'pollysmith/PS-Gravity-Trails'. The 'Code' tab is selected. At the top, it displays 'main' branch, 1 branch, 0 tags, and various navigation buttons like 'Go to file', 'Add file', and 'Code'. Below this is a table of commits:

Author	Commit Message	Date	Commits
pollysmith	Add files via upload	47430cc 10 minutes ago	2 commits
	.gitignore	Initial commit	13 minutes ago
	README.md	Initial commit	13 minutes ago
	index.html	Add files via upload	10 minutes ago

**184** Scroll down to “GitHub Pages”.



The screenshot shows the 'GitHub Pages' settings page. It includes sections for 'Source' and 'Theme Chooser'.

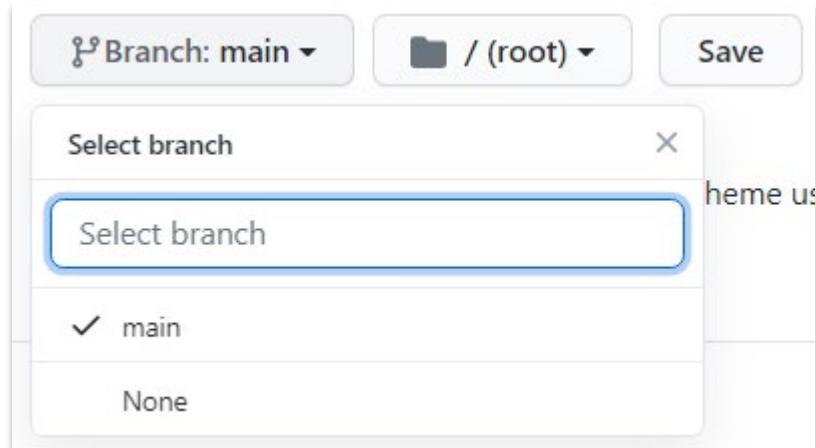
**Source**  
GitHub Pages is currently disabled. Select a source below to enable GitHub Pages for this repository. [Learn more](#).

Buttons: None ▾, Save

**Theme Chooser**  
Select a theme to publish your site with a Jekyll theme using the gh-pages branch. [Learn more](#).

Button: Choose a theme

**185** Click the dropdown menu under “Source” and select “Main,” then click “Save.”



**186** Wait while the page is being published, you may have to click refresh until the page is published.

A screenshot of the GitHub Pages publishing interface. The top navigation bar says 'GitHub Pages'. Below it, a green success message box contains the text 'Your site is published at <https://pollysmith.github.io/gravityfalls/>'. Below the message, there's a 'Source' section with a dropdown menu showing 'Branch: main' selected. To the right of the source section, a yellow callout box contains the text 'Use this link to share your project!'. A black arrow points from the bottom right of the 'Source' section towards the yellow callout box. At the bottom of the page, there's a 'Theme Chooser' section with a 'Choose a theme' button.

**187** Click on the hyperlink to open your game page. Test out your game to make sure all parts work and look the way you want.



**188** Your game is now published! You can share the github url to have friends and family play your game!

## Project Submission and Reflection

Now that you have a working game, take time to add your own personal touches to the project. What aspects from other games could you add?

Once you feel like you have a good product that represents your vision of the game, have a Sensei and at least one other Ninja playtest it. Use the Playtest Survey Planning Document for questions to ask them when they finish. Record their answers in your Ninja Planning Document.

Based on the results of the playtest and survey, make changes to your game. Once you are complete, share the updates with your Sensei and fill out the reflection section of your Ninja Planning Document.

Before you submit your game for grading, use the Gravity Trails Project Requirements Checklist to make sure your game has all of the required features.



### Ninja Planning Document

Use your Ninja Planning Document to record feedback from Senseis and other Ninjas in your Center.



## Codey Raceway

Your goal is to plan, program, and playtest a racing game. You will design your own course with obstacles and power-ups. You can also use the skills you learn in this challenge when you create future games that have obstacles and power-ups!



The game we will create together uses track pieces that allow you to design your very own unique track. The player's goal will be to reach the finish line as quickly as possible, but you are free to add additional tasks!

## Plan and Design

The first step is to plan out what your game will look like. Using your Ninja Planning Document, sketch out your track and the rest of the environment. Think of similar games that you have played to help you.



Mario Kart Tour by Nintendo  
Built in Unity



Fall Guys by Mediatonic  
Built in Unity

You should include a finish line, obstacles that prevent you from going through the track, and four item box spawning locations. Add details to your track, are there specific objects you want to use to make your track unique.



### Ninja Planning Document

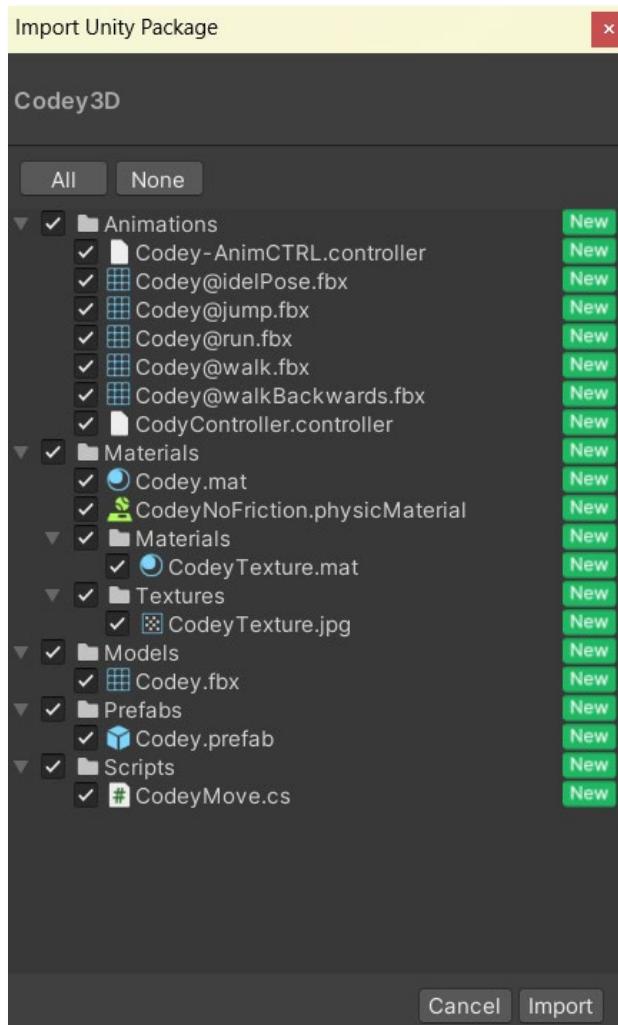
Take at least 5 but no more than 10 minutes and complete your Ninja Planning Document – Track Design

Take a look at the sample projects below for some inspiration!

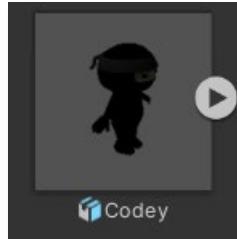


## Project Setup

- 1** Start a new Unity Project and name it *YOUR INITIALS – Codey Raceway*.  
Select **3D template**.
- 2** After it loads, rename the Sample Scene to Codey Raceway.
- 3** Import **Codey3D.unitypackage**. This is a Codey **game object** that can run around in 3D space.



- 
- 4** After Unity finishes importing, you will be able to find Codey in the **Prefabs** folder.



- 
- 5** Codey is missing a few important things that are needed to work properly.

Codey is missing...

- a unique **tag** for the Codey **game object**,
- a **component** that enables **collision** with other **game objects**,
- a **component** that enables **physics**, and
- a **script** named CodeyMove.

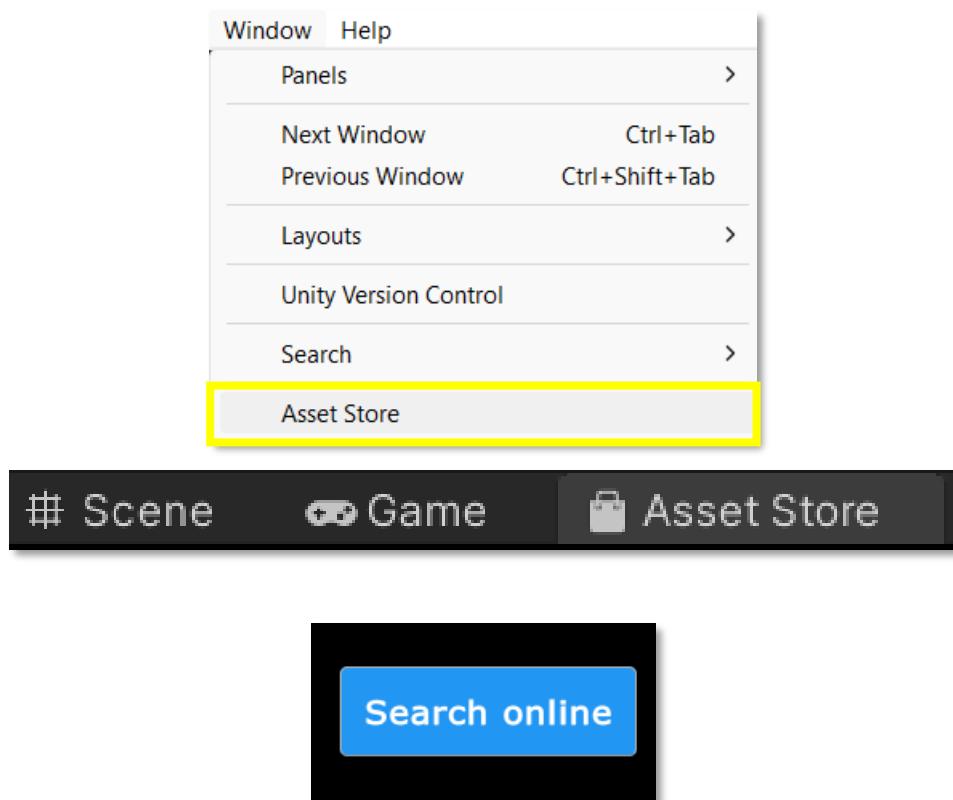
 **Sensei Stop**

Using the list above, add the required components to the Codey model. If you get stuck, work with your Sensei to add the components that you are missing.

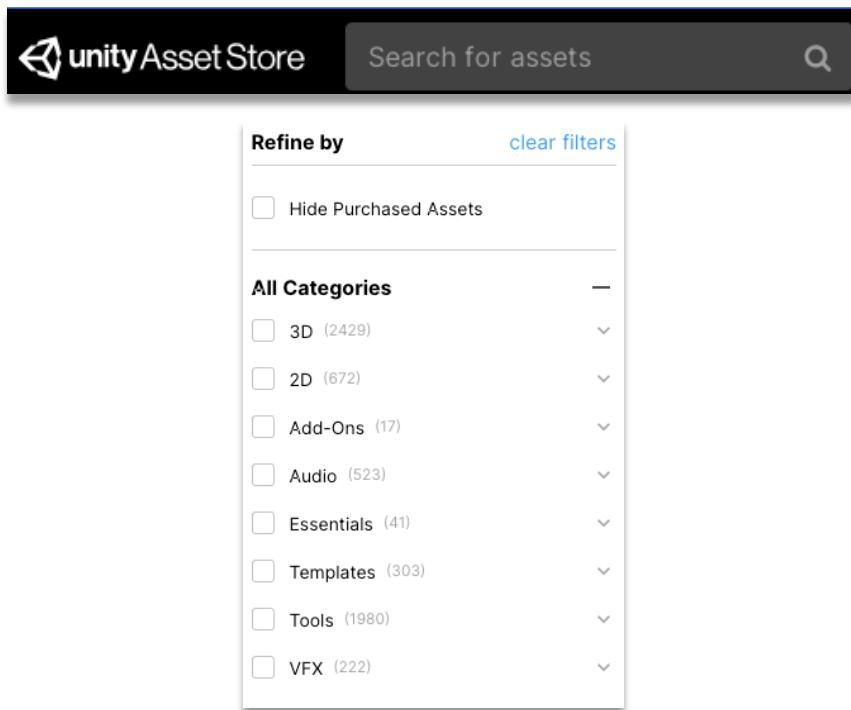
**6** Do you remember how you had a way to find multiple assets in the previous belts? In the early **White** through **Green** Belts you had the **GDP asset** library. Then, in **Blue** Belt, you had the **Toolbox** to help you find even more **assets** that other creators made.

Now, with Unity you have a great way to find assets that are free to use in the **Unity Asset Store**! You can find all kinds of amazing sprites, music, materials, and more to include in your game!

Open the **Unity Asset Store** by clicking on **Window** and then **Asset Store**. This will place an Asset Store tab in your Unity project so you can easily access it at any time. After you go to the asset store Tab, click **Search Online**.



- 7** Use can use the search box and the filters on the right side to narrow down what kind of **asset** you would like. You can use assets from the Unity Asset Store to customize your **games** and make it fit your theme!



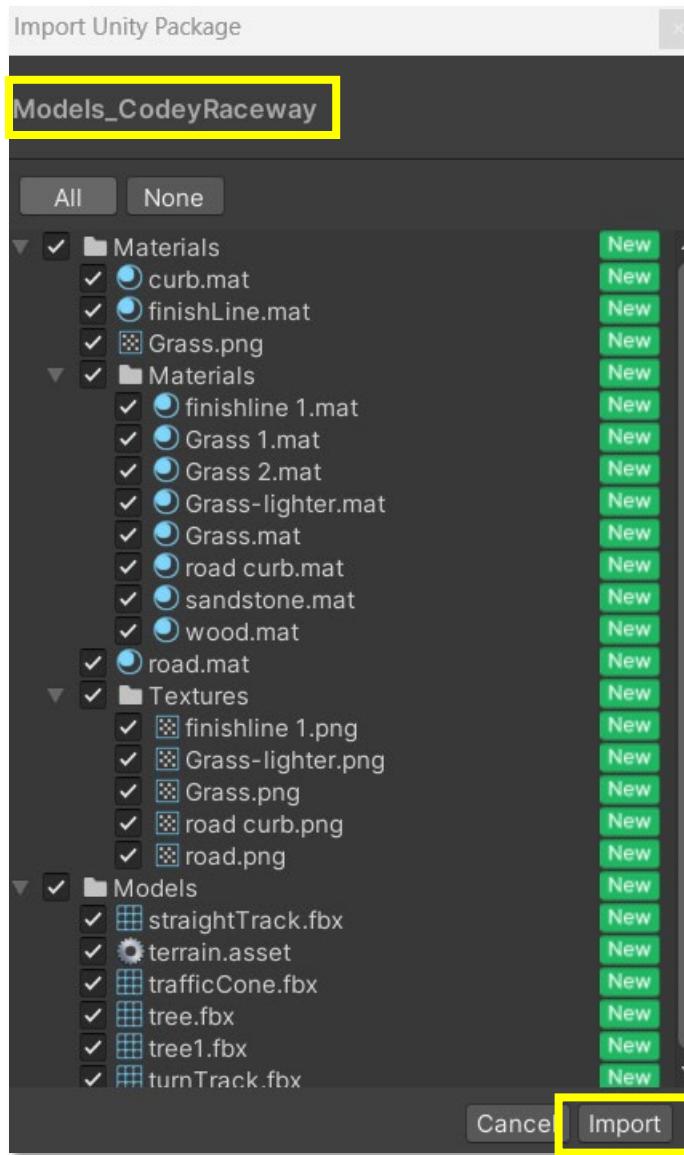
### \_IMPORTED\_ASSETS

Some assets are not free, meaning you need to buy them to use them. To only search for FREE assets, scroll down and on the right-hand side change the Pricing to Free Assets.

- 8** For this game, you have two options to customize your world. The next section will show you how to import **assets** from Code Ninjas to build your track. The following section will show you how to use the Unity Asset Store to find **assets** to use in your game.

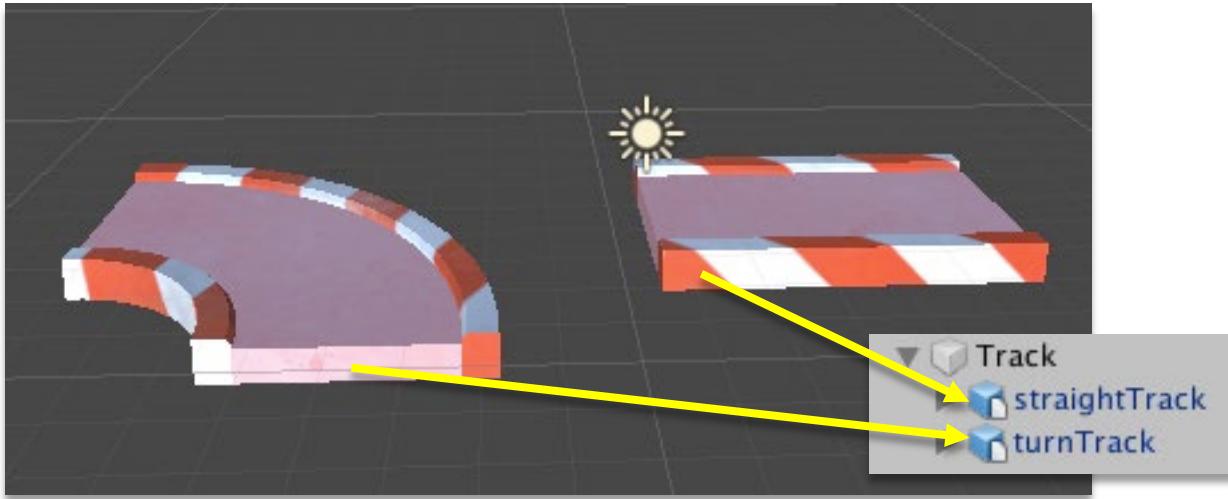
## On Track

9 Import the **Models\_CodeyRaceway.unitypackage**.



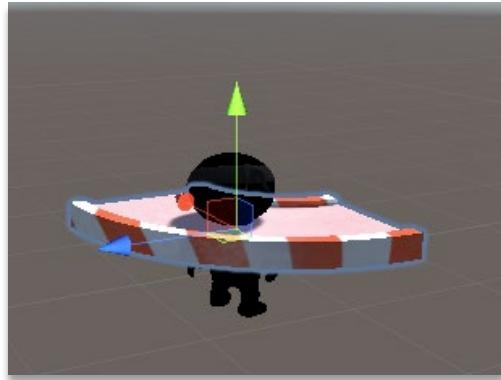
10 Create an empty game object and rename it **Track**. Use this game object to keep your track objects organized.

- 
- 11** From your **Models** folder, drag a **straightTrack** and a **turnTrack** on to the Track game object.



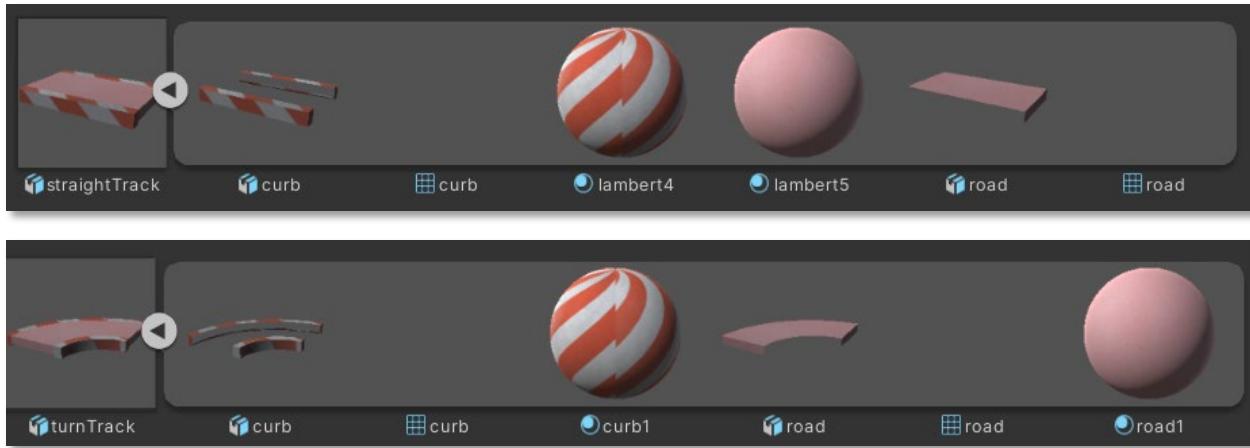
- 
- 12** Before moving forward, select either the **straightTrack** or the **turnTrack** piece in the **Hierarchy**. Notice how none of the track pieces have a collider. What does this mean? Let's find out!

Drag Codey from the **Models** folder on top of one of the track pieces and see what happens.



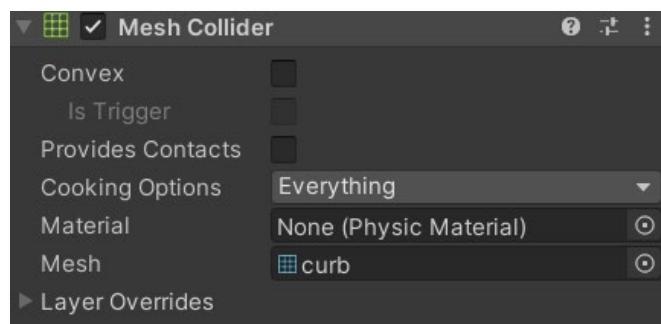
Codey goes straight through the pieces! We can fix this using a collider.

- 13** In the **Hierarchy** both the turnTrack and the straightTrack have two parts to them. Look in the **Models** folder and expand either the **straightTrack** or the **turnTrack**.



Both tracks have their own smaller parts that make up this one piece. Using the **Mesh Collider**, you will be able to form a collider for the **curb** and **road** parts.

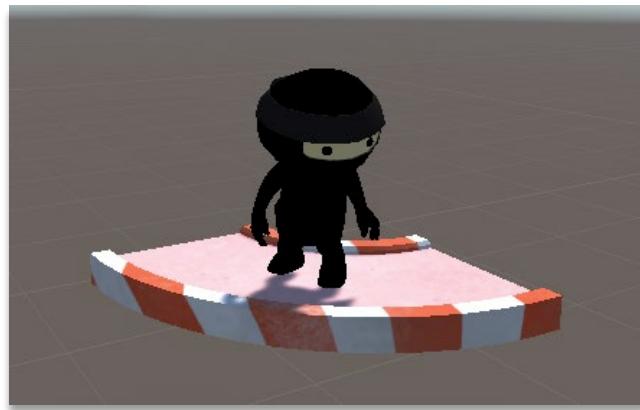
- 14** Select the **curb** game object and add a new **Mesh Collider** component. The **Mesh Collider** will be able to create a **collider** in the exact same shape of the curb piece.



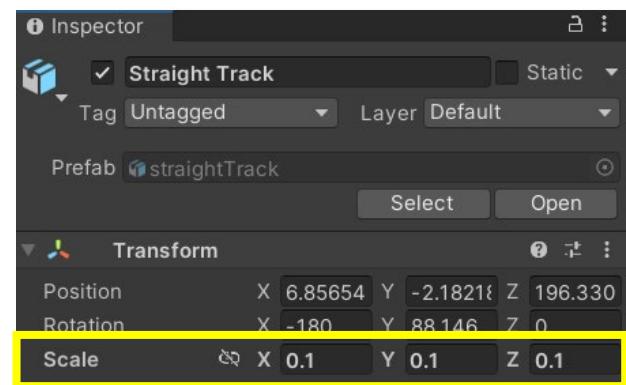
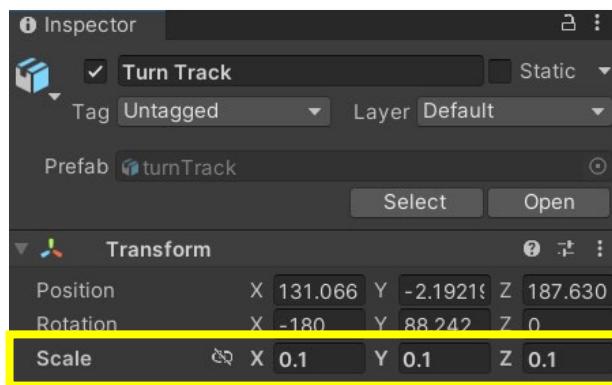
**15** Follow the same process and add the **Mesh Collider** to the **road** game object.



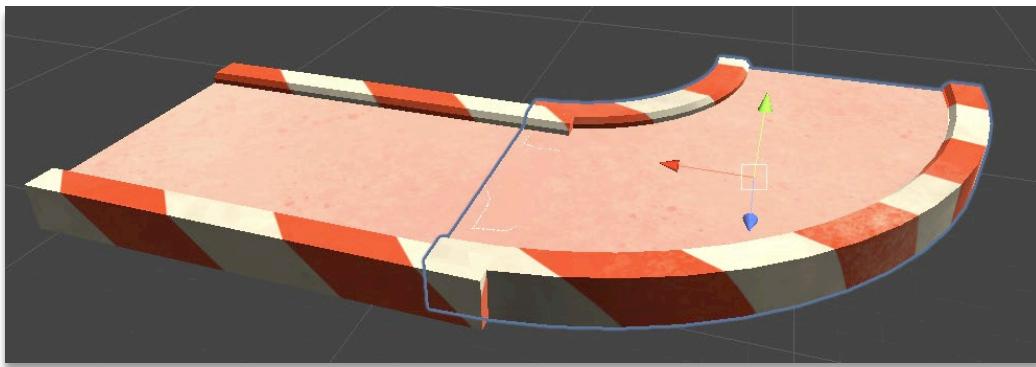
**16** Playtest your game. Codey no longer goes through the game objects!



**17** Stop your game. The track parts are kind of small, aren't they? Adjust the **scale** size of the pieces to your liking. Use the scale **sizes** below as an example.

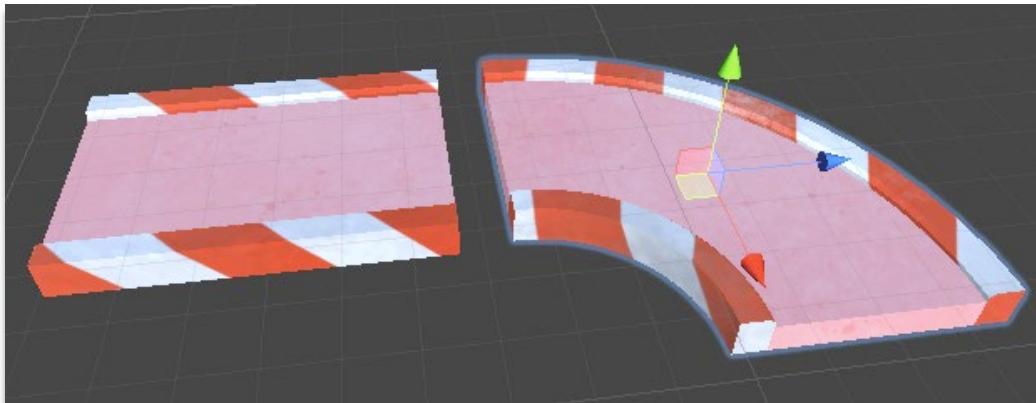


**18** The next step is to get the pieces to look like one complete part. If you just use the **Move Tool**, it is hard to tell if pieces are overlapping one another.



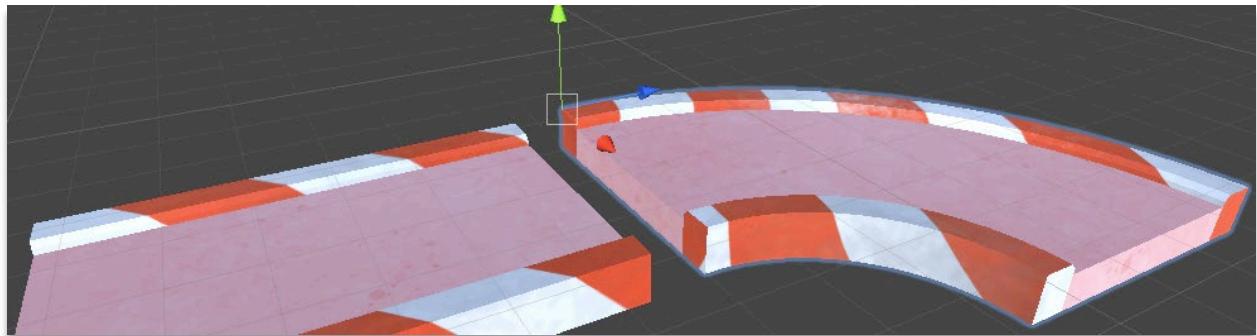
Unity has a great way to help us piece our tracks together – called the **Vertex Tool**. With the Vertex Tool, you can trigger a command in Unity using your keyboard and mouse.

First, select the piece you want to move.



**19** On your keyboard, hold the **v** key to activate the Vertex Tool. You can tell that the Vertex Tool is active if there is a small square following your mouse.

- 20** While holding the **v** key, move your mouse to a corner of the track piece. Click and hold in the square and drag it to the corner of the track piece you want to connect it to.

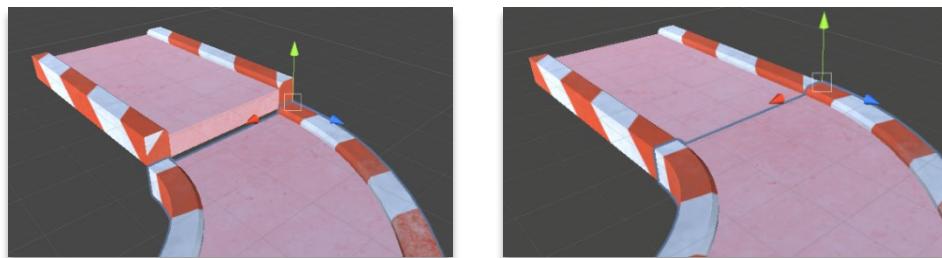


The two track pieces should now fit together without any overlapping parts!

 **Track Tips**

If you can't see where you are connecting the piece, hold the right mouse button and move the mouse around to have a better view of the corner pieces.

- 21** Look at the Y positions in the Inspector and make sure the track pieces have the same value.

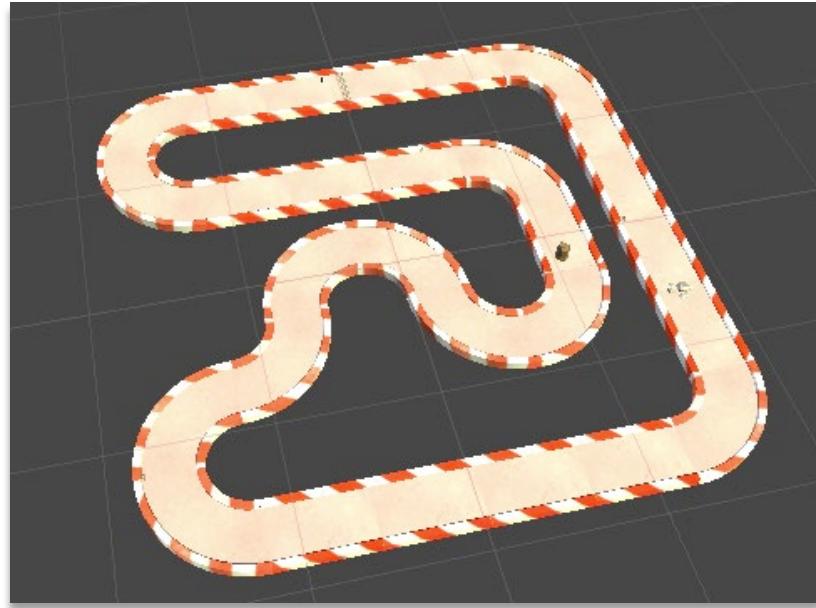


- 22** Duplicate the **turnTrack** and **straightTrack** in the **Hierarchy** for additional road pieces so you don't have to add the **Mesh Collider** component to track pieces every time!

 **Sensei Stop**

What Unity tools are you using to build your track? Are you changing the rotation, using your mouse, or making the scale negative? Explain what happens to the track depending on the tool you are using.

- 23** Have fun building your track! Take a look below to see the example track what we made.

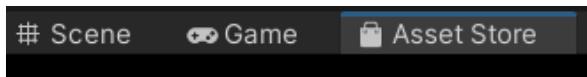


If you are using the Code Ninjas track pieces, you can skip the next section.

## Shopping Spree

---

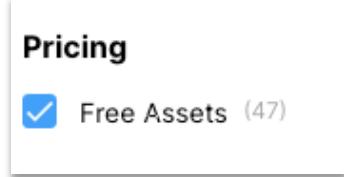
- 24** If you want to search and see what other tracks look like, open the **Unity Asset Store** by clicking on the tab or using the Window menu. And clicking on **Search Online**.



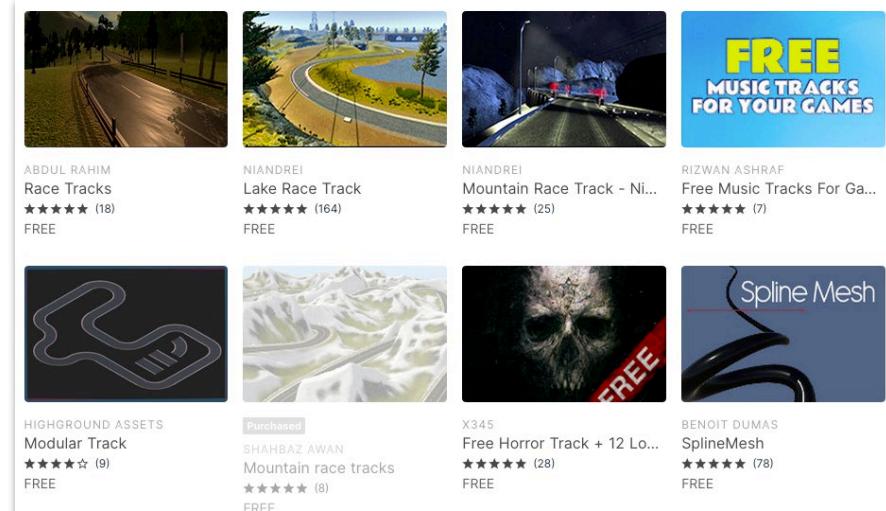
- 
- 25** Search for "tracks" and press **Enter**.



- 
- 26** On the right side, scroll down to find **Pricing** and check **Free Assets**.

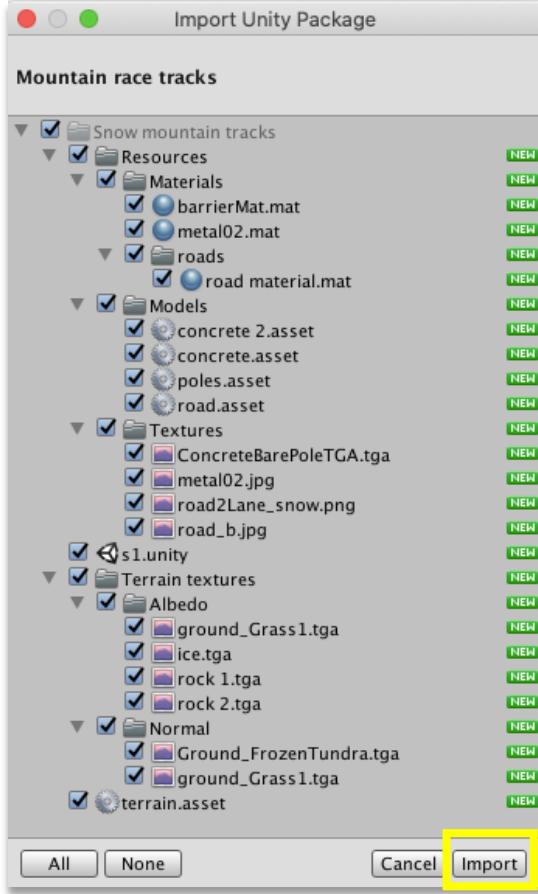


**27** You should see a few tracks you can download to your project! Look for one that you like and click on it.



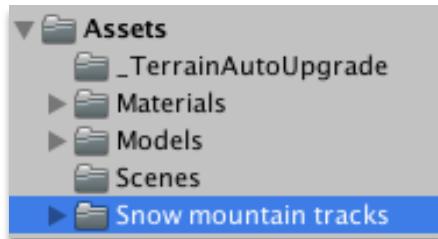
**28** After selecting the one you like, select **Download** then **Import**.

**29** An import pop-up will appear. Select **All** then **Import**.



It might take some time to add the entire package.

**30** Every time you import a package from the store it will be placed under your **Project** tab inside your **Assets** folder. In this example it is called "Snow mountain tracks".



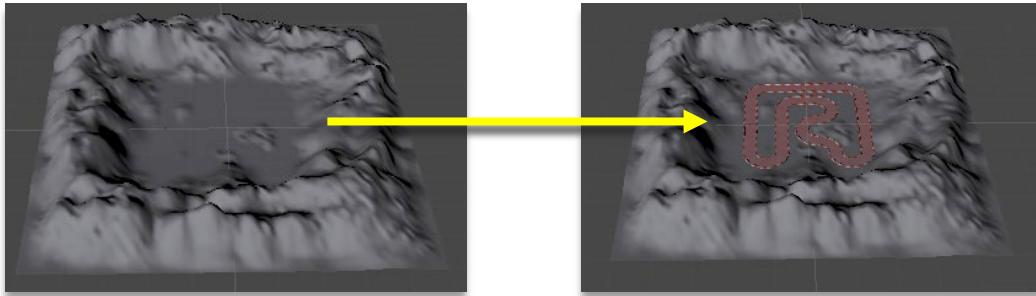
Each package is different. Go through the folder and use the assets you like the most for your track.

## Moving Mountains

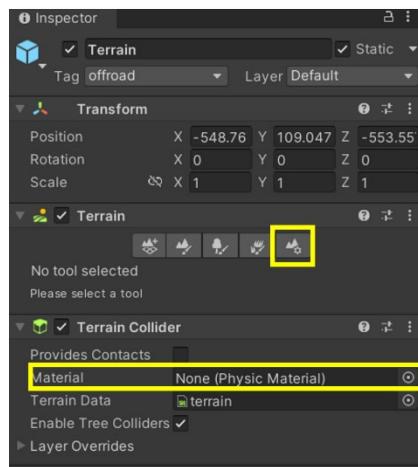
**31** In the Models folder you can also find a **terrain**. Drag this either into the **scene** or into your **Hierarchy** to see what it looks like.



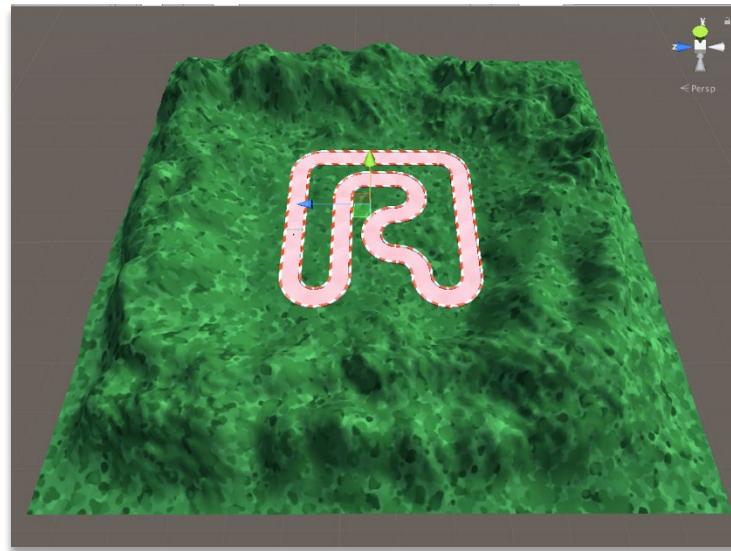
**32** The **terrain** can be used to give your track an extra layer of creativity. Adjust the placement of it so the track is above the terrain.



**33** You can also add a **material** to the terrain to adjust how the game looks. To do this, select the **Terrain**. Under the **Terrain component**, select the settings icon. You can add different types of terrain depending on the material you use.

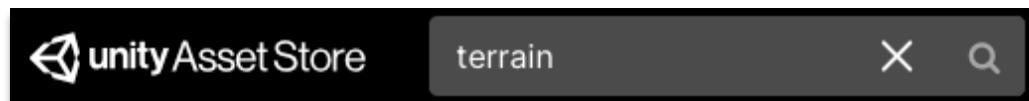


**34** You can create a new **material**, use an existing **material**, or find a **material** in the **Unity Asset Store**. In the example below, we used the grass **material**!



**35** Now that you see what the terrain can do, decide if you want to use the one provided, find a new **terrain** in the **Unity Asset Store**, or not use it at all.

Follow the same steps you used to find tracks to find a new terrain.



Some of the packages may have more than what you are looking for.  
For example, the track we imported already came with a terrain.

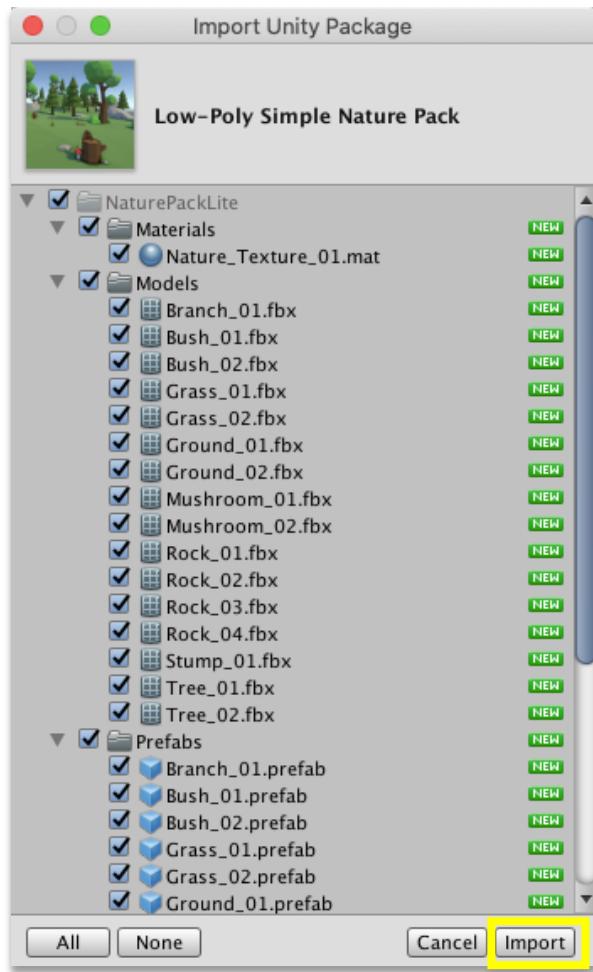
## Exterior Decorating

**36** Let's make your scene even more unique! Use the **Unity Asset Store** to add more to your scene. Start by searching for **nature pack** and selecting **Free Assets** in the **Pricing** category.

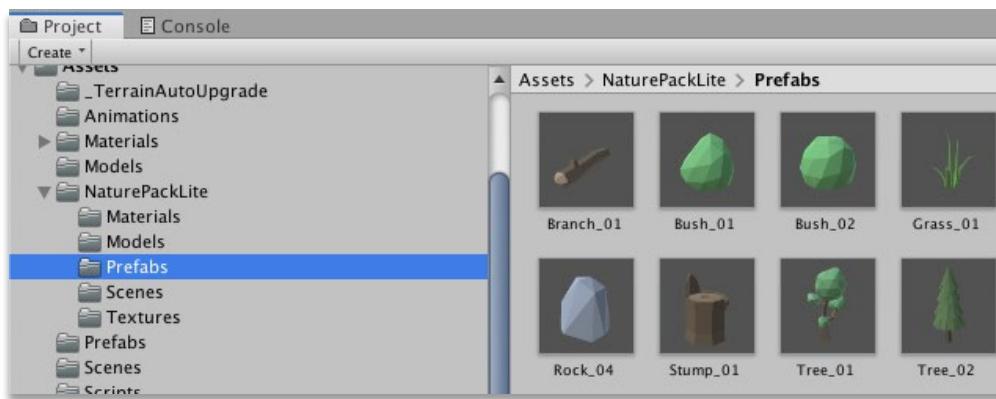
The screenshot shows the Unity Asset Store interface. At the top, there is a search bar with the text "nature pack" and a magnifying glass icon. Below the search bar, several asset packages are listed in a grid format:

Asset Name	Creator	Description	Rating	Status
Terrain Tools Sample Asse...	UNITY TECHNOLOGIES	Low-Poly Simple Nature P...	★★★★★ (91)	FREE
Low-Poly Simple Nature P...	JUSTCREATE	Simple Low Poly Nature P...	★★★★★ (24)	FREE
Lowpoly Nature & Village Pack	NEUTRONCAT	RPG Poly Pack - Lite	★★★★★ (4)	FREE
Grass And Flowers Pac...	VLADISLAV POCHEZHERTSEV	Lowpoly Nature & Village ...	★★★★★ (70)	FREE
Low Poly Rock Pack	HASAN3DMODELS	(not enough ratings)	★★★★★ (19)	FREE
Parks and Nature Pack - L...	BROKEN VECTOR		★★★★★ (6)	FREE

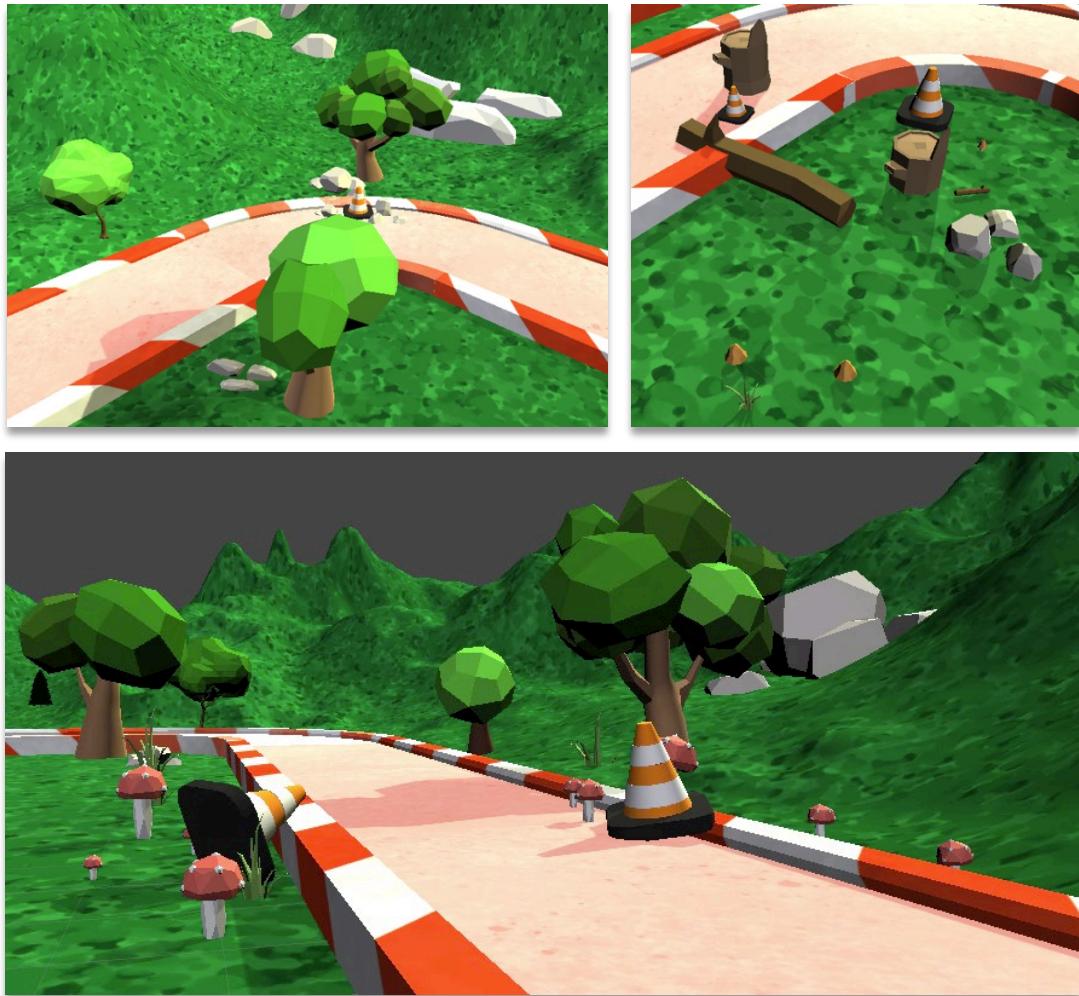
**37** When you find one you like, make sure to **import** the complete **package**.



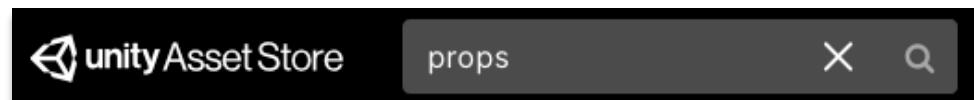
**38** You will find a new folder in the **Project** tab. In this folder there will be **prefabs** that you can add into your **scene**.



**39** To spark your creativity, take a look at what we added to make the world more unique!

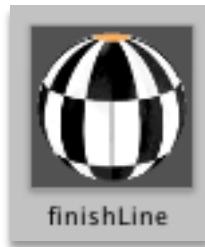


**40** You can find other objects to use in your scene by searching for **props**, **race**, **environment**, and **obstacles**.

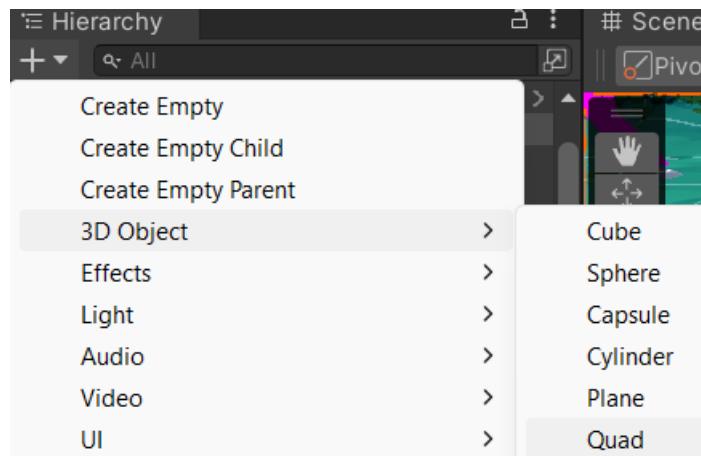


## The End of the Road

- 41** We can't have a race without a finish line! There is a **finish line material** in the **Materials** folder you can use.

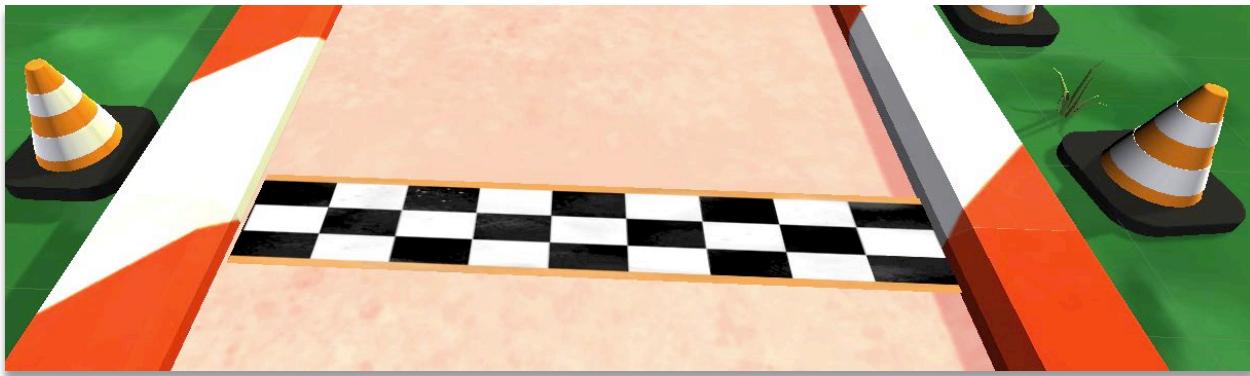


- 42** This material looks the best on a **Quad game object**. Add a new **Quad game object** to your **scene**.



**43** The Quad will get placed somewhere random in your scene and it will be very small.

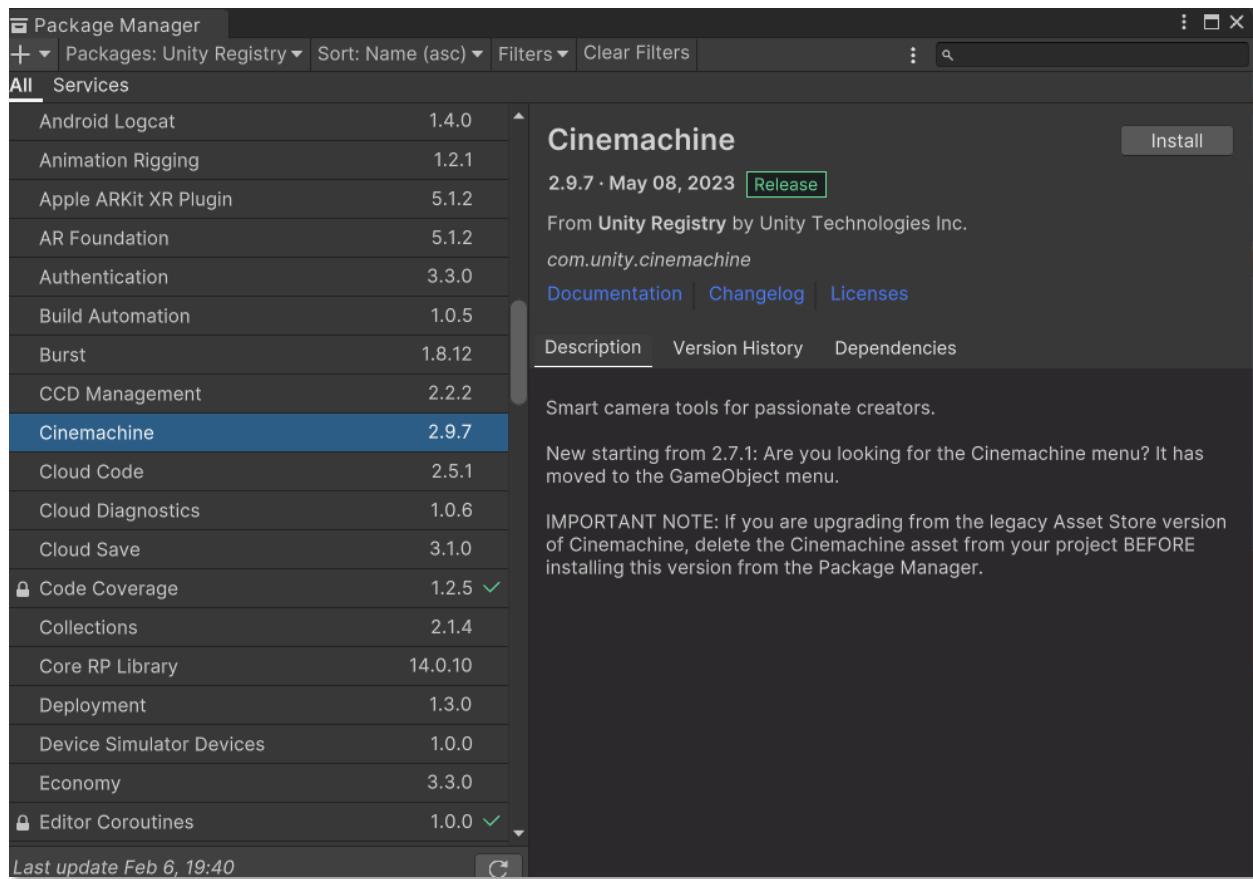
Change the **position**, **size**, and **rotation** of the Quad so it is somewhere on your track. Drag the finishLine **material** onto the Quad. Rename the **game object** to Finish Line.



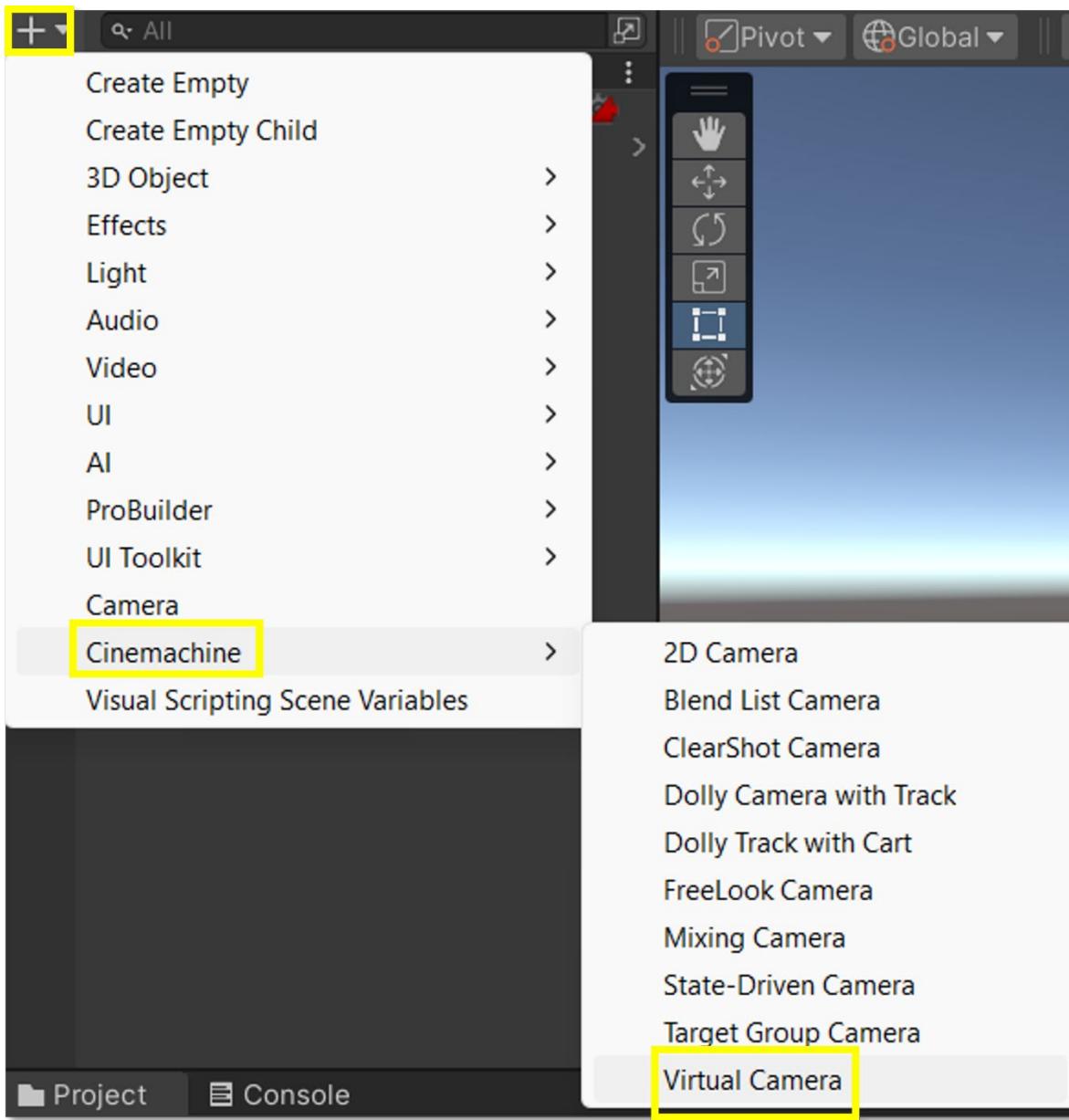
Notice how the finish line looks like part of the road. That's how you'll want yours to look when you're done.

## Eyes on the Road

**44** Generally, in a game we want the camera to follow the main character. This section will teach you how to make the camera follow behind Codey. To do this, we will use a **Cinemachine**. In the tabs, click on **Window** followed by **Package Manager**. Make sure to click on **Packages** under the **Package Manager** tab, then select **Unity Registry**.

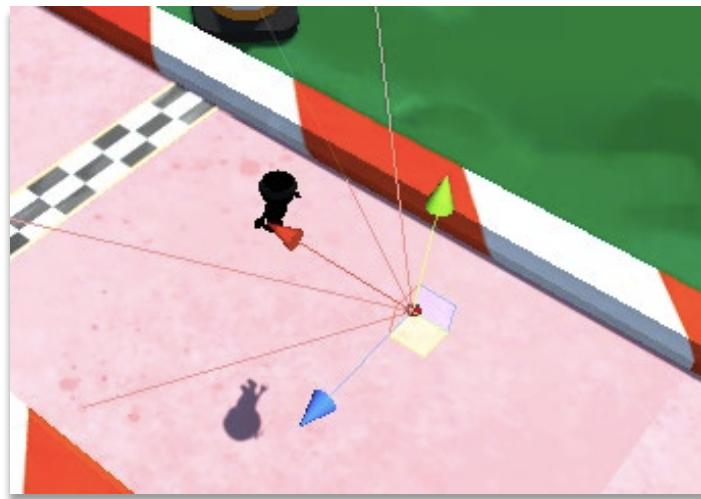
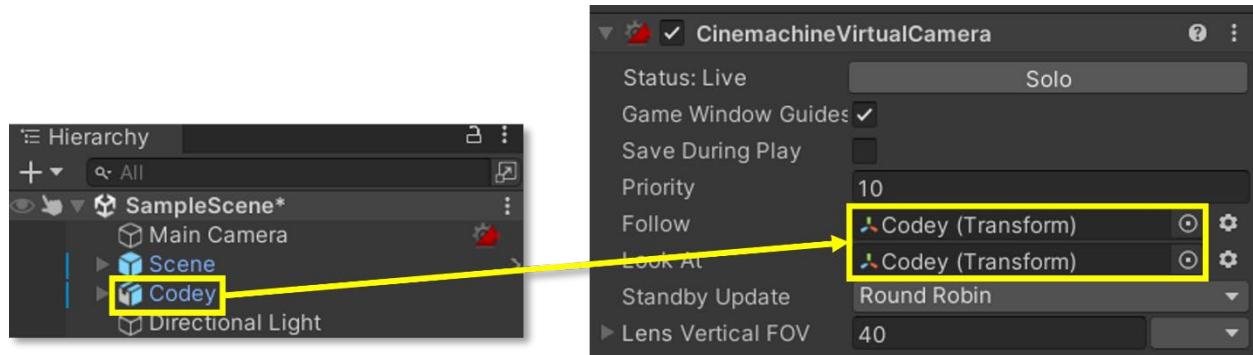


**45** You should now see a **Virtual Camera** in your object **Hierarchy**. If not, click **+** then **Cinemachine**, then **Virtual Camera**.



 **Virtual Camera**

**46** Select **Virtual Camera**. Then, from the **Inspector**, drag the **Codey** from the Hierarchy into the **Follow** and **Look At** properties in the Cinemachine Virtual Camera component.



Make sure you have the **CM vcam1 game object** selected. Notice that the camera is always behind the player?

---

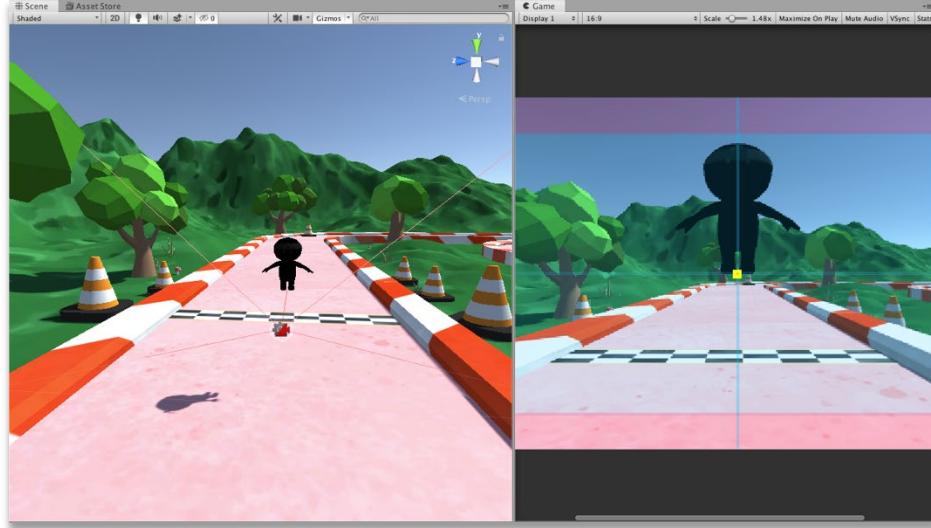
**47** Play your game and look at the current view of the camera.



It is way too low! Let's fix that. Stop your game.

---

**48** Move your **Game Tab** to the right of your **Scene Tab** so you can see both at the same time.

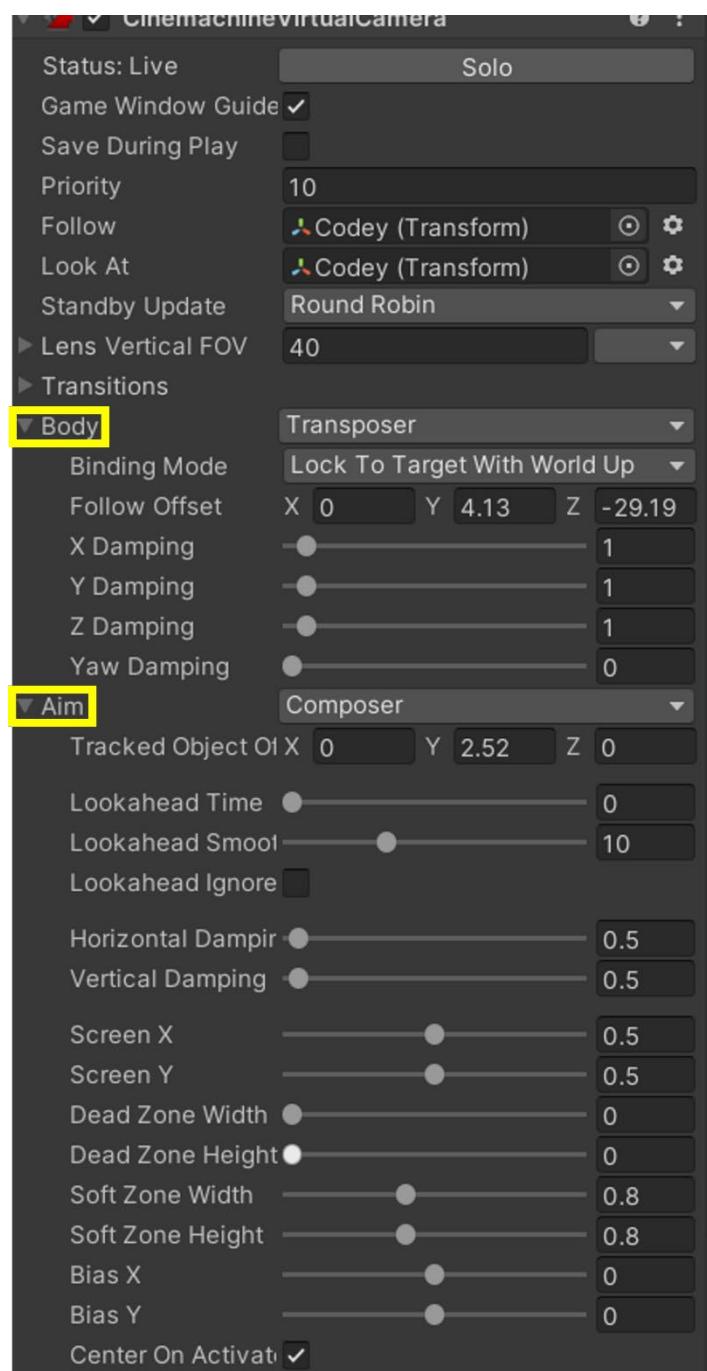


**49** While the **CM vcam1** object is still selected in the **Hierarchy**, you should notice a small yellow square in the **Game Tab**. This square tells our camera where to look.

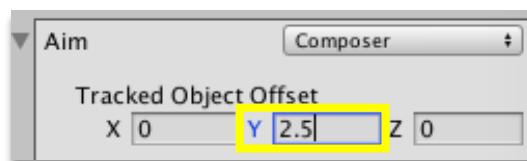


See how the yellow square is at Codey's feet? This is what makes the camera's view so low. Since we don't want the camera to always look at Codey's feet, we will adjust it to look between Codey's body and head.

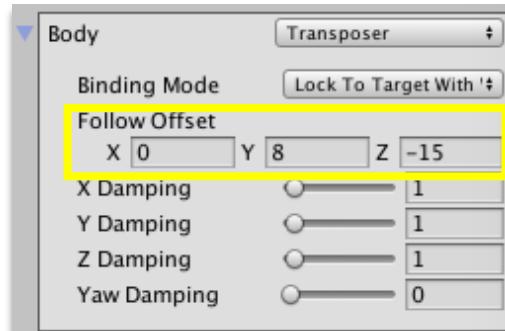
**50** You will be able to adjust the camera view in the **Inspector**. The two components you want to change are the **Body** and **Aim**.



**51** Adjust the **Aim** first. This allows you to move the yellow square between Codey's body and head. Since the square is already in the center, we do not want to move it in the **X** or **Z** directions.

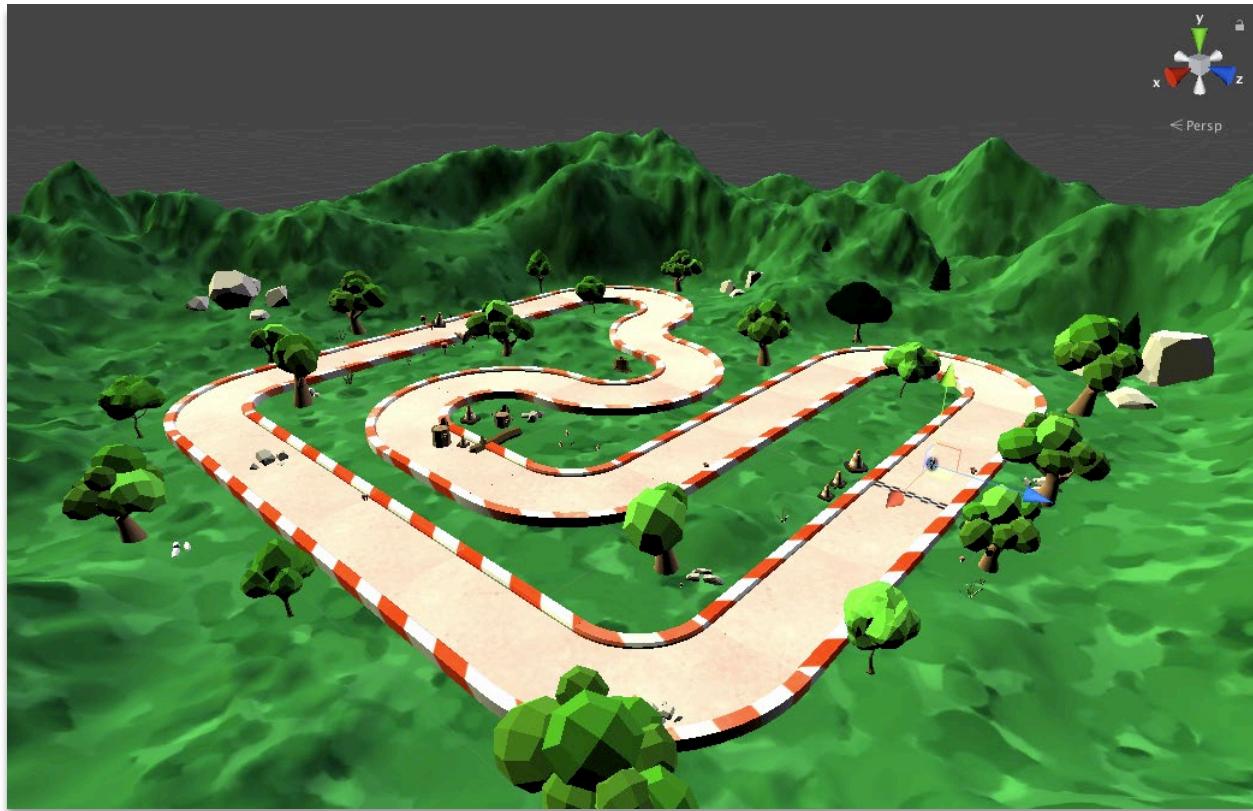


**52** The **Body** component allows you to zoom in or zoom out of the view. Adjust the three Follow Offset values.



## Lap it Up

**53** At this point, you should have Codey and your **scene** set up! Take some time to play your game, making sure all the game objects in the scene are placed where you want them. Feel free to go back into your scene and make any changes.

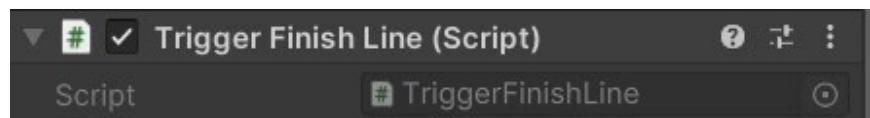


In all racing games, you must go through the finish line to win the race. Let's program the logic into the game to make the player win once Codey crosses the finish line.

**54** Click on the **Scripts** folder and create a new script named **TriggerFinishLine**.



**55** Attach this script to your **FinishLine** game object and open it in Visual Studio.



**56** In this script, we will check for when Codey triggers the finish line.

Delete the Start and Update **functions**. Add an **OnTriggerEnter** **function**.

```
public class TriggerFinishLine : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
    }
}
```

A screenshot of a code editor showing a C# script named "TriggerFinishLine". The script contains a public class definition for "TriggerFinishLine" that inherits from "MonoBehaviour". Inside the class, there is a private void function named "OnTriggerEnter" that takes a "Collider" parameter. The entire "OnTriggerEnter" function block is highlighted with a thick yellow rectangular border.

**57** Inside the OnTriggerEnter **function**, write an **if statement** that checks the tag of the other game object and compares it to the tag you gave Codey.

In this example, Codey's tag is "Player".

```
public class TriggerFinishLine : MonoBehaviour
{
    -references
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
        }
    }
}
```

**58** To signal when Codey triggers the finish line object, let's print "**You Win!**" to the console.

```
public class TriggerFinishLine : MonoBehaviour
{
    -references
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            print("You Win!");
        }
    }
}
```

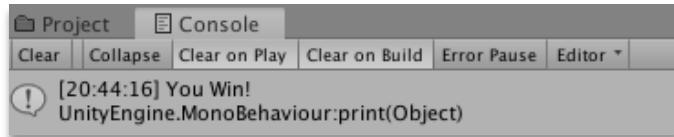
**59** Save your **script** and return to Unity. Playtest your game and try to trigger the finish line. Check to see if the “You Win!” message shows up in your console.

What happened? Why won’t the message show? What’s missing from the finish line **game object**?

 **Sensei Stop**

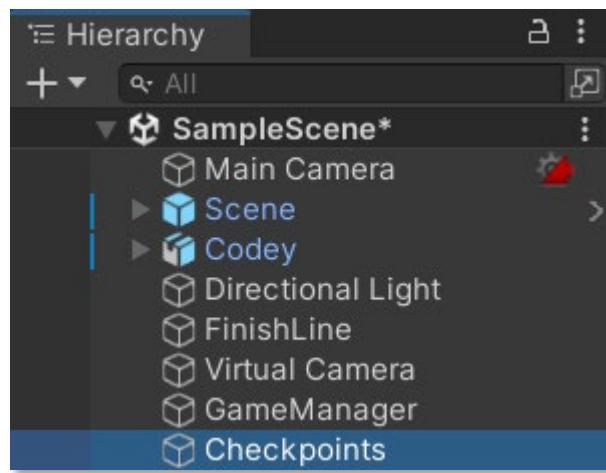
Discuss with your Code Sensei and share why you think the message inside the OnTriggerEnter function did not appear. Change the finish line game object to make the message appear.

**60** After discussing with your Sensei and making adjustments, check again for the message.



**61** In most racing games, to avoid cheating, the game creators add checkpoints throughout the track that the player does not notice. The checkpoints we create can be very small or very big. The player does not need to know where they are placed as long as they go through all of them.

Create an empty game object and rename it **Checkpoints**. We will use this object to organize the checkpoints. You can use checkpoints in any game to make sure your player goes through a specific path while playing!

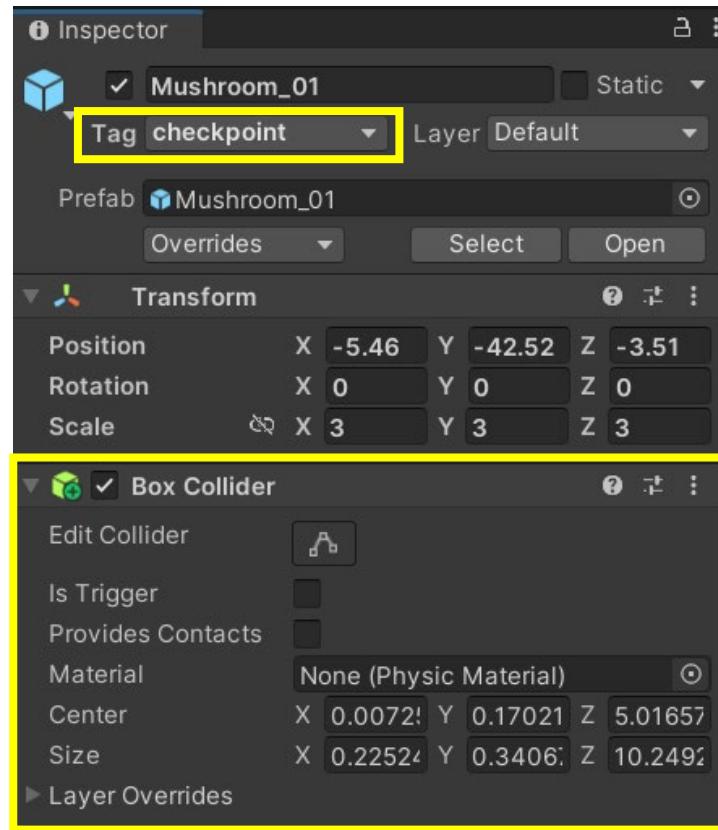


**62** Our example will use the **Mushroom\_01** prefab. You can find this object in the **NaturePackLite's Prefabs** folder.



Though we are using the Mushroom prefab, feel free to use any game object you want. The steps you follow will be the exact same! Drag your chosen object from the prefabs folder into the Checkpoints **game object**.

**63** Give your game object the **checkpoint** tag and attach a **Box Collider**.



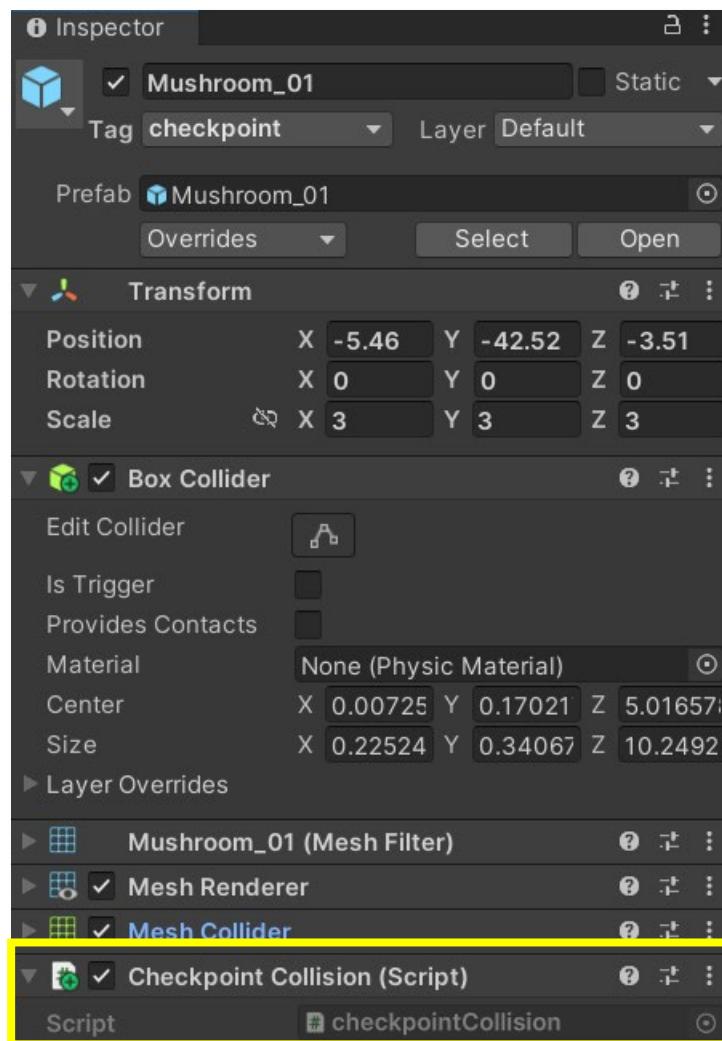
**64** Reposition your object near the finish line of your track. This will be our first checkpoint.



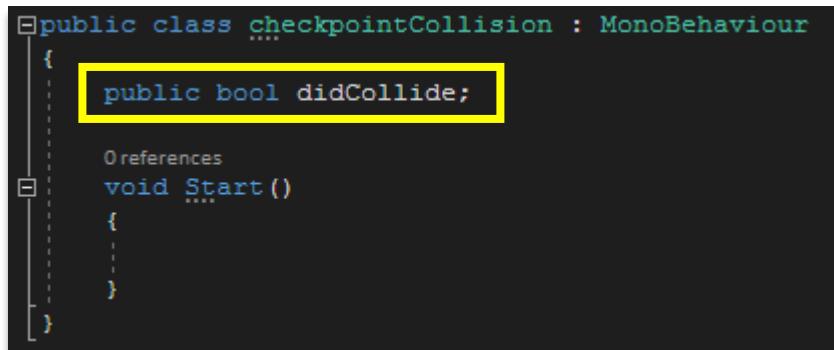
**65** Create a new script in the script folder and name it **checkpointCollision**.



**66** Attach this script to the object you are using as the first checkpoint. Remember, we used the **Mushroom\_01** object in our example, but you could pick any object you want.



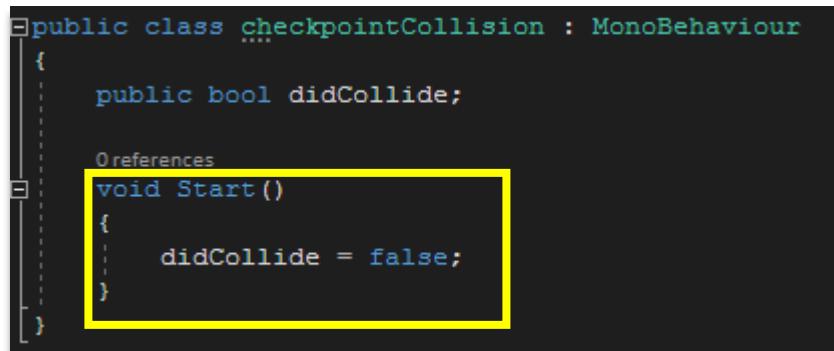
**67** Open the **checkpointCollision** script. Delete the Update function. Create a **public bool** variable and name it **didCollide**.



```
public class checkpointCollision : MonoBehaviour
{
    public bool didCollide;

    void Start()
    {
    }
}
```

**68** In the Start **function**, set the value of **didCollide** to **false**. This makes sure that Unity knows that we have not collided with the checkpoints.

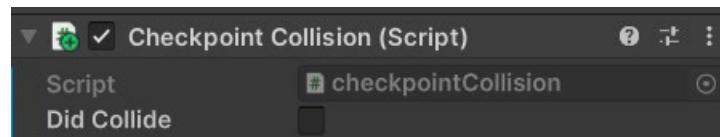


```
public class checkpointCollision : MonoBehaviour
{
    public bool didCollide;

    void Start()
    {
        didCollide = false;
    }
}
```

**69** Save your **script** and return to Unity.

By making this **bool variable** public, we can see in the **Inspector** if the checkbox gets checked when Codey collides with the checkpoint. A **private bool** would not allow you to view this change in Unity!



**70** Create a **OnTriggerEnter** function so we can check to see when Codey collides with the checkpoint.

```
public class checkpointCollision : MonoBehaviour
{
    public bool didCollide;

    void Start()
    {
        didCollide = false;
    }

    private void OnTriggerEnter(Collider other)
    {
    }
}
```

**71** In our **OnTriggerEnter** function, we must check if Codey has collided with the game object and if the **didCollide** variable is **false**.

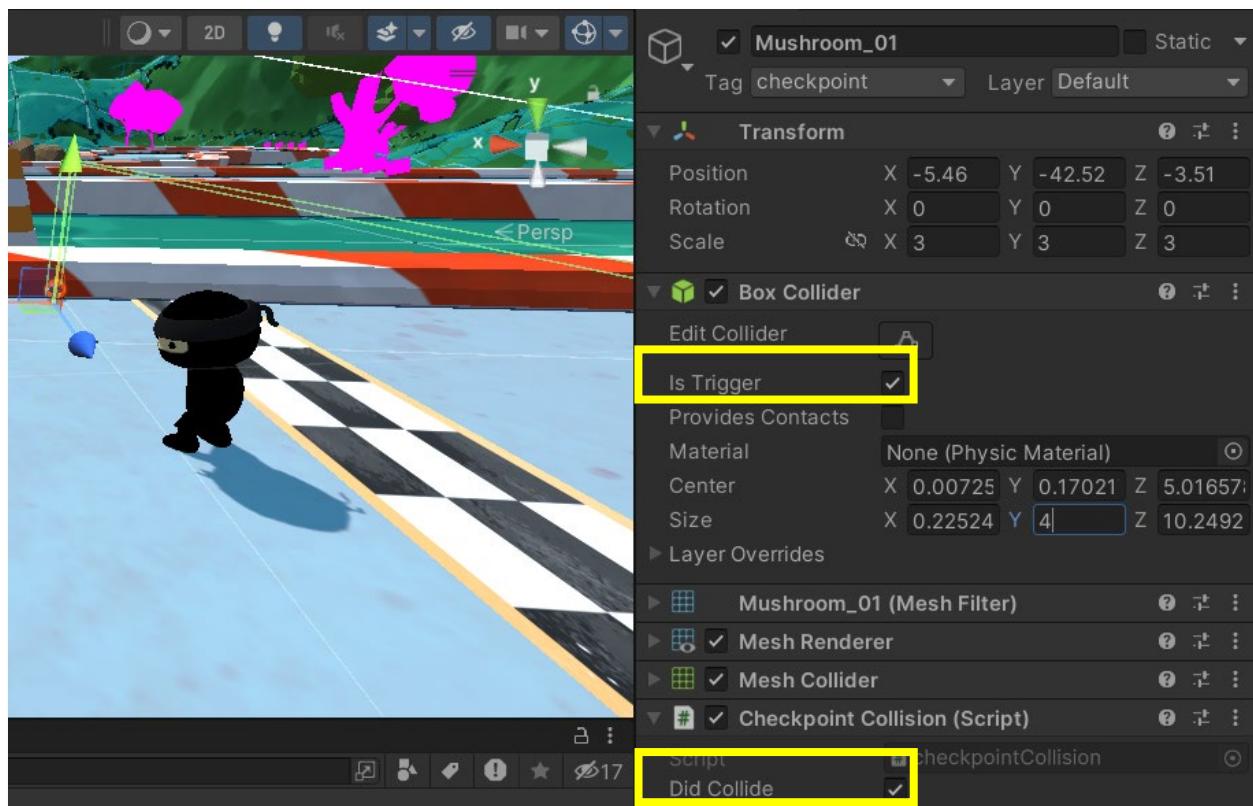
When both of our conditions are true, we want to set **didCollide** to **true**.

```
public class checkpointCollision : MonoBehaviour
{
    public bool didCollide;

    void Start()
    {
        didCollide = false;
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player" && didCollide == false)
        {
            didCollide = true;
        }
    }
}
```

**72** Playtest your game. Adjust the size of the checkpoint's box collider so it covers the width of the track. What happens when the Mushroom collider's Is Trigger property is disabled? What about when it is enabled? How does this affect Codey and the **didCollide** variable?

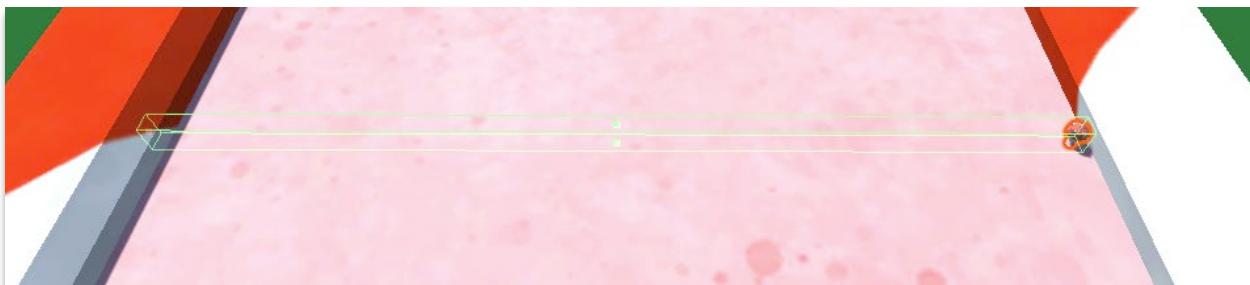


### Sensei Stop

What makes a trigger collider different from a regular collider?  
Show your Code Sensei the difference when you collide with a  
trigger collider and a regular box collider.

**73** If we only have one checkpoint, players can still cheat. Add at least **3 more checkpoint triggers** throughout your track. To make this easier, duplicate the game object you are using as the checkpoint in the **Hierarchy**.

Based on where you place your checkpoints, the object might need to be rotated to stretch across the track.



**74** Now that we have our checkpoints set up, we need to make sure our player goes through all of them. Let's catch those cheaters!

In the Hierarchy, create an empty game object named **GameManager**. We have used GameManagers before to keep track of different events going on in our games. In the Purple Belt we called it a GameController, but they do the exact same thing.



This **GameManager object** will keep track of the number of checkpoints Codey has gone through.

---

**75** Create a **CheckpointCounter** script and attach it to the **GameManager** object in the **Inspector**.



---

**76** Open the script. You can delete the **Update** function. Create a **public int variable** named **numberOfCheckpoints**. This **public int variable** will check how many total checkpoint **objects** there are at the start of the game. Remember that **int** is short for integer, another name for a whole number.

```
public class CheckpointCounter : MonoBehaviour
{
    public int numberOfCheckpoints;
    void Start()
    {
    }
}
```

---

**77** Create a second **public int** variable called **triggeredCheckpoints**. This variable will keep count of the number of checkpoints Codey has gone through.

```
public class CheckpointCounter : MonoBehaviour
{
    public int numberOfCheckpoints;
    public int triggeredCheckpoints;
    void Start()
    {
    }
}
```

## 78

At the very start of the game, we want to find every game object that is a checkpoint. How do we do that?

We need to count how many **objects** have a **tag** value of “**checkpoint**” at the start of the game. In the script’s Start function, use the **numberOfCheckpoints** variable and the **FindGameObjectsWithTag** function to find all the objects with the tag **checkpoint**.

```
public class CheckpointCounter : MonoBehaviour
{
    public int numberOfCheckpoints;
    public int triggeredCheckpoints;
    void Start()
    {
        numberOfCheckpoints = GameObject.FindGameObjectsWithTag("checkpoint");
    }
}
```

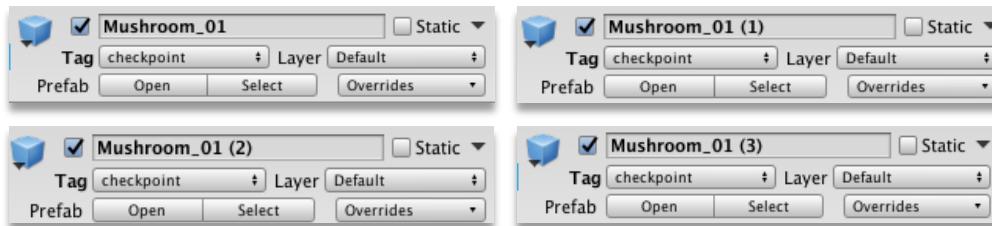
### Sensei Stop

Tell your Code Sensei how the **FindGameObjectsWithTag** function will help us find the total number of checkpoints in our scene. What must we do to all our objects need to be considered a checkpoint? If necessary, make the changes to the object you are using as checkpoints. Why do you think there is an error?

**79** The **FindGameObjectsWithTag** function returns an array of all the objects with the tag we are searching for.

Based on our scene, the function returns an **array** of the four Mushroom objects because they all have the "checkpoint" tag.

[Mushroom\_01, Mushroom\_01 (1), Mushroom\_01 (2), Mushroom\_01 (3)]



Using this **array**, we can get the total number of objects with the tag using the key word **Length**. Any time you want to count the number of instances of a certain tagged object, you can use **.Length** to get the number.

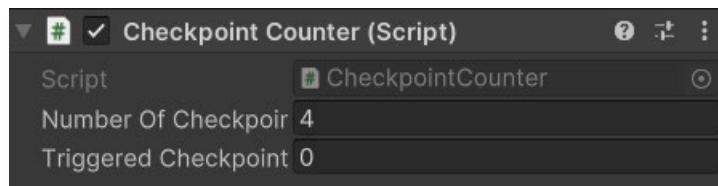
```
public class CheckpointCounter : MonoBehaviour
{
    public int numberOfCheckpoints;
    public int triggeredCheckpoints;
    void Start()
    {
        numberOfCheckpoints = GameObject.FindGameObjectsWithTag("checkpoint").Length;
    }
}
```

Because the **type** of the **variable** on the left hand side and the **value** on the right hand side are both **integers**, the **error** goes away!

**80** Since we made **numberOfCheckpoints** a public variable, we can see in the **Inspector** how many game objects with the tag **checkpoint** are in the game.

**Playtest your game** and select your **GameManager** object.

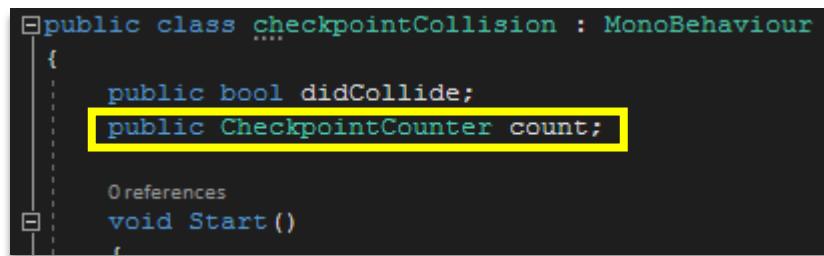
In your **CheckpointCounter** script, you should see the **Number of Checkpoints** variable change to the number you created.



**81** Stop your game.

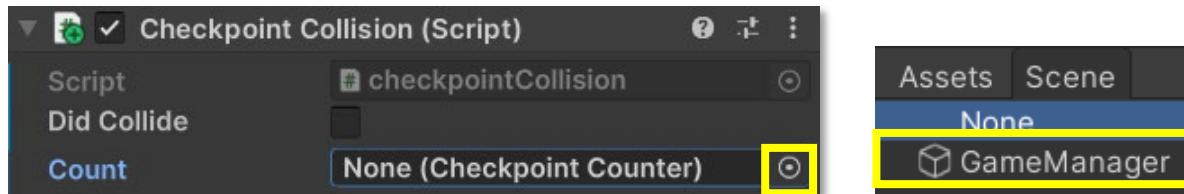
Remember the **Triggered Checkpoints** variable we also created? That variable needs to be updated every time we cross a checkpoint. Since the **Triggered Checkpoints** variable is public, other scripts can use it too. Luckily, we already created that condition in our **checkpointCollision** script.

Open the **checkpointCollision** script. To use the **TriggeredCheckpoint** variable in our script, we must use the name of the script where our original variable is located. In this case, our variable is in the **CheckpointCounter** script. Name this variable **count**.



**82** Save your Script and return to Unity. Select one of your checkpoints and look at the **CheckpointCollision** script in the **Inspector**.

You now have a spot to add in a component. Click the circle to the right of Count and Unity will know what script you are trying to reference. Double click on the **GameManager**.



Repeat this for all of your checkpoint objects.

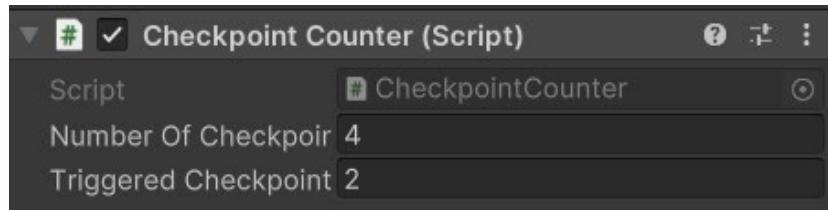
**83** We are now able to access any public variables from the **CheckpointCollision** script. In our **OnTriggerEnter** function, we want to add one to the **TriggeredCheckpoints** variable every time Codey crosses a checkpoint. Increment the count object's **triggeredCheckpoints** variable by one if Codey **collides** with a checkpoint.

```
public class CheckpointCollision : MonoBehaviour
{
    public bool didCollide;
    public CheckpointCounter count;

    void Start()
    {
        didCollide = false;
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player" && didCollide == false)
        {
            didCollide = true;
            count.triggeredCheckpoints++;
        }
    }
}
```

**84** Save your script and return to Unity. Select the **GameManager** object and Press Play. Walk through all your checkpoints and watch the **TriggeredCheckpoint** variable increase!



#### Sensei Stop

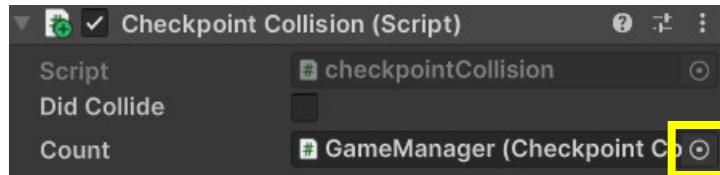
Go through your entire track with your Code Sensei to confirm that both the `numberOfCheckpoints` and `triggeredCheckpoints` values match at the end.

**85** Before we move on, we need to adjust our condition that triggers the winning message in the console when Codey crosses the finish line!

Open the **TriggerFinishLine** script. Create a variable named `checkpointTracker` that allows us to reference the **CheckpointCounter** script.

```
public class TriggerFinishLine : MonoBehaviour
{
    public CheckpointCounter checkpointTracker;
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            print("You Win!");
        }
    }
}
```

**86** Save your script and return to Unity. Select the **FinishLine** in the **Hierarchy**. In the **TriggerFinishLine** component click on the circle to the right and select the **GameManager**.



**87** We need to adjust the **if** condition inside the **TriggerFinishLine** script.

Our **checkpointTracker** variable allows us to access the **numberOfCheckpoints** and **triggeredCheckpoints** variables. Before Codey can win the race, the player must cross all the checkpoints on the track.

Add a second **if conditional** statement that checks to see if the **checkpointTracker's triggeredCheckpoints variable** has the same **value** of the **numberOfCheckpoints variable**.

```
public class TriggerFinishLine : MonoBehaviour
{
    public CheckpointCounter checkpointTracker;
    References
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            if (checkpointTracker.triggeredCheckpoints == checkpointTracker.numberOfCheckpoints)
            {
                print("You Win!");
            }
        }
    }
}
```

**88** We also want to add an else condition that prints "Cheater!" in the console if a player has not gone through all the checkpoints.

```
public class TriggerFinishLine : MonoBehaviour
{
    public CheckpointCounter checkpointTracker;
    References
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            if (checkpointTracker.triggeredCheckpoints == checkpointTracker.numberOfCheckpoints)
            {
                print("You Win!");
            }
            else
            {
                print("Cheater!");
            }
        }
    }
}
```

 **Sensei Stop**

Describe to your Code Sensei how your if condition has changed. Then demonstrate what happens when you cross the finish line without going through all the checkpoints versus when you have gone through all checkpoints.

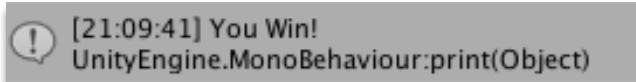
**89** Save your script and return to Unity. Playtest your game. What does the console print the first time you contact the FinishLine?

 [21:03:23] Cheater!
UnityEngine.MonoBehaviour:print(Object)

That's right! We have caught all the cheaters that think crossing the finish line will let them win the game.

---

**90** Go through your entire track and make sure to hit all the checkpoints. What does the console print this time?



You win! Awesome, we have the lap system working! We'll continue building on this in the next section.

---

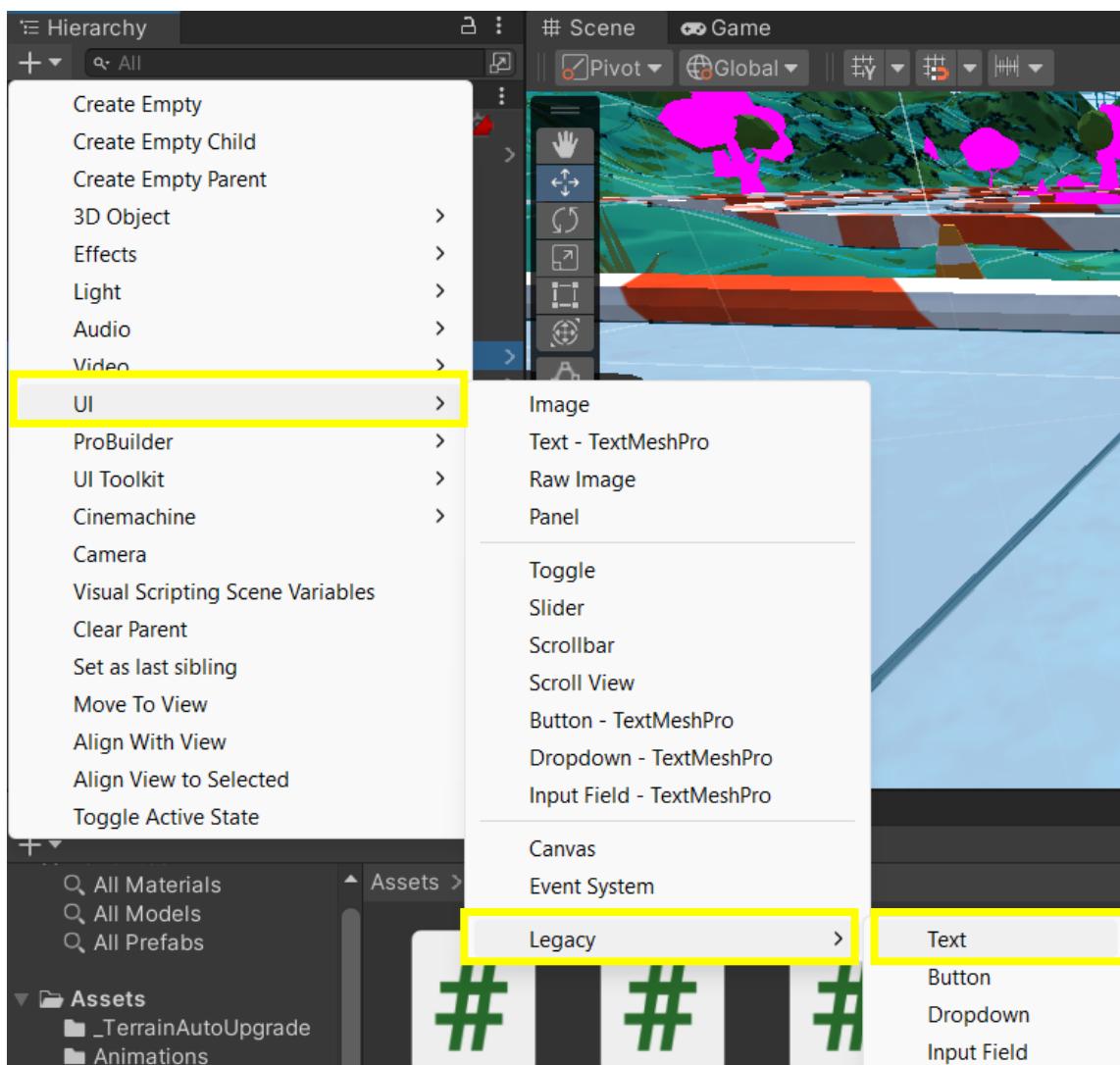
## It's About Time

---

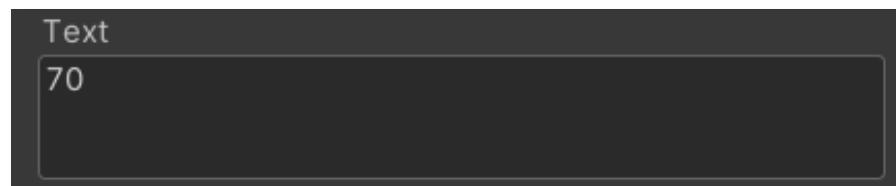
**91** Now that we have our lap system working, let's challenge the player to see if they can make it around the entire obstacle course with a time crunch! You can add a timer to any game to make it more exciting. You will determine the amount of time you give the player based on the size of your track.



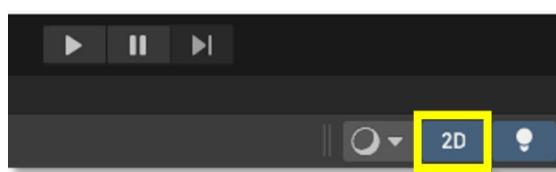
**92** In the **Hierarchy**, create a **UI Legacy Text** object.



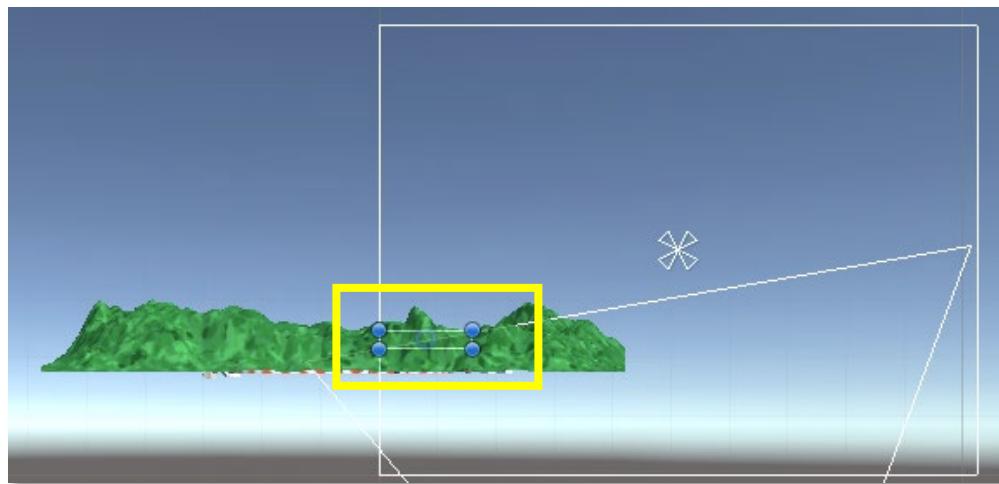
**93** Rename this object **GameTimer**.in the **Inspector** change the **Text** to the amount of time you want the player to have to finish the track. In the example image below, we are giving Codey 70 seconds.



**94** In order to see the **Text**, you must click on **2D**. This can be found close to the **Scene** tab.

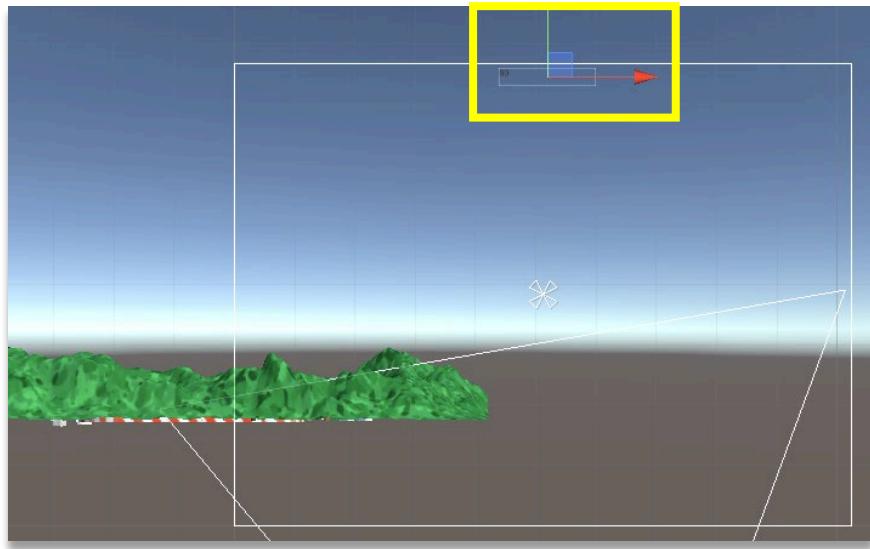


**95** Select the **GameTimer** in the Hierarchy. Using your mouse, zoom out until you can see the **Text Box**.

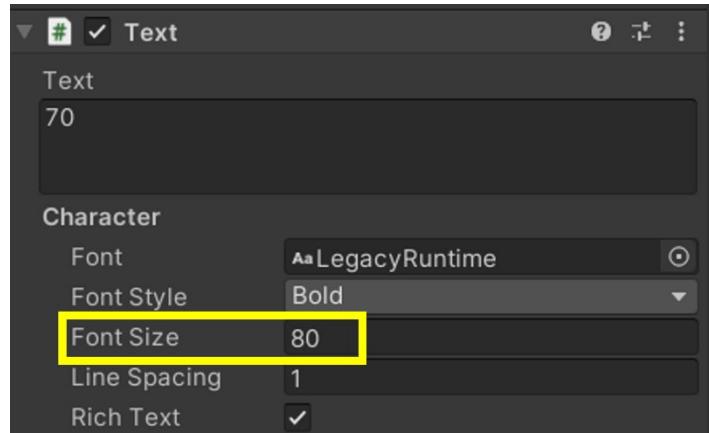


Do you see the other much bigger white box? That is the **Canvas** object you see in the Hierarchy. As long as you position the **GameTimer** object inside this box, you will see it in your Game screen.

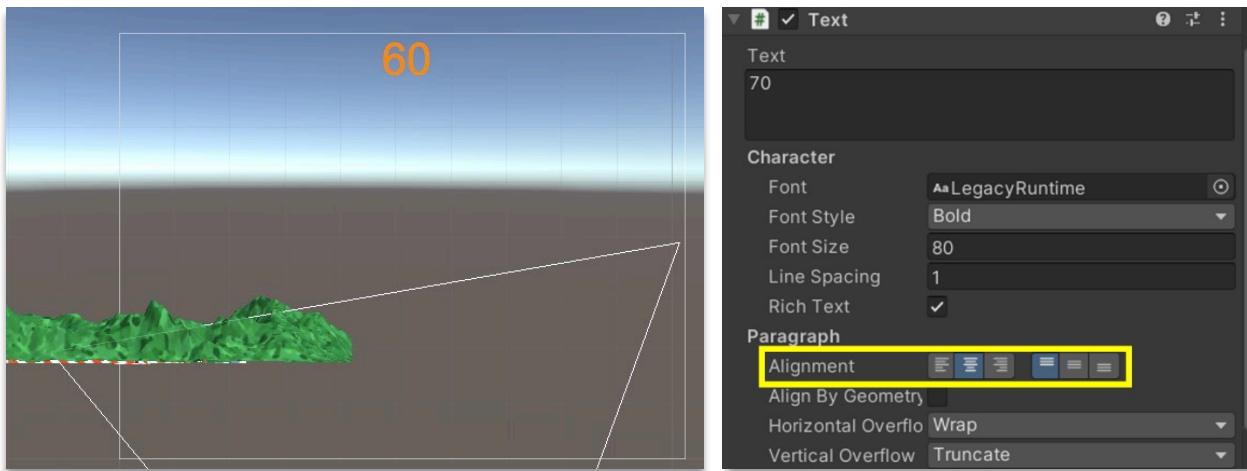
**96** Position your timer somewhere near the top of the Canvas.



**97** The text is a bit small. Adjust it by changing the **Font Size** in the Inspector.



**98** In the **Inspector**, change the **Alignment** component to the center button. This will center the text. Use the **Character** and **Paragraph** components to adjust the text how you like!



**99** One more thing you can add to the **GameTimer** is the **Outline** component. This will help the text show up better by adding a line around the number. Make sure to use a darker color to help it stand out!



**100** Press Play and look at your **GameTimer**!



**101** We also want to create a **CountdownTimer** that tells our player when the race will begin. This timer will count down from 3.

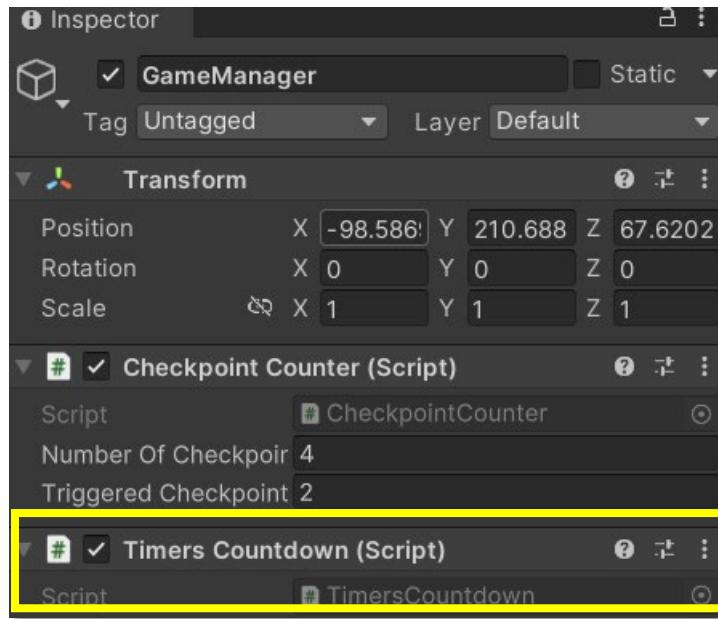
 **Sensei Stop**

Create a second text UI game object inside the Canvas game object. Make the CountdownTimer look different from GameTimer. Give the text a starting value of 3 and place it in the center of the Canvas.

**102** After you have both the Countdown Timer and the Game Timer in your game, we can begin counting them down. Create a new script and name it **TimersCountdown**.



**103** Instead of attaching this script to our timers, we will attach it to the **GameManager**. We said earlier the **GameManager** will control different aspects of our game, and the timers are another one.



**104** Open the Timers Countdown script. You can delete the Start function.

In the Dropping Bombs activity that you built in the Purple Belt, you learned that you must include **Using UnityEngine.UI** at the very top of your script to program **UI** objects.

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

**105** Create two **public Text variables** named **lapTime** and **startCountdown** so we can access and modify the text of our two UI objects.

Create two **public float variables** named **totalLapTime** and **totalCountdownTime** so we can keep track of how much time has elapsed for each timer.

```
public class TimersCountdown : MonoBehaviour
{
    public Text lapTime;
    public Text startCountdown;

    public float totalLapTime;
    public float totalCountdownTime;

    void Update()
    {
    }
}
```

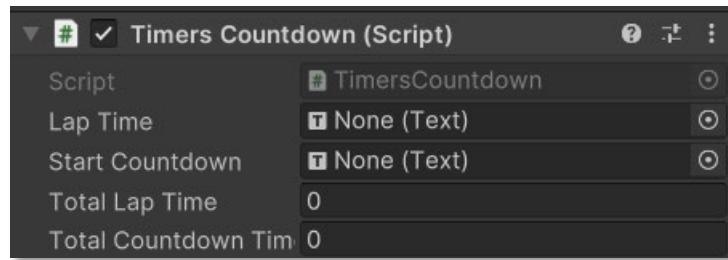
We made all four variables public so we can update them from anywhere in Unity, not just from inside of this script.

 **Sensei Stop**

Explain the difference between an int variable and a float variable to your Code Sensei. Why should we use a float when we want to track how much time has passed instead of an int?

## 106 Save your script and return to Unity.

Select your **GameManager**. In the **Inspector** your **TimersCountdown component** has properties for the four public variables.



### Sensei Stop

Fill these four variables on your own. How much time do you think Codey needs to complete your track? Is it 10 seconds or 600 seconds?

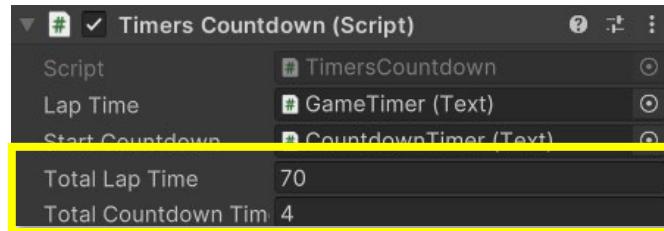
## 107 After you set up all four components, return to the **TimersCountdown** script. Use **Time.deltaTime** to subtract the time that has passed since the last Update call to update both the **totalLapTime** and **totalCountdownTime**.

```
public class TimersCountdown : MonoBehaviour
{
    public Text lapTime;
    public Text startCountdown;

    public float totalLapTime;
    public float totalCountdownTime;

    void Update()
    {
        totalLapTime -= Time.deltaTime;
        totalCountdownTime -= Time.deltaTime;
    }
}
```

**108** Save your script and return to Unity. Click on the **GameManager** to show it in the **Inspector**. Playtest your game. Look at the **TotalLapTime** and **TotalCountdownTime** decrease in the **Inspector**!



**109** Our next step is to update the numbers on the game scene. Return to the **TimerCountdown** script. We need to set the value of our text variables to the value of our counter variables. However, UI Text objects need **strings** and not **floats**.

In the **Update function**, set the text **properties** of our two **UI Text variables** to the time remaining by using **totalLapTime.ToString()** and **totalCountdownTime.ToString()**.

```
void Update()
{
    totalLapTime -= Time.deltaTime;
    totalCountdownTime -= Time.deltaTime;

    lapTime.text = totalLapTime.ToString();
    startCountdown.text = totalCountdownTime.ToString();
}
```

**110** Save your script and return to Unity. Playtest your game and watch your UI timers count down!



**111** Having the decimals show up on the screen does not look so good. We can fix that by using the **Mathf.Round** to only display whole numbers. Stop your game and open the **TimersCountdown** script.

Modify your Update function to round both the **totalCountdownTime** and **totalLapTime** variables before converting them to strings.

```
void Update()
{
    totalLapTime -= Time.deltaTime;
    totalCountdownTime -= Time.deltaTime;

    lapTime.text = Mathf.Round(totalLapTime).ToString();
    startCountdown.text = Mathf.Round(totalCountdownTime).ToString();
}
```

### )Mathf

Mathf might be fun to pronounce, but it just gives us some helper functions to perform with floats! Some examples include rounding and square roots.

**112** Save your **script** and return to Unity. Playtest your game. You will now see the numbers on the timer change by one every second.



**113**

Usually when we start a racing game, we have one countdown timer at the middle of the scene to signal the start of the race and one countdown timer in the top of the scene that tells the player how long they have to complete the race.

This means that our **CountdownTimer** must turn to 0 before our **GameTimer** can start decreasing. Stop your game and open the **Timers Countdown** script.



Add a **public CodeyMove** variable to gain access to the **Speed** variable.

In the Update function, add **conditionals** that checks for the following:

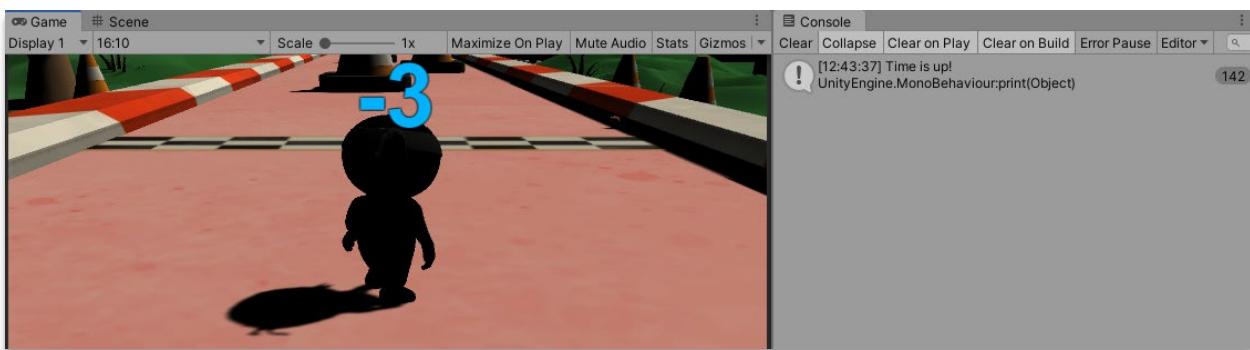
- If the **totalCountdownTime** is greater than 0:
  - subtract **Time.deltaTime** from **totalCountdownTime**,
  - update the **startCountdown** text to the new **value**, and
  - set Codey's **speed** to 0.
- If the **totalCountdownTime** is less than or equal to 0:
  - set the **startCountdown** text to an empty string,
  - subtract **Time.deltaTime** from **totalLapTime**,
  - update the **lapTime** text to the new **value**, and
  - set Codey's **Speed** to 40 or another value.
- If the **totalCountdownTime** is less than 0:
  - print "Time is up!" in the **console**.



### Sensei Stop

Use the provided pseudocode to help you code your advanced timer logic. Collaborate with your Code Sensei on a solution if you get stuck.

**114** Once the **GameTimer** is less than or equal to zero, the message will show in the console.



## Hazardous Conditions

---

**115** With our timers set up, we can now create some obstacles that challenge Codey.

Obstacles are used to prevent the player from completing the challenge. Luckily, these can be any object, and all we need to do is place them strategically on our track.



Use the Unity Asset Store to find objects to place along the track to challenge the player!

---

**116** To keep things organized, create an empty game object, and rename it **Obstacles**. Drag all your objects you want to use as obstacles into this game object.



**117** With your obstacles in the scene, playtest your game. Try to run into every object you placed. Does Codey collide or go through the object? What component are the obstacles missing to make sure Codey properly collides?

Your obstacles need a **Box Collider** so Codey can't go through them.

**118** Use your Ninja Planning Document and the screenshots below as inspiration for your track.



---

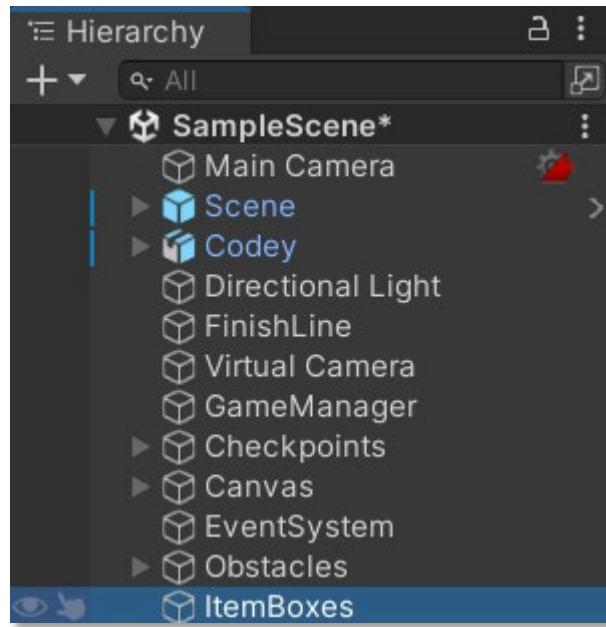
**119** Codey seems to be in trouble as the obstacles are all over the track, but we can add to our game to help the player out!

Power-ups are bonus abilities that many games provide to give players an advantage in their game. You can use the skills you learn from this lesson to add power-ups to any game that you create in the future.

In this section we will create multiple item boxes that will allow Codey to access, then use a power-up. We will use the **Instantiate** function that we learned from Shape Jam!



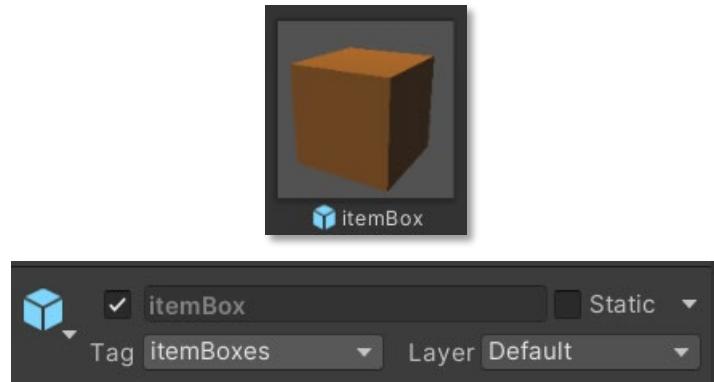
**120** Create an empty game object and rename it **itemBoxes**. We are going to spawn multiple item boxes so this will help us keep things organized.



**121** In the **Hierarchy**, create a **Cube** and rename it **itemBox**. Drag this into the **Prefs** folder.



**122** Add a material and resize the object to customize it to your liking. Also create a new tag named **itemBoxes**, and make sure the itemBox prefab gets the **itemBoxes** tag.



**123** Since it does not need to be in the scene for now, delete the **itemBox** game object from the Hierarchy.

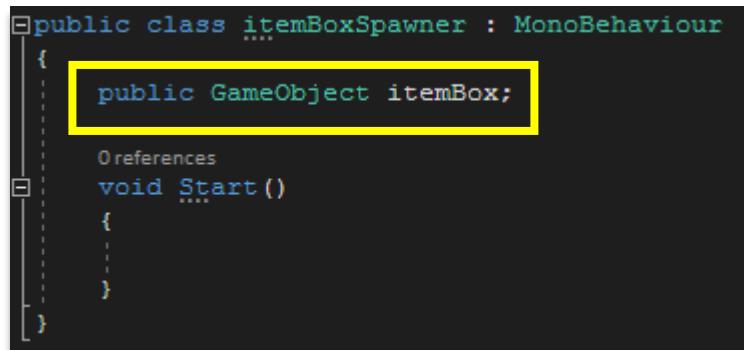
**124** Create another empty game object and rename it **spawnLocation**. Make this object a child of itemBoxes.



**125** This **spawnLocation** object will help us place our item boxes when we clone them in their place. Move the **spawnLocation** object somewhere on your track. It will be hard to see the position since there is no actual object there but in the next few steps, you will be able to see where your boxes are getting spawned.

**126** Create a **new script** in the Scripts folder and name it **itemBoxSpawner**. Attach this script to **spawnLocation**. Open this script in Visual Studio. You can delete the Update function.

**127** We want to use this **script** to spawn our item box **prefab**, so create a **public GameObject** variable named **itemBox**.

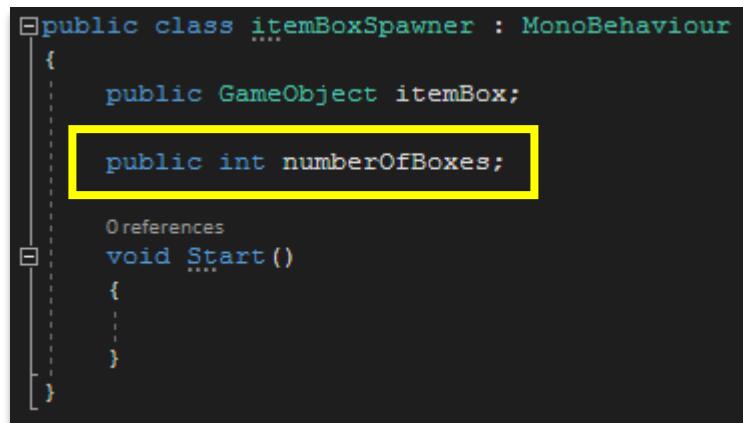


**128** Save your script and return to Unity. Select the **spawnLocations** object. We need to add a game object to the **Item Box** component in the Inspector. Drag the **itemBox** prefab we created into the empty component.



## 129 Open the **ItemBoxSpawner** script.

How many item boxes do you want to spawn? If we don't have a set number, we can end up with millions of item boxes. To avoid this, let's create a maximum number of item boxes we want to spawn in one location. Create a **public int** variable named **numberOfBoxes**.



## 130 We want to create our item boxes when the game starts.

Inside the Start function create a **for loop** that:

- starts at index 0,
- goes up to the value of **numberOfBoxes**, and
- increments the index by one.

Don't worry about the inside of the loop just yet – we will code that in the next few steps!



### Sensei Stop

Write a for loop inside of the Start function based on the three parameters above. When you are done, have your Code Sensei check your code!

**131** After checking in with your Code Sensei, we want to **Instantiate** a new **itemBox** each time the loop runs. Name it **itemBoxClone**.

```
void Start()
{
    for (int i = 0; i < numberOfBoxes; i++)
    {
        GameObject itemBoxClone = Instantiate();
    }
}
```

**132** You might have noticed we have an incomplete **Instantiate** function.

To **Instantiate**, we need 3 parameters:

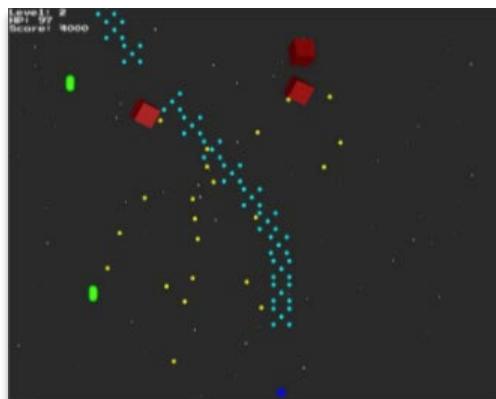
- the object we want to clone,
- the position where we want to place the object, and
- the rotation of the object.

We already have the game object we want to clone, named **itemBox**. The position and rotation of the **spawnLocation's transform**.



Tell your Code Sensei which three parameters you will be using, then finish the **Instantiate** function to create the item boxes.

If you get stuck, look back at the Shape Jam activity.



---

**133** Once you have completed your Instantiate code, save your script and return to Unity. Playtest your game.

If you cannot see the boxes, double-click on the **spawnLocation** object in the Hierarchy and your Scene window will focus on the game object in the game scene.

If you still cannot see the boxes, check the Item Box Spawner script in the Inspector. Is the value of **Number of Boxes** still 0? If that's the case, then we have found the problem! Change the value to the number of item boxes you want.

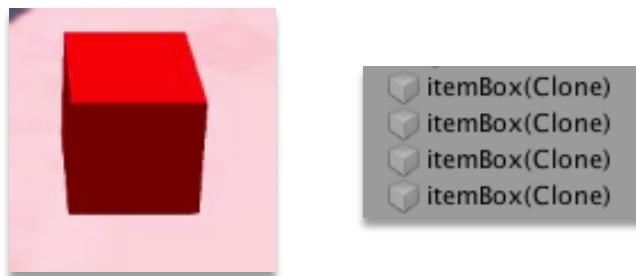


---

**134** Great! Now we can see the item box! But wait, aren't we supposed to see more than one item box?

While still in Play mode, check the Hierarchy. The number of itemBox(Clone) objects should equal the value of Number Of Boxes on the Item Box Spawner script.

Why can't we see all of them?



---

**135** Double-click on one of the **itemBox(Clone)** objects and using the **Move Tool** move it to the right or left. Repeat this process for all of your cloned boxes.



---

**136** Stop the game. Before you move on, position the **spawnLocation** somewhere on the track if you haven't done so already.

Your spawnLocation should be placed slightly above our track pieces. Select one of the track pieces. In the **Inspector** you can see the **Y value position**. Position the spawnLocation right above the track piece's **Y position**.

Play your game and make sure the itemBox spawns on top of the track.



## 137 Open the **itemBoxSpawner** script.

We need to modify the position in the **Instantiate** function. Replace **transfer.position** with **new Vector3(transform.position.x, transform.position.y, transform.position.z)** as this will help us change the exact positions we spawn our boxes.

```
void Start()
{
    for (int i = 0; i < numberOfBoxes; i++)
    {
        GameObject itemBoxClone = Instantiate(
            itemBox,
            new Vector3(
                transform.position.x,
                transform.position.y,
                transform.position.z
            ),
            transform.rotation);
    }
}
```

## 138 Save your script and return to Unity. Playtest your game. The code we adjusted does not move the item boxes, but it will help us offset them in the next few steps.

Stop your game. Create two **public int** variables that we will use to modify the position when we instantiate the boxes.

```
public class itemBoxSpawner : MonoBehaviour
{
    public GameObject itemBox;

    public int numberOfBoxes;

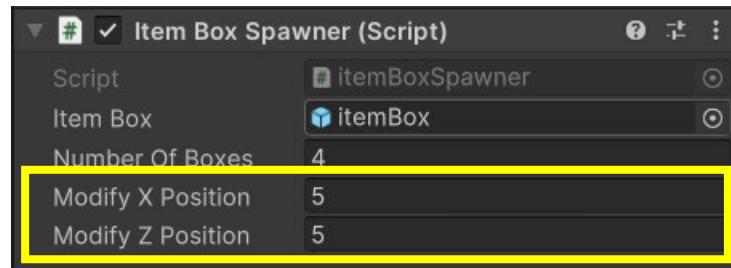
    public int modifyXPosition;
    public int modifyZPosition;

    void Start()
    {
```

**139** Since the Y direction is towards the sky, we only want to modify the **X and Z position**. In the **Instantiate** function, add the **modifyXPosition** to the **transform.position.x** and **modifyZPosition** to the **transform.position.z**.

```
void Start()
{
    for (int i = 0; i < numberOfBoxes; i++)
    {
        GameObject itemBoxClone = Instantiate(
            itemBox,
            new Vector3(
                transform.position.x + modifyXPosition,
                transform.position.y,
                transform.position.z + modifyZPosition
            ),
            transform.rotation);
    }
}
```

**140** Save your script and return to Unity. In the **Inspector**, change the value for the **Modify X Position** and **Modify Z Position** variables.



**141** Playtest your game! Why aren't the item boxes spread out? Stop your game. Let's go back and take a closer look at our **Instantiate** function.

```
GameObject itemBoxClone = Instantiate(  
    itemBox,  
    new Vector3(  
        transform.position.x + modifyXPosition,  
        transform.position.y,  
        transform.position.z + modifyZPosition  
    ),  
    transform.rotation);
```

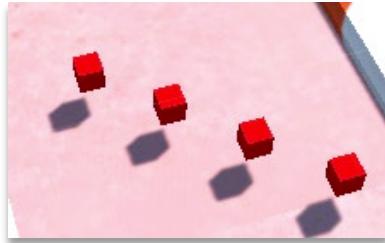
Since the values of `modifyXPosition` and `modifyZPosition` never change, we are still instantiating the boxes in the exact same position.



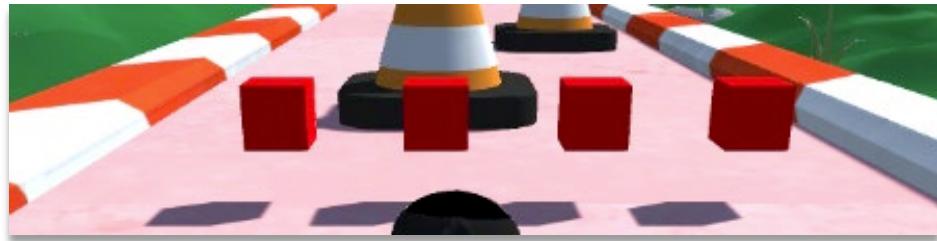
**142** We need to use the for loop's index variable to change the position of each new box.

```
void Start()  
{  
    for (int i = 0; i < numberOfBoxes; i++)  
    {  
        GameObject itemBoxClone = Instantiate(  
            itemBox,  
            new Vector3(  
                transform.position.x + modifyXPosition * i,  
                transform.position.y,  
                transform.position.z + modifyZPosition * i  
            ),  
            transform.rotation);  
    }  
}
```

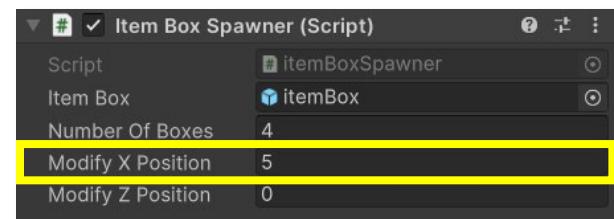
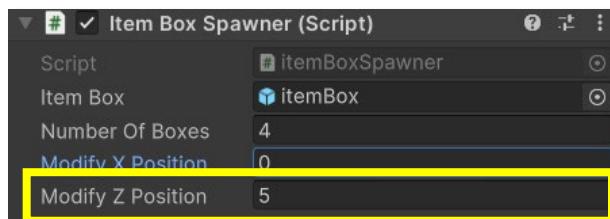
## 143 Save your script and return to Unity. Playtest your game!



Our cubes are spread out, but they positioned in a diagonal line! We want our itemBoxes to spawn across the track like in the image below.



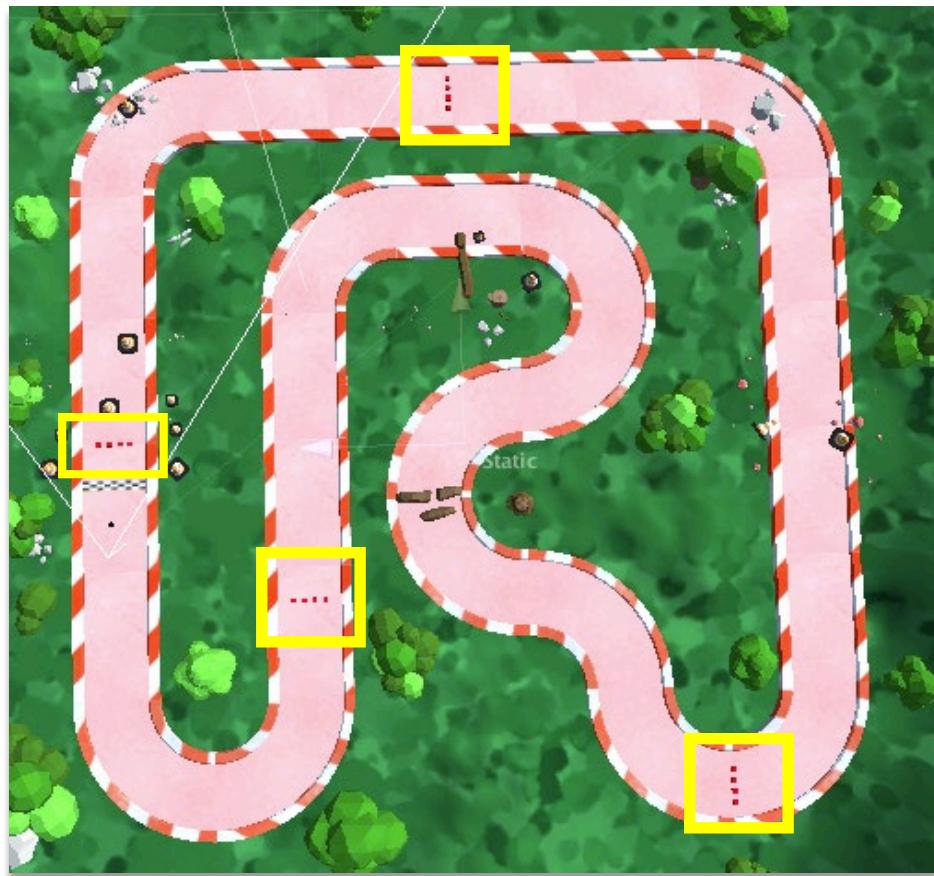
Based on where you place your item boxes on the track, you might not need both the **modifyXPosition** and **modifyZPosition variables**. In our above example, setting one to 0 will make the boxes Instantiate in a straight line.



Demonstrate and explain to your Code Sensei what happens when you set either **modifyXPosition** or **modifyZPosition** variables to zero.

**144** Duplicate the **spawnLocation** game object and move the new spawn locations to other parts of your track. Remember these will be used to help Codey get through the obstacles!

Use your Ninja Planning Document and the example below for inspiration.



If you collide with these new item boxes they currently don't do anything. In the next steps we will focus on adding additional features to them.

---

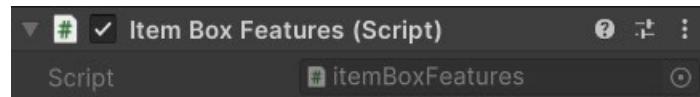
**145** We can make the boxes rotate to show the player they are items to collect and not obstacles to avoid.

Create a new script and name it **itemBoxFeatures**.



---

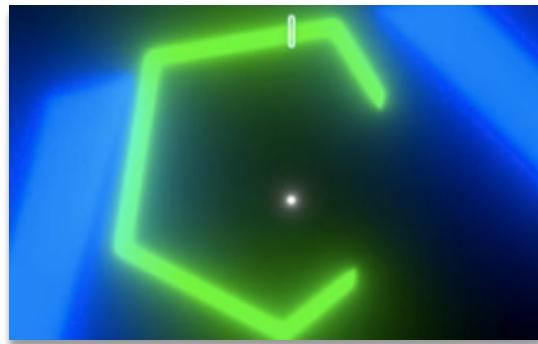
**146** In the **prefabs** folder, select the **itemBox** we created. Click **Add Component** in the **Inspector** and add the **itemBoxFeatures** script.



## 147 Open the script in Visual Studio.

Using what you have learned so far, come up with a solution to rotate the player in 3D space.

In the **SuperShapes** activity from the Purple Belt, you programmed the shapes to rotate in 2D. How can you modify that code and apply it to our 3D item boxes?



### Sensei Stop

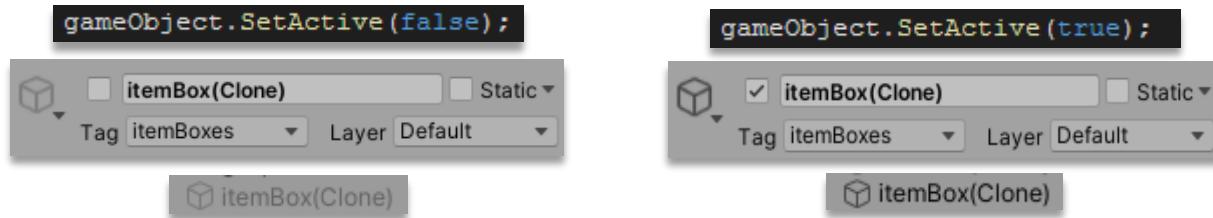
Look back over the Super Shapes curriculum. What pieces of code do you think might be helpful here? Discuss with your Code Sensei what parts of the code might need to change.



**148** We want to make the itemBox disappear when Codey collides with it. Then it should reappear after a few seconds. We can do this using the **Invoke** function!



**149** The **SetActive** function is used to make a game object disappear and reappear.



### Sensei Stop

Discuss with your Code Sensei how to make the item boxes disappear and reappear. What function do you need to use to detect if Codey collided with a box? How can you make sure only Codey can collide?

---

**150** Save your script and return to Unity! Playtest your game and make sure that the item box disappears after Codey collides with it.



---

**151** We need to create a function that will make our item box reappear after Codey collects it.

In the **itemBoxFeatures** script create a new **private void function** named **itemBoxRespawn**.

```
private void itemBoxRespawn()
{
}
```

---

**152** In this function we want to do the opposite of what we did in our OnCollisionEnter function by using the SetActive function to enable our gameObject.

```
private void itemBoxRespawn()
{
    gameObject.SetActive(true);
}
```

**153** Save your script and playtest your game! Are your item boxes respawning? Did you notice that we forgot to Invoke our function?

You previously used the **Invoke** function in the Evil Fortress of Doctor Worm activity. What pieces of code from this activity can we use and modify for our game?



 **Sensei Stop**

Use the `Invoke` function to respawn an item box a few seconds after Codey collides with it. Use previous games to help you remember what is special about the two `Invoke` parameters.

**154** Save your script and return to Unity. Playtest your game and make sure your item boxes disappear and reappear.

## Feel the Power

**155** Codey is having trouble getting through some of the placed obstacles. We can help him out by creating something that can destroy obstacles around the track.

Start by creating a **Sphere** object in the Hierarchy and rename it **Shell**. You can adjust the **Scale**, so it looks like an oval and customize it in any way you'd like.



Once you are done shaping the Shell, drag it into the **Prefabs** folder and delete it from the **Hierarchy**.

We will be programming this **Shell** object to **Instantiate** from Codey's position and go straight from where Codey is standing. The Shell will then destroy any obstacle in its way.

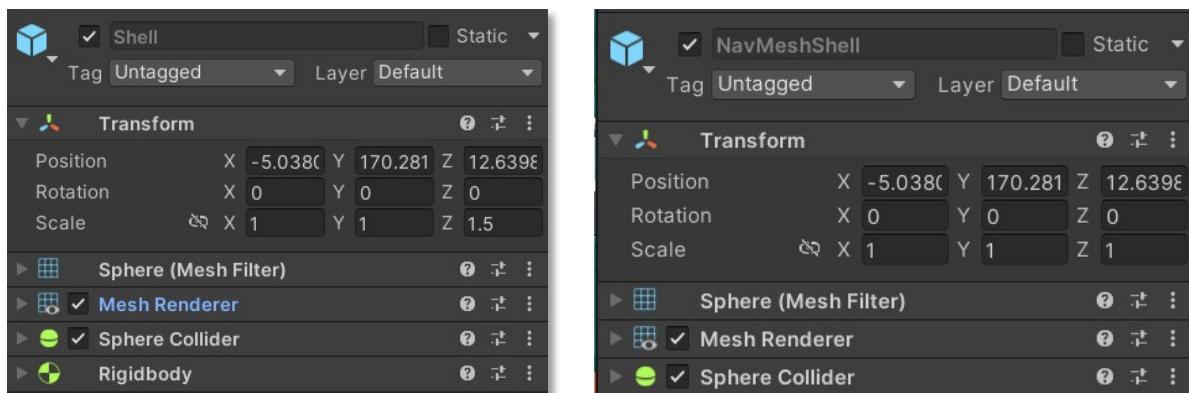


**156** One power-up is not enough. Let's duplicate the **Shell** object and rename it **NavMeshShell**. You might remember that a NavMesh enables an object to navigate through the scene. We will use this shell to find the closest obstacle and destroy it!

Make this new shell visually different from the first!



**157** Add **Sphere Collider** and **Rigidbody** components to the Shell and NavMeshShell prefabs.



---

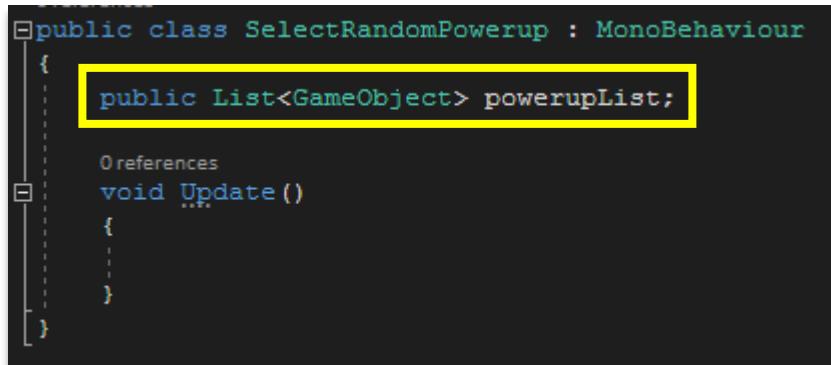
**158** The next step is to create the logic for Codey to use one of the power-ups when he collides with an itemBox.

Create a new script in the **Scripts** folder named **SelectRandomPowerup** and attach it to Codey.



---

**159** Open the script in Visual Studio. You can delete the Start function. To give Codey different power-up options, we need to create a list. This list will contain all the Game Objects that Codey will be able to **Instantiate** later.



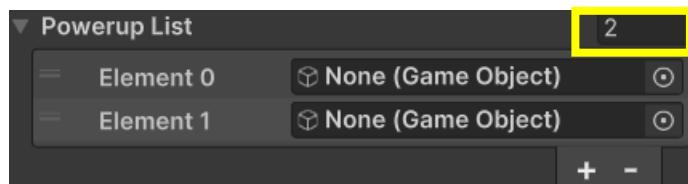
```
public class SelectRandomPowerup : MonoBehaviour
{
    public List<GameObject> powerupList;

    void Update()
    {
    }
}
```

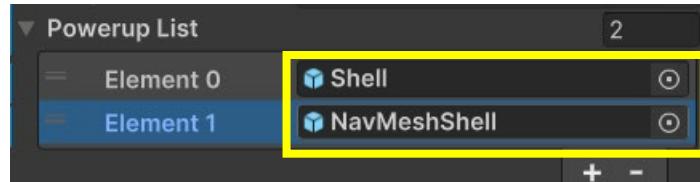
The screenshot shows a code editor window with a dark theme. It displays a C# script named "SelectRandomPowerup". The script starts with a class definition "public class SelectRandomPowerup : MonoBehaviour". Inside the class, there is a public variable "powerupList" of type "List<GameObject>". Below this, there is a method "Update" with an empty body. A yellow rectangular box highlights the line "public List<GameObject> powerupList;".

**160** Save your script and return to Unity. Click on Codey and in the Inspector, we need to update the **Select Random Powerup Script** component.

Change the size of the Powerup List from 0 to 2.



**161** We now have two additional components that we must fill. Drag the **Shell** and **NavMeshShell** from the Prefabs folder into Element 0 and Element 1.



**162** Open the **SelectRandomPowerup** script.

We need a helper variable that will give us a random number between 0 and the total number of Game Objects in our list. Create a new **public int** variable named **randomNumberInList**.

```
public class SelectRandomPowerup : MonoBehaviour
{
    public List<GameObject> powerupList;
    public int randomNumberInList;

    void Update()
    {
    }
}
```

**163** Create an **OnCollisionEnter** function that checks when Codey collides with an object with the tag “**itemBoxes**”.

```
public class SelectRandomPowerup : MonoBehaviour
{
    public List<GameObject> powerupList;
    public int randomNumberInList;

    void Update()
    {

    }

    private void OnCollisionEnter(Collision other)
    {
        if (other.gameObject.tag == "itemBoxes")
        {
        }
    }
}
```

**164** When we collide with an itemBox, we want to set the **randomNumberInList** variable equal to a random number in the range between 0 and the count of our powerup list.

```
private void OnCollisionEnter(Collision other)
{
    if (other.gameObject.tag == "itemBoxes")
    {
        randomNumberInList = Random.Range(0, powerupList.Count);
    }
}
```

**165** We can use this number to select and store a powerup for Codey to use it whenever the player wants.

Create a new **public Game Object** named **chosenPowerup**.

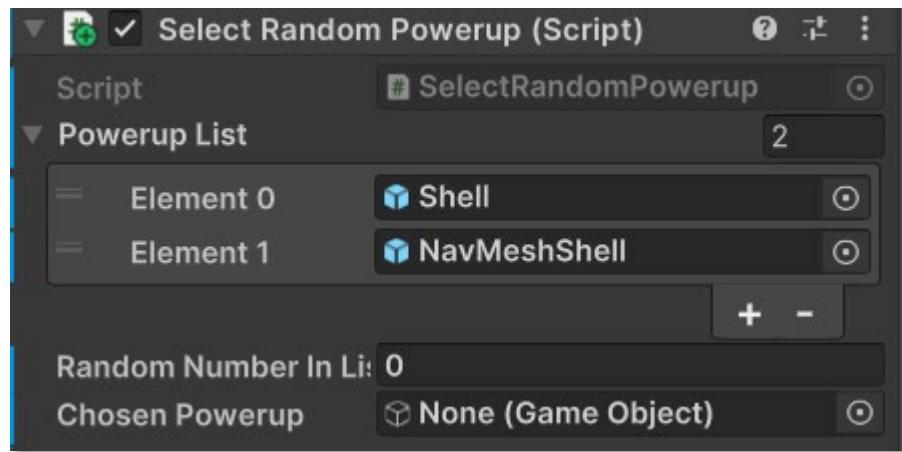
```
public class SelectRandomPowerup : MonoBehaviour
{
    public List<GameObject> powerupList;
    public int randomNumberInList;
    public GameObject chosenPowerup;

    void Update()
    {
```

**166** By default, **chosenPowerup** does not have a Game Object attached. After we collide with the **itemBox**, we want to use **randomNumberInList** to assign a Game Object from the list to the **chosenPowerup** variable.

```
private void OnCollisionEnter(Collision other)
{
    if (other.gameObject.tag == "itemBoxes")
    {
        randomNumberInList = Random.Range(0, powerupList.Count);
        chosenPowerup = powerupList[randomNumberInList];
    }
}
```

**167** Save your script and return to Unity. Playtest your game. Click on Codey and look at **chosenPowerup** in the Inspector.



We have not collided with an item box yet, so **chosenPowerup** is empty.

Collect an item box and see how the public variables in the script change.



Collect more than one box to make sure both your powerups are being randomly selected!

**168** Right now, all Codey can do is collect a powerup. Implement the same logic we used to Instantiate an item box to Instantiate the **chosenPowerup** in front of Codey.

 **Sensei Stop**

In the Update function create a conditional statement that checks to see if the player pressed the space key and if a powerup has been chosen. If both conditions are true, spawn the spawn powerup at Codey's position.

**169** Save your script and return to Unity. Playtest your game and see what happens when you collide with an itemBox and press the spacebar.



Oh no! The shells are spawning on top of each other and pushing Codey up!

**170** We first need to position the spawning shells in front of Codey, not below. Then, we need to make sure Codey can only spawn one powerup after colliding with an item box.

 **Sensei Stop**

Modify the second parameter of the Instantiate function to spawn the chosen powerup in front of Codey. After you instantiate a powerup, reset the value of chosenPowerup to null.

**171** Save your script and return to Unity. Playtest your game. Collide with the item box and press the spacebar. Do you see the item getting placed in front of Codey now?



---

**172** The last thing to do is make our Game Objects destroy the obstacles in our track!

In the **Scripts** folder, create a new script and name is **ShellMovement**. We will use this script to create the basic movement for the **Shell**. Attach this script to the Shell game object in the prefabs folder.



---

**173** Open the script in Visual Studio. You can delete the Start function. When the Shell is Instantiated, we want it to move straight ahead.

In the **Update** function, change the Shell's transform position by adding adding the **transform.forward** vector. Multiply the **transform.forward** vector by **Time.deltaTime** and a speed to make sure the shell always moves at a constant rate.

A screenshot of a code editor window showing a C# script named "ShellMovement". The script contains the following code:

```
public class ShellMovement : MonoBehaviour
{
    void Update()
    {
        transform.position += transform.forward * Time.deltaTime * 50;
    }
}
```

A yellow rectangular box highlights the line of code: `transform.position += transform.forward * Time.deltaTime * 50;`.

**174** When our shell collides with an obstacle, we want to destroy both the shell and the obstacle to clear up the track!



 **Sensei Stop**

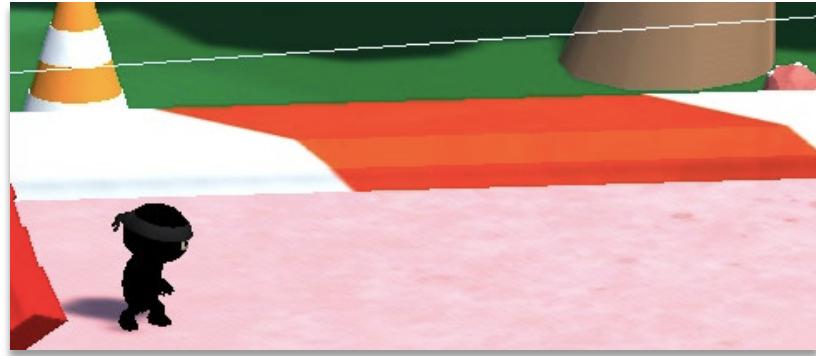
In the ShellMovement script, check to see if the shell collides with an obstacle, then destroy the other obstacle game object and the shell game object. How can we make sure we only run this code when the shell collides with an obstacle?

**175** Save your script and playtest your game! Collect an item box and see what happens when a green shell collides with an obstacle.



**176** If your obstacles are not being removed, you might have forgotten to add the “obstacle” tag to all of your obstacle game objects.

Once you properly tag your objects, playtest your game again. If you made the correct adjustment, you should see the obstacle get destroyed!



While the logic for the Shell game object is now complete, remember that we have not yet programmed the NavMeshShell. If you randomly get the NavMeshShell as the powerup, it will not move!

To make your game more unique, you can replace the Shell game object with any other game object you want!

---

**177** To program the NavMeshShell, we will use a **NavMesh** to have the shell find the closest obstacle, navigate towards it, and destroy it.

Create a new script in the **Scripts** folder and name it **NavMeshMovement**.

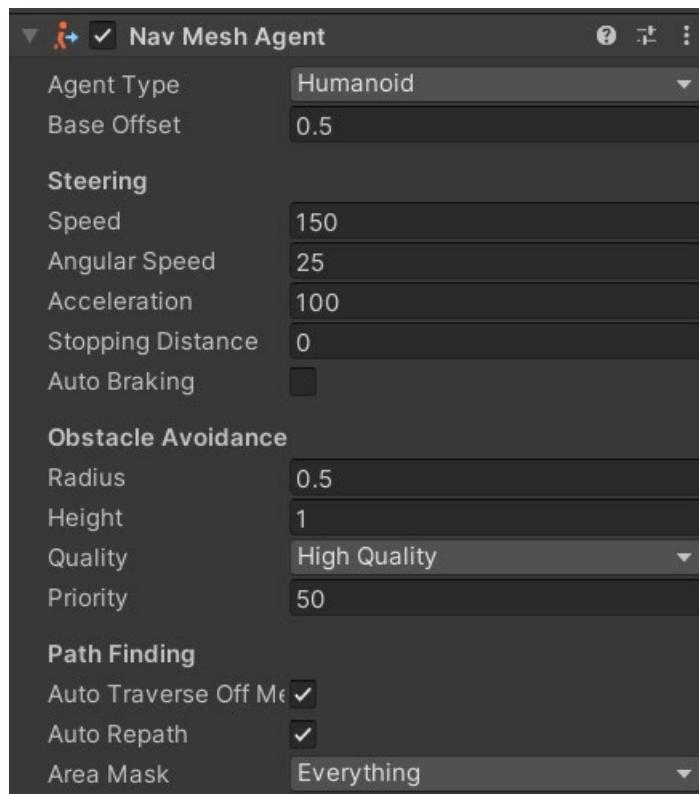


---

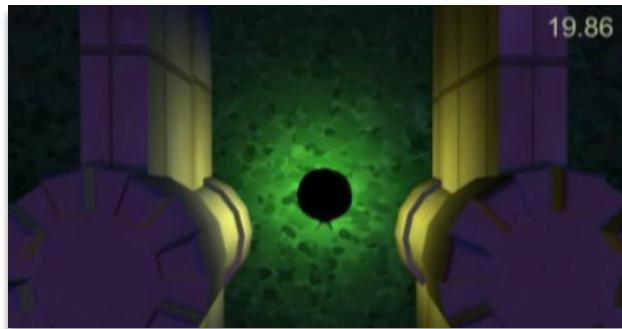
**178** Attach this script to your **NavMeshShell** prefab.

---

**179** Add a **Nav Mesh Agent** component to the NavMeshShell.



**180** Look back at the Labyrinth activity in Brown Belt. How can we program our NavMeshShell to know what an obstacle is? Also think about what code we can borrow from the Shell game object. As a hint, remember to bake your map when you add the right components!



 **Sensei Stop**

Following steps 2 through 8 of Labrinth, make your road walkable and program the logic in the NavMeshMovement script. Use the `GameObject.FindGameObjectWithTag` function to find an obstacle for your shell. Add the collision code from the Shell object so the NavMeshShell can also destroy obstacles.

**181** After you work out a solution, save your script and return to Unity! Playtest your game. When you collect the **NavMeshShell power-up**, watch it go toward an obstacle without directing it!

Now the NavMeshShell logic is complete! You can replace the NavMeshShell game object with any other game object you want.



## 182

Using what you have learned in the previous belts, show off your skills and program your very own powerup!

You can add other power-ups that give Codey an advantage in your game, or you can program something that makes it harder for Codey to get to the finish line.

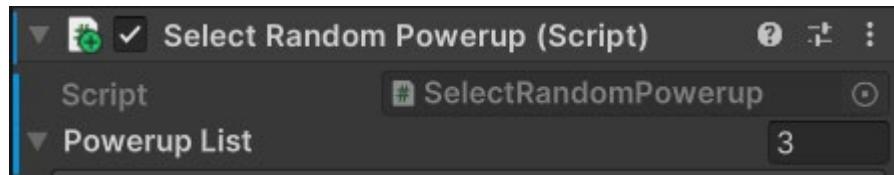
You could give Codey a burst of speed, teleport Codey a short distance forward, or the ability to pass through obstacles.

If you want to make the game harder, you could slow Codey down or make Codey spin randomly.

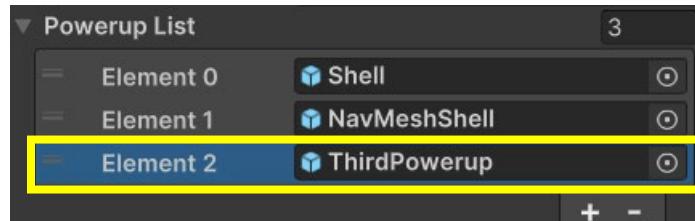
Use your Ninja Planning Document to help you come up with an idea.

## 183

Once you have created your new power-up, make sure to adjust the size of the **PowerupList** in the **SelectRandomPowerup** script located on the Codey object.



Then drag in your third powerup to the additional Element spot.



Have fun and get creative!

## Get Off My Lawn!

---

**184** There is always the possibility that Codey falls off the track and falls onto the Terrain.



When this happens, we want to restart the game all over again to give the player another chance.

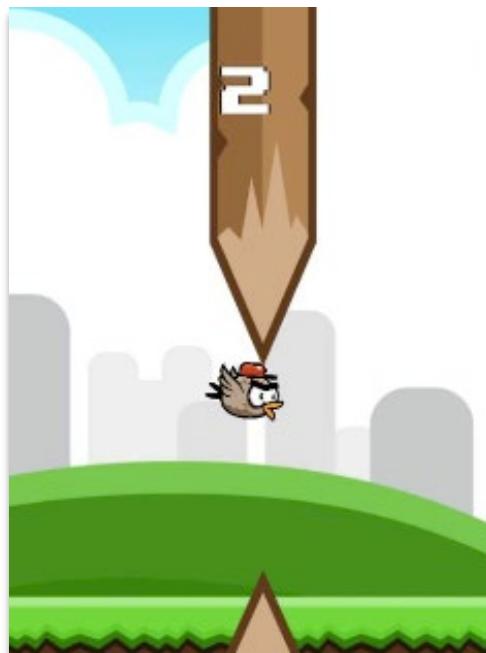
---

**185** Create a new script and name it **Respawn**. Attach this script to Codey.



**186** Open the script. Remember when you learned how to manage the scene in the Meany Bird activity? Feel free to revisit that activity to help you respawn Codey when he falls off the track in this exercise.

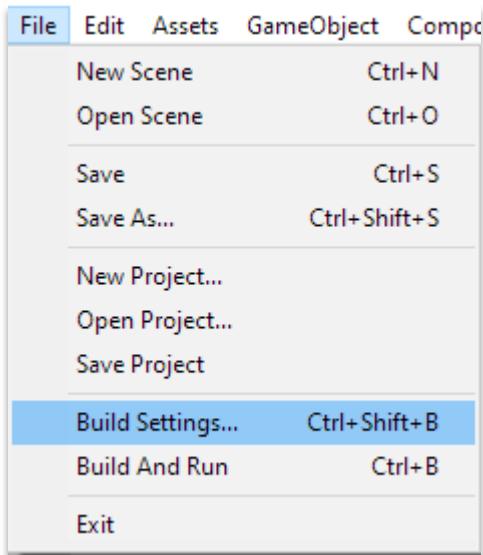
Create the logic to restart the level when Codey touches the Terrain. Use what you have learned so far in Codey Raceway to distinguish the Terrain from all the other game objects in your collision code.



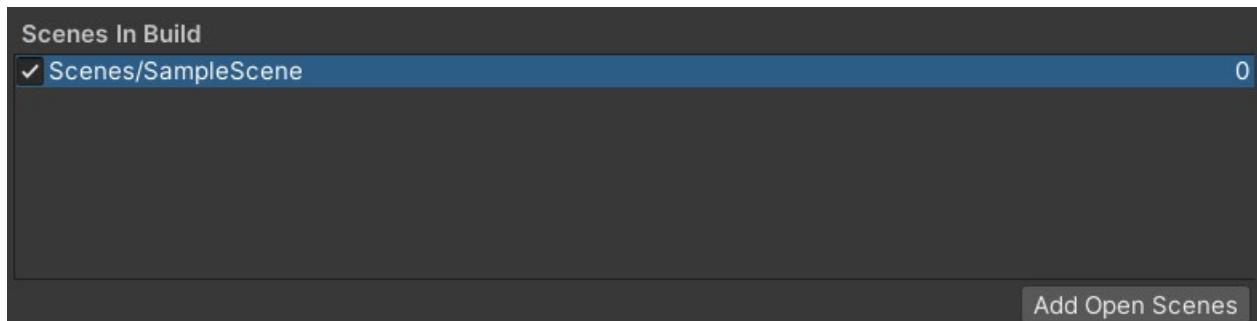
 **Sensei Stop**

Code logic that reloads the scene if Codey collides with any terrain objects.

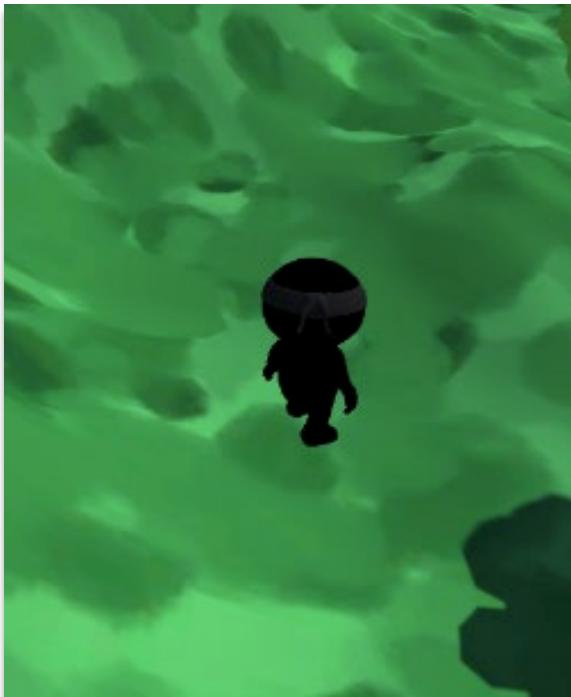
**187** Save your script and return to Unity. In the tabs at the very top of the screen, click on **File** then **Build Settings**.



**188** Click **Add Open Scenes** and make sure your **SampleScene** has been added.



**189** Close out of the window and playtest your game. What happens when Codey goes off the track?



## Victory Lap

**190**

Right now, there's nothing in the game to show whether the player has won or lost. Your challenge is to incorporate a way to show your player that they have won or lost the game. You can do this by creating a new scene.

Replace the print statements we used when the player won or lost with something unique that you come up with!



**Sensei Stop**

You can create a new canvas with a simple win or lose message! You have all creative freedom in this activity! If you need help, discuss your message idea with your Code Sensei.

## Project Submission and Reflection

Now that you have a working game, take time to add your own personal touches to the project. What aspects from other games could you add?

Once you feel like you have a good product that represents your vision of the game, have a Sensei and at least one other Ninja playtest it. Use the Playtest Survey Planning Document for questions to ask them when they finish. Record their answers in your Ninja Planning Document.

Based on the results of the playtest and survey, make changes to your game. Once you are complete, share the updates with your Sensei and fill out the reflection section of your Ninja Planning Document.

Before you submit your game for grading, use the Codey Raceway Project Requirements Checklist to make sure your game has all of the required features.



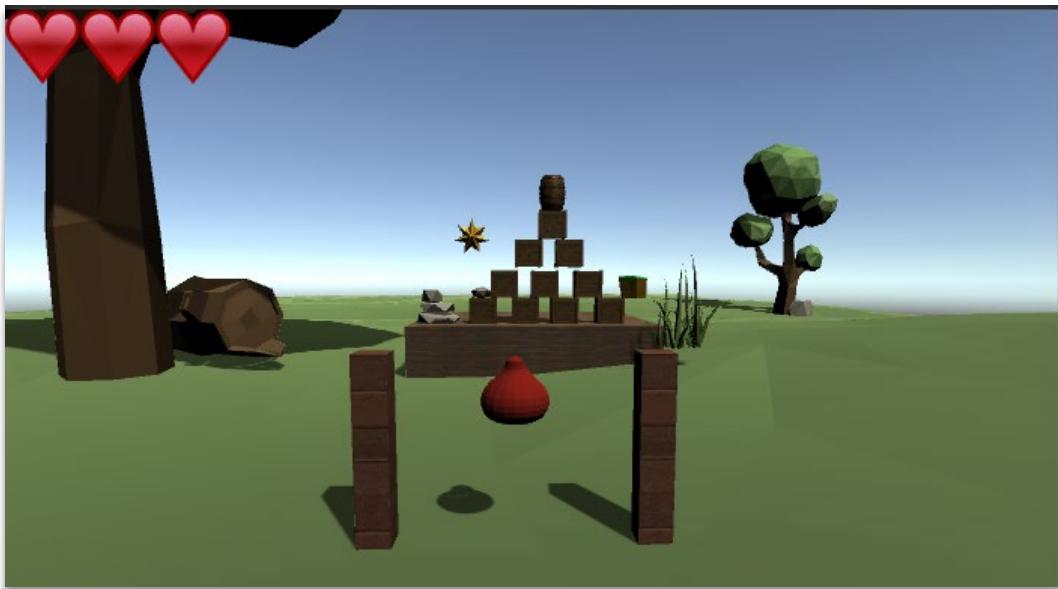
### Ninja Planning Document

Use your Ninja Planning Document to record feedback from Senseis and other Ninjas in your Center.



## Sulky Slimes

Your goal is to plan, program, and playtest a game where the entire game is driven by physics! You will use vectors and forces to create fun and unpredictable experiences for the player.



The game we will create together is influenced by the popular Angry Birds game series, but there are many other games and ideas that are built on similar concepts. As you plan and program, think about other similar games and try to come up with a few new ideas on your own

## Plan and Design

A game built on physics mechanics follows a few specific design principles. As the game designer, you must find a balance between player control and physics.



Kerbal Space Program by Squad  
Built in Unity



Donut County by Ben Esposito  
Built in Unity

The first step is to plan out what your game will look like. Using your Ninja Planning Document, sketch out a basic overview of what you want your game to look like.

### Ninja Planning Document

Take at least 5 but no more than 10 minutes and complete your Ninja Planning Document – Designing a Scene

Take a look at the sample projects below for some inspiration!



## Project Setup

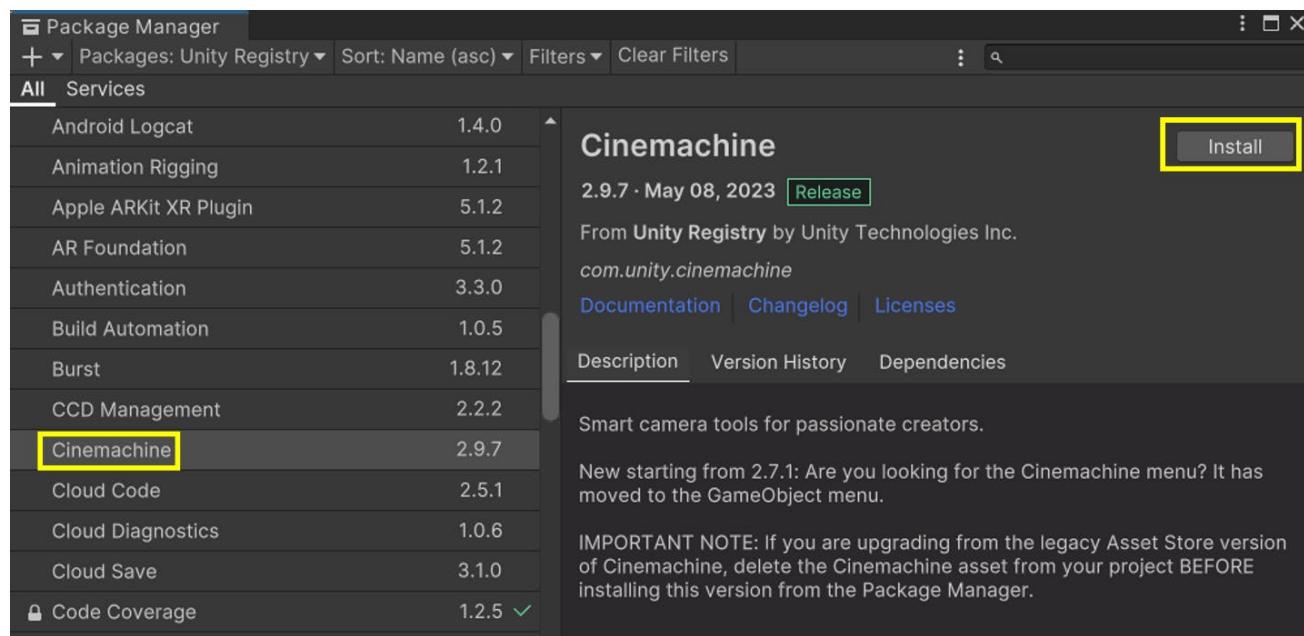
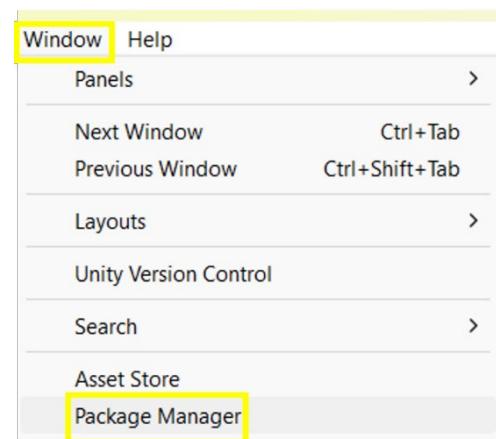
---

- 1** Start a new Unity Project and name it *YOUR INITIALS – Sulky Slimes*.  
Select **3D template**.
- 2** After it loads, rename the Sample Scene to Sulky Slimes.

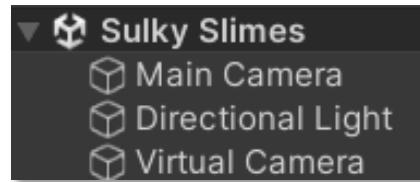


### 3 Open the **Unity Package Manager** from the **Window Menu**.

Find **Cinemachine** and click **Install** in the bottom right of the window.

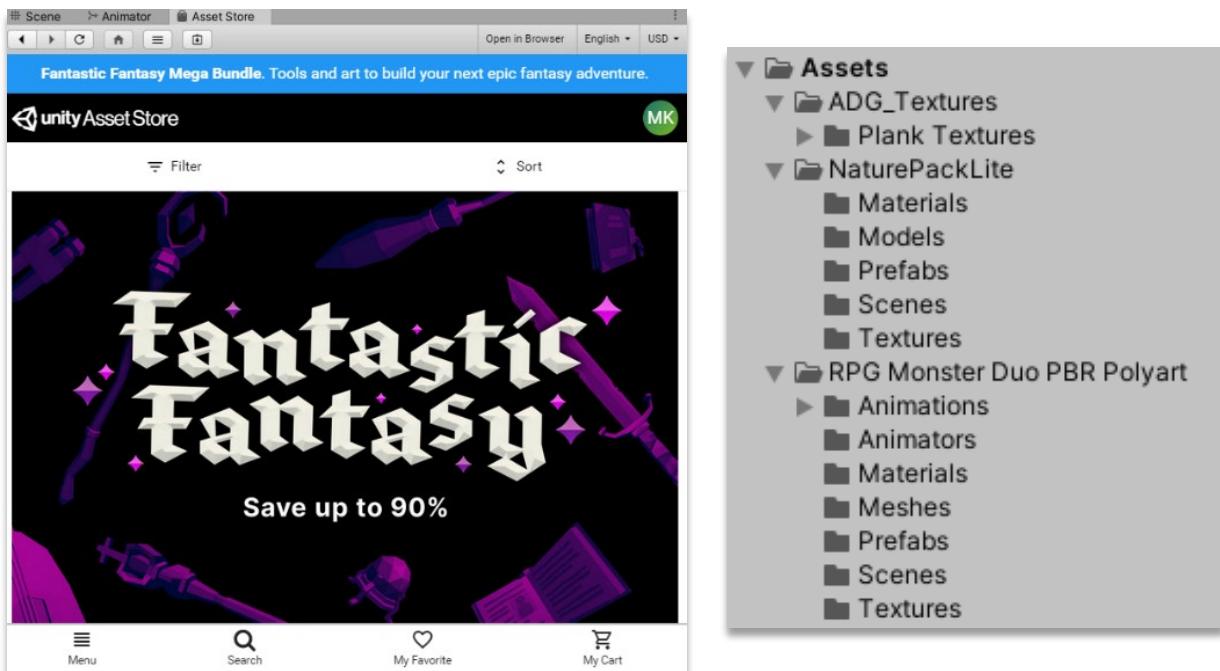


### 4 You should now see a **Virtual Camera** in your object **Hierarchy**. If not, go to **Cinemachine -> Create Virtual Camera**.



## A Whole New World

5 Use Code Ninjas assets, assets from the Unity Asset Store, or your own creations to build a **basic ground**, a **platform** for your targets, and **simple catapult** or **slingshot** structure to launch your object. Your platform and object launcher should both be on the ground. Don't worry about how far apart they are because we will adjust that after playtesting.

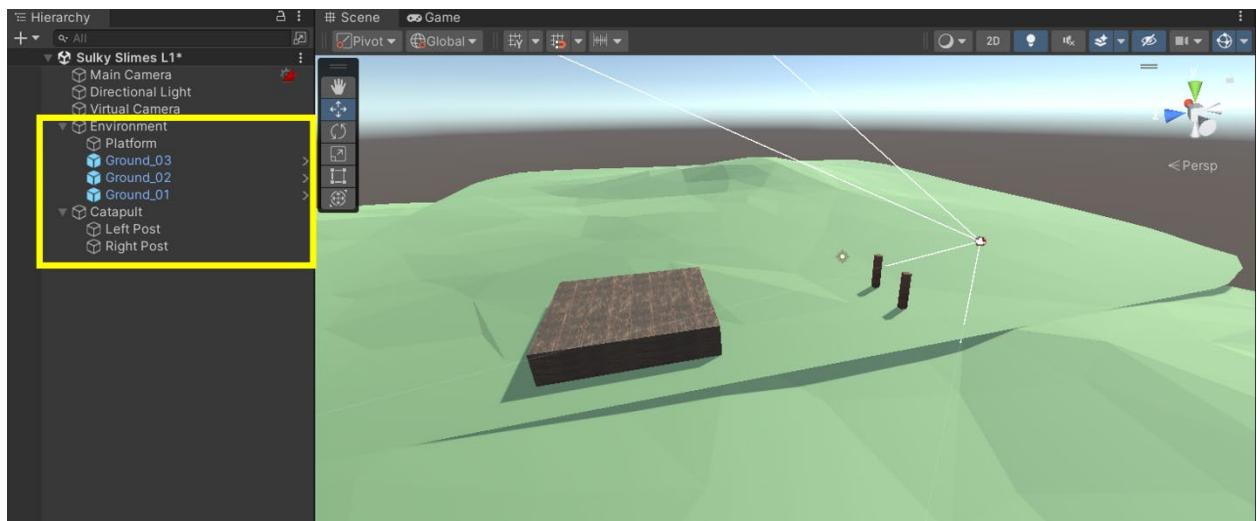


### Ninja Planning Document

Use your Ninja Planning Document to help you  
create your vision in Unity!

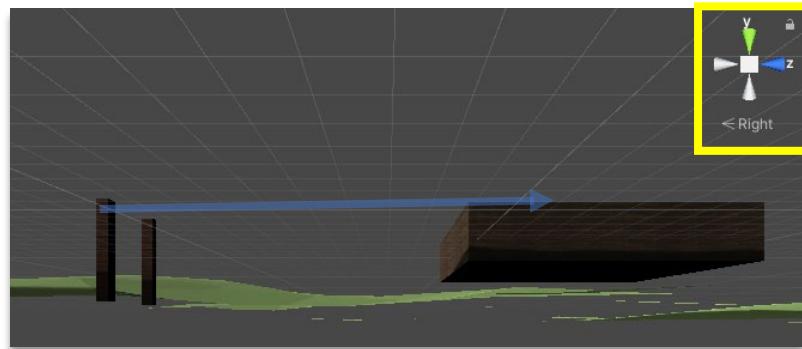
6

Organize your environment in empty **game objects**. You should have an object for your **catapult** and your **environment**. Your **platform** can be a simple object.

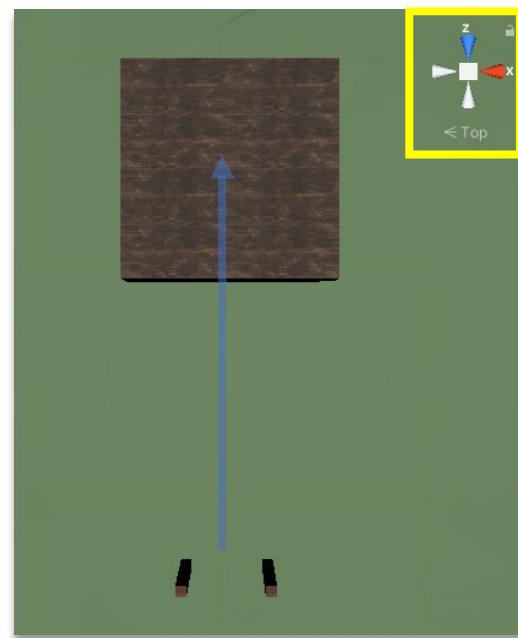


- 7** To ensure our math works out later in the game, we want to align our catapult with the platform. We want to launch our object in the positive z direction. Click the red, green, and blue arrows on the axis indicator in the top right of the scene view to see your scene from the top and the right. Use the screenshots and the blue arrows to help you align your objects.

When viewing from the top, the platform should be above the catapult.



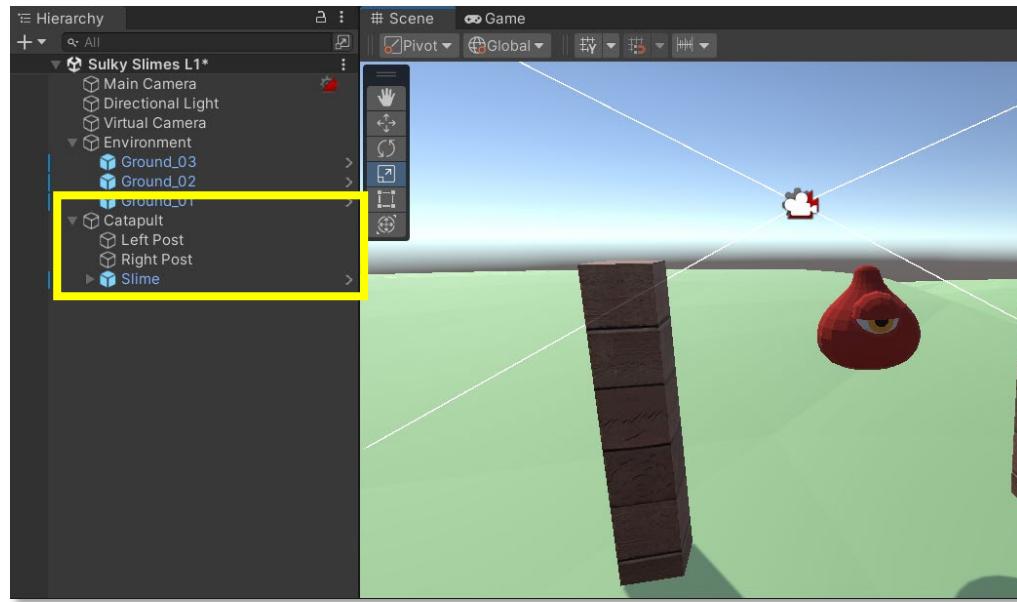
When viewing from the right, the platform should be to the right of the catapult.



Aligning the catapult and the platform like this will also let us easily align the mouse's coordinates with the game's coordinates.

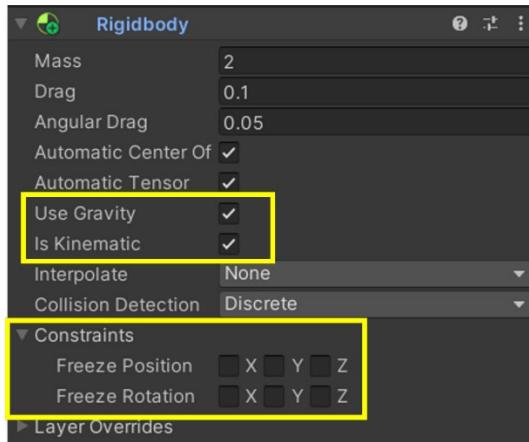
## Character Building

8 Add a **Slime** character to your catapult object. The catapult posts don't serve a gameplay purpose, they are just a visual to enhance the player experience. Feel free to make them any size or shape so they fit the character's size and theme!

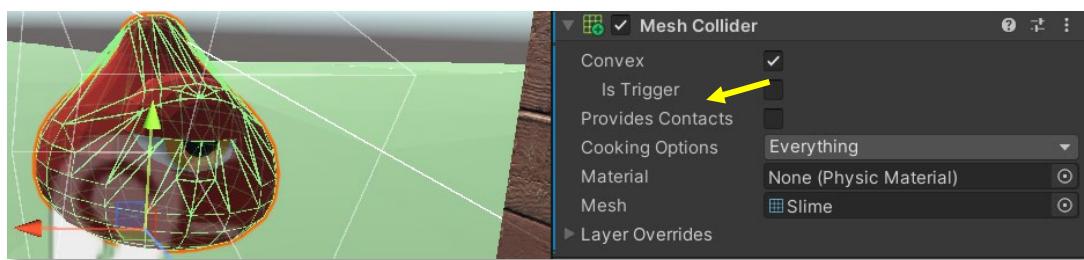
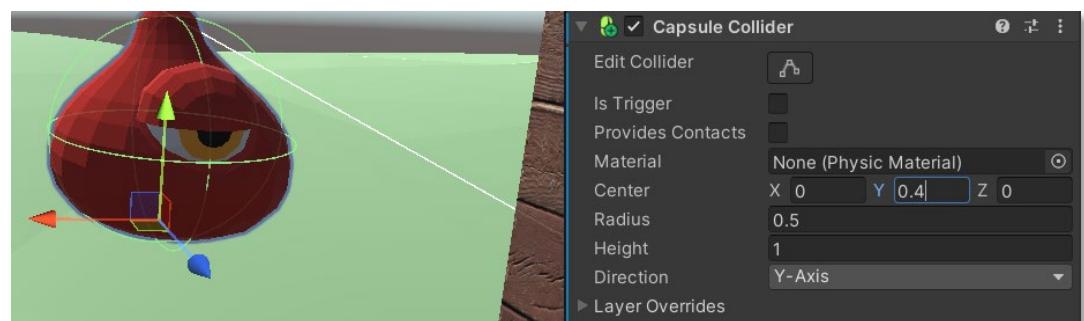
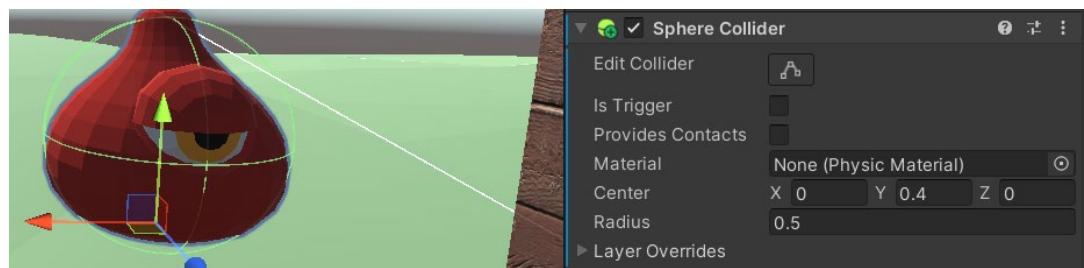
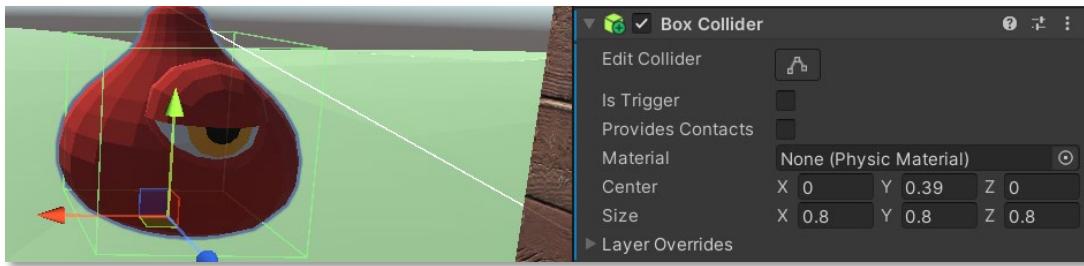


Scale it and have it hover like in the image.

9 Make sure your character has a Rigidbody. We will alter many of these values during our playtests, but make sure **Use Gravity** and **Is Kinematic** are checked and the **Freeze Position** and **Freeze Rotation** boxes are unchecked.

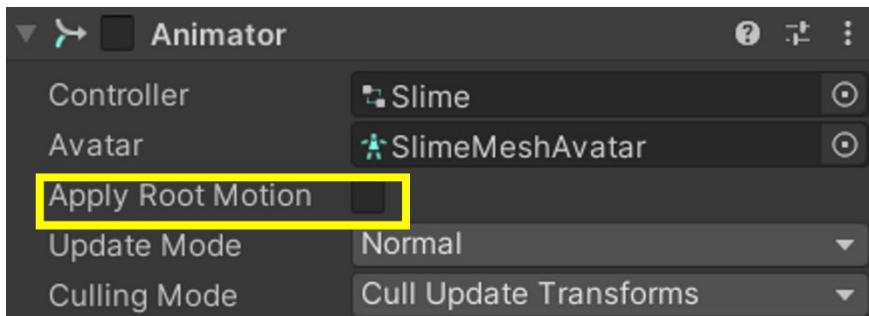


**10** Make sure your character has a **Collider** that is positioned and sized to fit your character.



If your character has a **Mesh Collider**, make sure that the **Convex** option is enabled. The **Convex** option tells Unity that you want this character to interact with other types of **colliders**.

- 11** If your character has an **Animator component**, make sure that the **Apply Root Motion** property is unchecked. If you keep this enabled, the animations could possibly interact with your physics and make your character behave unpredictably!

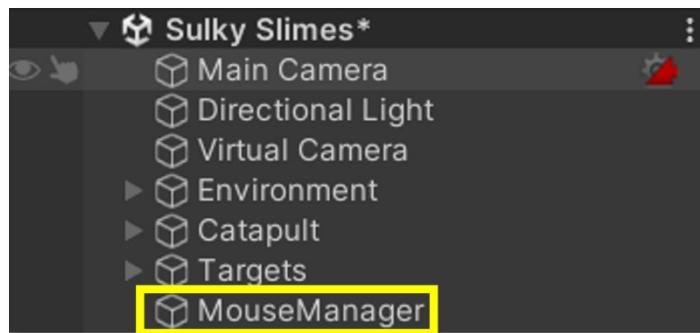


### Imported Assets

No matter what components your character started with, it should have exactly one Rigidbody and one Collider. You will need to modify the assets you download from the Unity Store to make them fit your game!

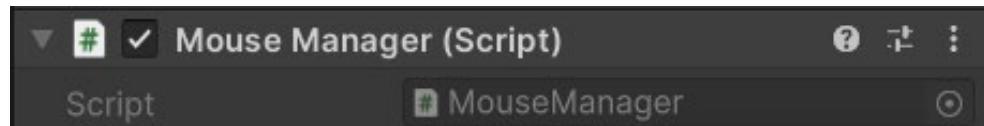
## Mighty Mouse

- 12** We need to use the mouse movements to launch our slime into the air. Add a new empty object and call it **MouseManager**. The placement in the scene does not matter.

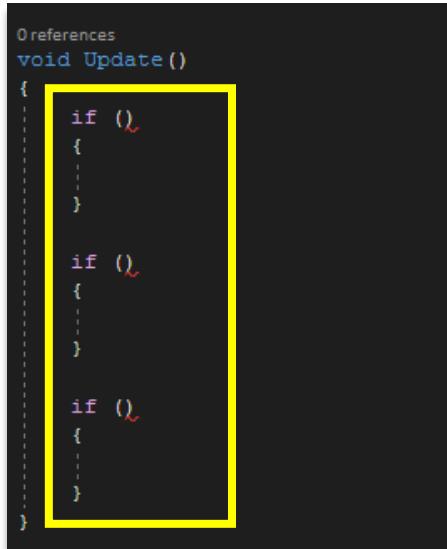


- 13** Create a new **script** in your Script folder in the **Assets** tab and name it **Mouse Manager**.

Attach the **script** to our **MouseManager game object** and open it in Visual Studio.



**14** First, let's listen for user **input**. We want to know when the **left mouse** is clicked, held, and released. In the **MouseManager's Update** function, create three empty **if statements**.



```
0 references
void Update()
{
    if () {
    }

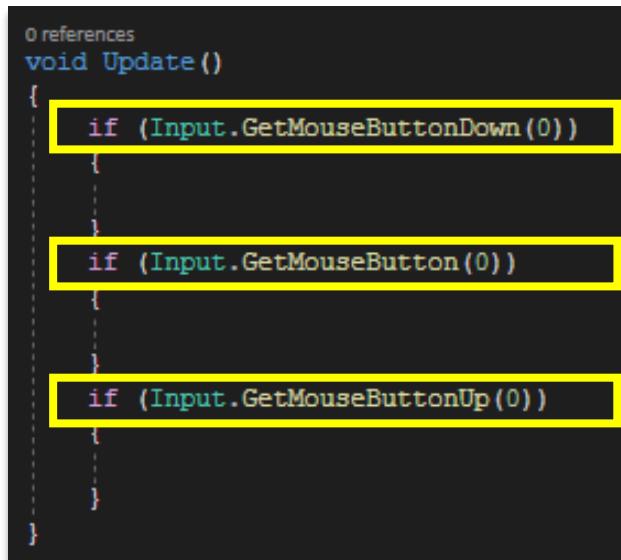
    if () {
    }

    if () {
    }
}
```

A screenshot of a Unity code editor showing the `Update()` method. Inside the method, there are three identical `if` statements, each enclosed in a yellow rectangular selection box. The condition part of each `if` statement is currently empty, consisting only of an opening parenthesis and a closing brace.

**15** To have Unity listen for these three **events**, we need to use three different **functions** on Unity's **Input object**. We need to tell each **function** which mouse button to look at. Unity's **value** for the **left mouse button** is 0.

Code the three **if statements** check to see if the left mouse button is clicked, held, or released.



```
0 references
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {

    }

    if (Input.GetMouseButton(0))
    {

    }

    if (Input.GetMouseButtonUp(0))
    {

    }
}
```

A screenshot of a Unity code editor showing the `Update()` method. The three `if` statements from the previous image have been replaced by calls to `Input.GetMouseButton` with the argument `(0)`. Each of these three new `if` statements is also enclosed in a yellow rectangular selection box.

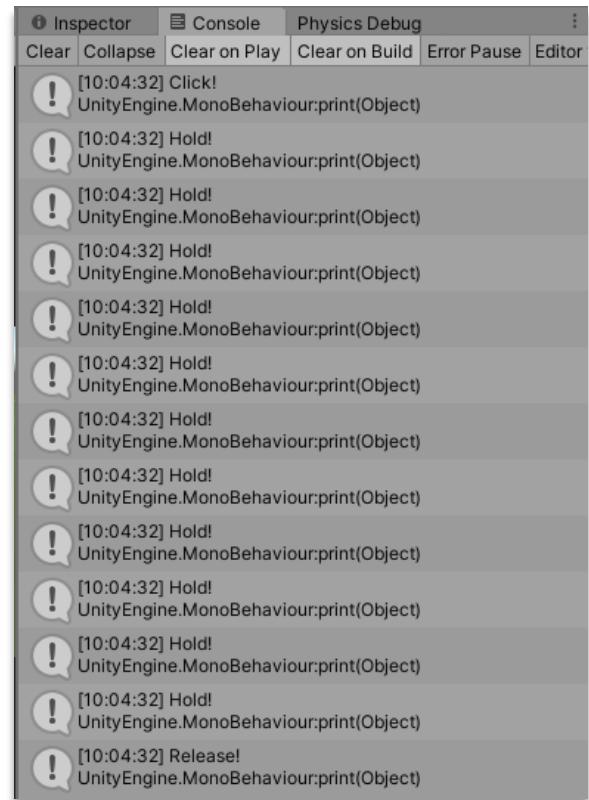
**16** Inside each of the **if statements**, put a **print** statement to check to see when each of these will run.

```
0 references
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        print("Click!");
    }
    if (Input.GetMouseButton(0))
    {
        print("Hold!");
    }
    if (Input.GetMouseButtonUp(0))
    {
        print("Release!");
    }
}
```

**17** Playtest your game. Click the left mouse button and look at the **console**.

Notice how the first **if statement** runs once when the mouse button was initially pressed, the second runs once every **frame** that the mouse button is held down, and the third runs once when the mouse button is released.

In this screenshot, the **Update function** was called thirteen times in less than one second.



**18** The **Update function** will run as fast as your computer can. To see how many frames per second Unity is **rendering**, you can click the Stats button while the game is in play mode.



**19** Our game logic code will go in these three **if statements**. We need to perform some math with **vectors** to convert how far the mouse was dragged into a **force** to apply to the slime.

If we want to see how far the player dragged the mouse, we need to keep track of where the player first clicked.

In the **MouseManager script**, create a **public class variable** named **clickStartLocation** to store the mouse position when the left mouse button is first clicked.

In the first **if statement**, store the value of **Input.mousePosition**.

```
public Vector3 clickStartLocation;

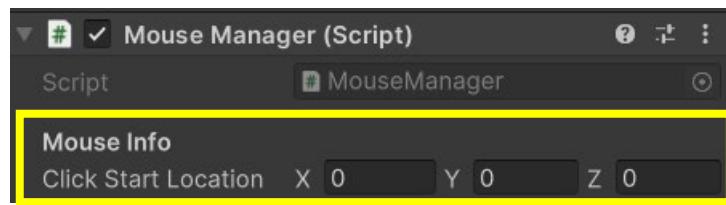
[References]
void Update()
{
    if (Input.GetMouseButton(0))
    {
        clickStartLocation = Input.mousePosition;
    }

    if (Input.GetMouseButton(0))
    {
        print("Hold!");
    }

    if (Input.GetMouseButtonUp(0))
    {
        print("Release!");
    }
}
```

**20** Save your script and playtest your game. Click and observe the value of **clickStartLocation** in the **Inspector**.

Try clicking around the game window to see how the value changes.



### Sensei Stop

Tell a Code Sensei how the mouse's coordinate system works. Where is the origin (where all values are zero)? Does the value of Z ever change? Why?

**21** When the mouse is held, we want to see how far the player has moved it. We want to calculate the difference between the starting click and the current mouse position.

Create a **local variable** named **mouseDifference** and set it equal to the difference of the click's starting location and the current mouse position.

```
public Vector3 clickStartLocation;

References
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        clickStartLocation = Input.mousePosition;
    }

    if (Input.GetMouseButton(0))
    {
        Vector3 mouseDifference = clickStartLocation - Input.mousePosition;
    }

    if (Input.GetMouseButtonUp(0))
    {
        print("Release!");
    }
}
```

**22** Since we are tracking the mouse's position on a 2D screen, **mouseDifference** will only have **X** and **Y** values – the **Z** value will always be 0. We need to somehow take our **X** and **Y** values and create a value for the **Z** coordinate.

Create a new **public variable** named **launchVector**. This is what we will use to send our slime through the air.

```
public Vector3 clickStartLocation;

public Vector3 launchVector;
```

**23** In the second **if statement**, set **launchVector** equal to a new **Vector3** that has the **x** and **y** values of our mouse difference **variable**. Since the mouse doesn't have a **z** value, you should use the **y value** to calculate the **z value** of our new **vector**.

```
public Vector3 clickStartLocation;
public Vector3 launchVector;

References
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        clickStartLocation = Input.mousePosition;
    }

    if (Input.GetMouseButton(0))
    {
        Vector3 mouseDifference = clickStartLocation - Input.mousePosition;
        launchVector = new Vector3(
            mouseDifference.x * 1f,
            mouseDifference.y * 1.2f,
            mouseDifference.y * 1.5f
        );
    }

    if (Input.GetMouseButtonUp(0))
    {
        print("Release!");
    }
}
```

We are using how far the mouse was dragged to determine the direction we launch our slime. Each of our **launchVector's coordinates** have a multiplier to make sure the slime goes in the right direction! Start off with the **values** provided here, but you can change these **values** based on your **playtesting**. These multipliers will fine tune the direction!

## To Infinity

**24** Since we are using this new **launchVector** to determine the direction of the launch only, we need to **normalize** it and use a constant to determine the amount of force to apply.

If you need a refresher on how to **normalize** a vector, look at Brown Belt's **Shape Jam!**

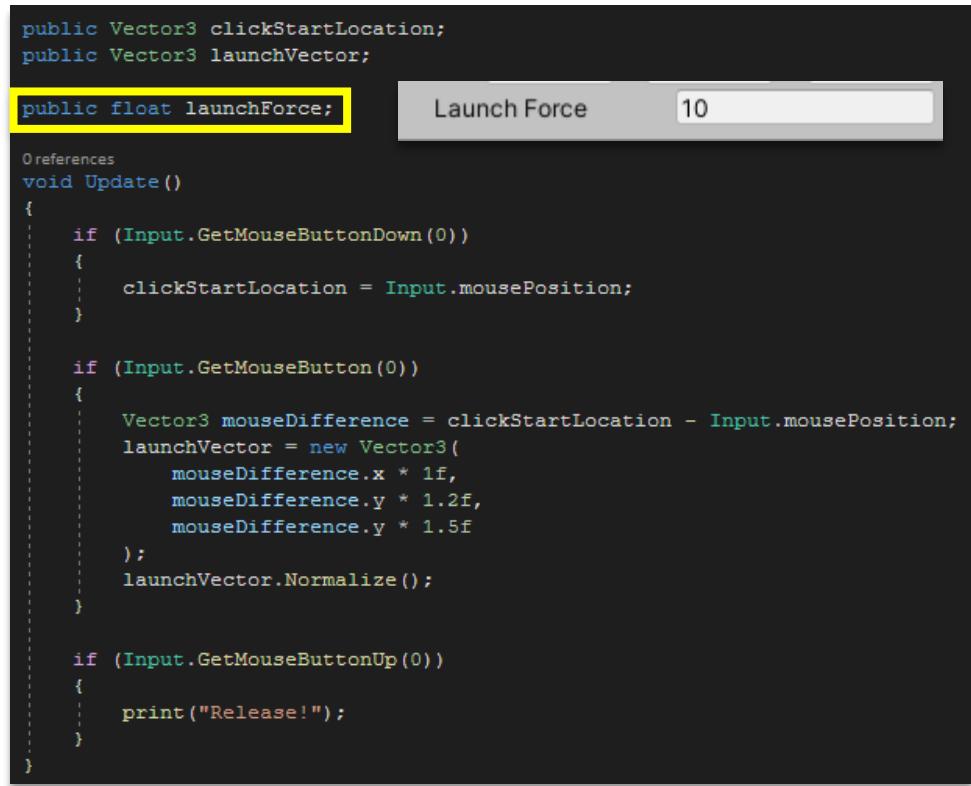
```
public Vector3 clickStartLocation;
public Vector3 launchVector;

[References]
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        clickStartLocation = Input.mousePosition;
    }

    if (Input.GetMouseButton(0))
    {
        Vector3 mouseDifference = clickStartLocation - Input.mousePosition;
        launchVector = new Vector3(
            mouseDifference.x * 1f,
            mouseDifference.y * 1.2f,
            mouseDifference.y * 1.5f
        );
        launchVector.Normalize();
    }

    if (Input.GetMouseButtonUp(0))
    {
        print("Release!");
    }
}
```

**25** Add a **public float** named **launchForce** and give it a starting **value** of 10 in the Unity Inspector. We need to **playtest** before we know exactly what a good number will be!



```
public Vector3 clickStartLocation;
public Vector3 launchVector;

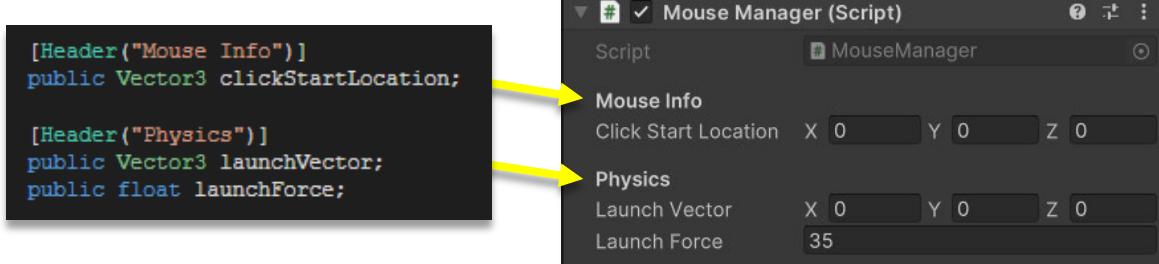
public float launchForce; // This line is highlighted in yellow

[Header("Mouse Info")]
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        clickStartLocation = Input.mousePosition;
    }

    if (Input.GetMouseButton(0))
    {
        Vector3 mouseDifference = clickStartLocation - Input.mousePosition;
        launchVector = new Vector3(
            mouseDifference.x * 1f,
            mouseDifference.y * 1.2f,
            mouseDifference.y * 1.5f
        );
        launchVector.Normalize();
    }

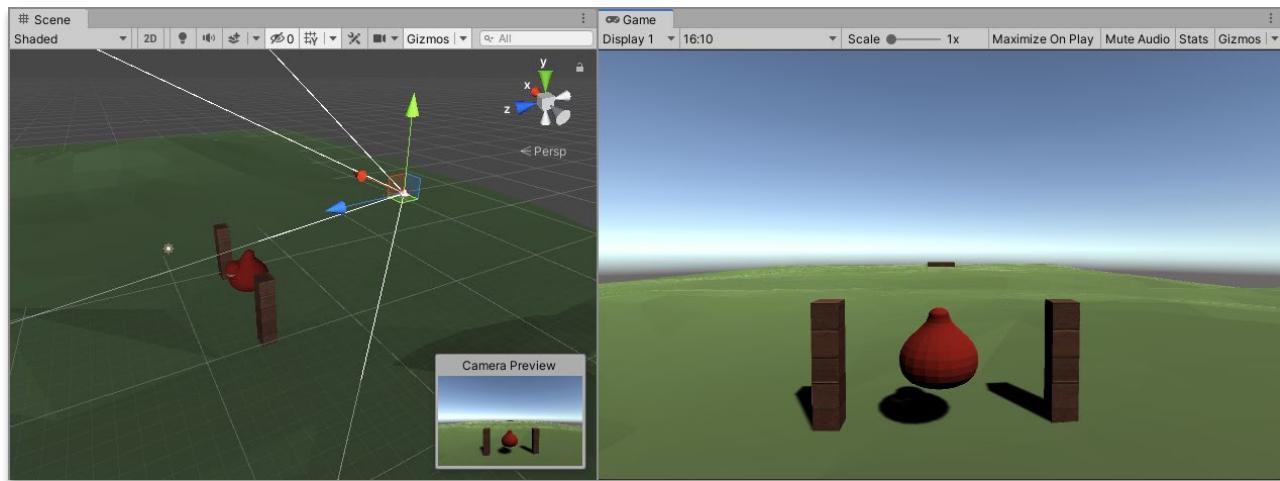
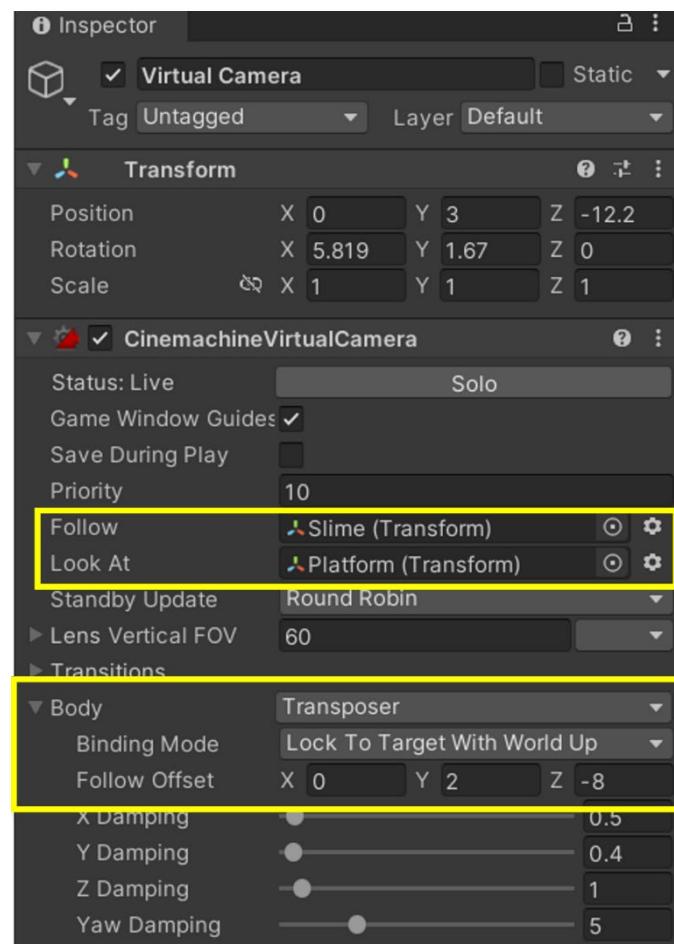
    if (Input.GetMouseButtonUp(0))
    {
        print("Release!");
    }
}
```

**26** Now that we have a few different **public variables**, we can use a special line of code to set up **headers** to help us stay organized. Add **[Header("Mouse Info")]** and **[Header("Physics ")]** above the related **variables**.

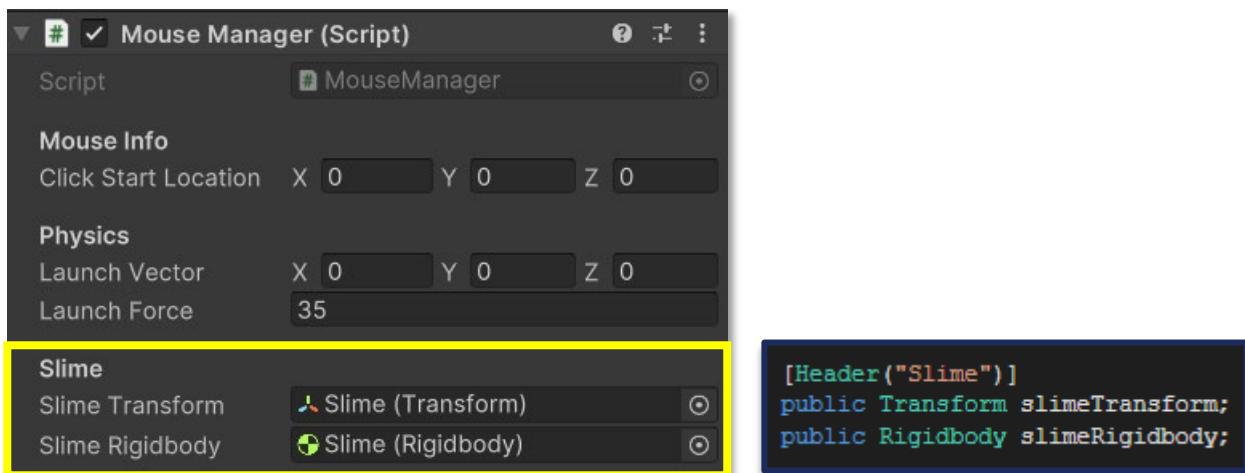


**27** Before we write the code that launches the slime, we need to make sure the **camera** is behind the slime.

Open the **Virtual Camera** object and adjust the **Cinemachine** **Virtual Camera** component. The **camera** should follow the slime and look at the platform. Set the **values** of the **Body's Follow Offset** to be behind and slightly above the slime. Your numbers might be different than the image.



**28** Now that we have our **camera** positioned and we are tracking how far the **mouse** has moved, we can use it to launch the slime. Open the **Mouse Manager script** again. We need to add two **variables**, a **public Transform** and **public Rigidbody**.

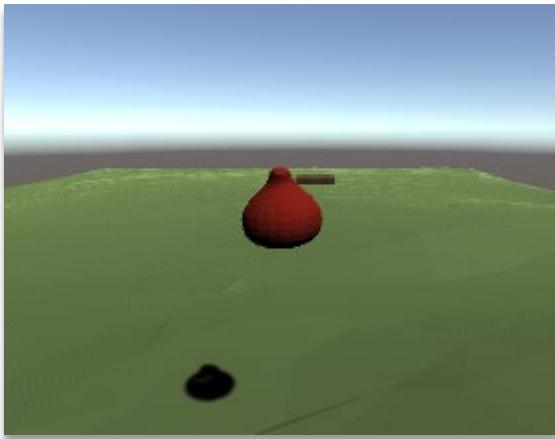


**29** Now that we access to the slime's transform and rigid body, we can program the logic to launch it! When the mouse button is released, set the slime's rigid body is kinematic to false to enable forces. Then, add a force.

#### Sensei Stop

Attempt to code these two lines of code. When the user releases the mouse button, set the slime's rigidbody's isKinematic property to false. Then, add an impulse force to rigidbody in the direction of our launch vector with the strength of our launch force. Reference Robomaina from Brown belt if you need help adding the force.

**30** Playtest your game. Adjust the **launch force** based on your playtests. Did your slime go straight up in the air? Try changing the **z coordinate** of the **launch vector**. Did your slime go in a random direction? Make sure that your launcher and platform are aligned properly. Did it flop to the ground or fly too far? Adjust your **launchForce**.



**31** We need to give the player a way to reset the slime. We need to create **variables** that store the slime's original **position** and **rotation** when the game starts. When the user presses a key or right clicks the mouse, we need to reset the slime's **isKinematic property**, **position (Vector3)**, and **rotation (Quaternion)**.



#### Sensei Stop

Write the code described to reset the slime. What types of variables do you need? Playtest your game and make sure your game works after multiple resets.

You can use **Input.GetMouseButton(1)** to check to see if the user right clicked or **Input.GetKeyDown("space")** to see if the user pressed the space bar.

**32** One problem with our game is that we aren't giving **feedback** to the player before they launch the slime. We can write code that makes the slime follow the mouse movements as the player is dragging the mouse to launch.

Before we **normalize** the launch **vector**, we can use it to move the slime. Because the mouse **coordinates** are based on the size of the computer screen and not the game scene, we need to **divide** our launch **vector** by a large number to make sure the slime doesn't fly out of the scene when the user drags the mouse.

```
if (Input.GetMouseButton(0))
{
    Vector3 mouseDifference = clickStartLocation - Input.mousePosition;
    launchVector = new Vector3(
        mouseDifference.x * 1f,
        mouseDifference.y * 1.2f,
        mouseDifference.y * 1.5f
    );
    slimeTransform.position = originalSlimePosition - launchVector / 400;
    launchVector.Normalize();
}
```

That's all the programming we need to create a physics-based game. We respond to user input, calculate and apply forces, and reset the game objects.

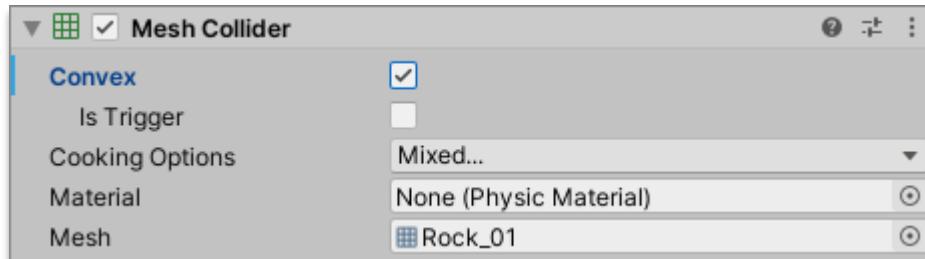
## Reach for the Star

**33** We now need to give the player a **goal** to complete. We need to add **objects** for the player to launch into and stars that the player needs to collect, but you should design your scene based on your planning document.

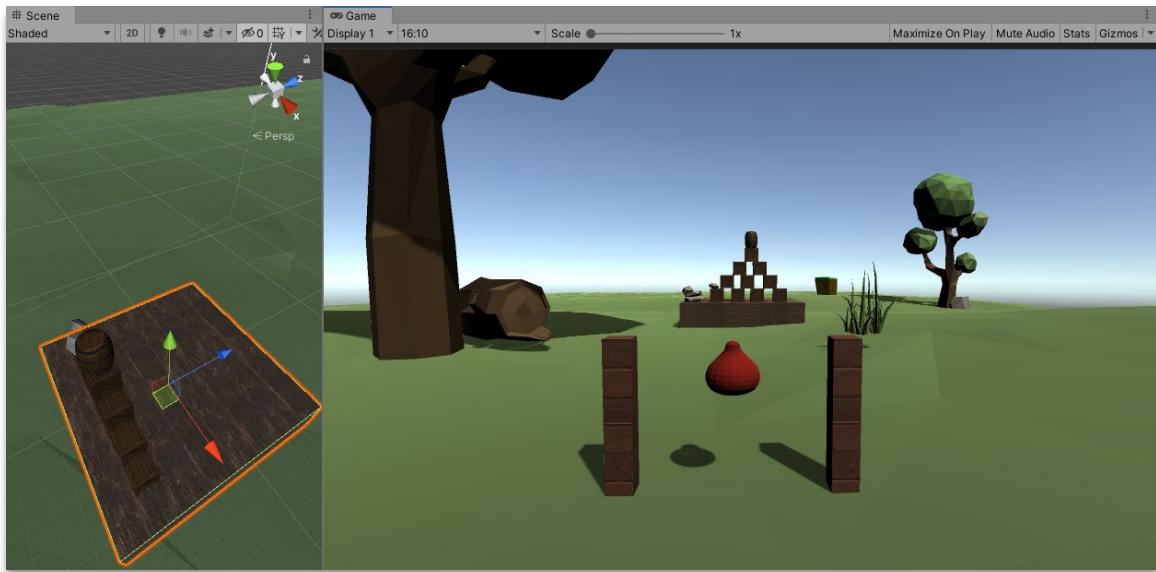
Search for **assets** or make your own and place them on your platform **object**. Create **empty objects** named “targets” and “collectables” to help you organize your scene.

Make sure your models have **colliders** and **rigidbodies**. Make sure that the **colliders** match the shape and the size of the **object**. Experiment by giving different objects different **masses**.

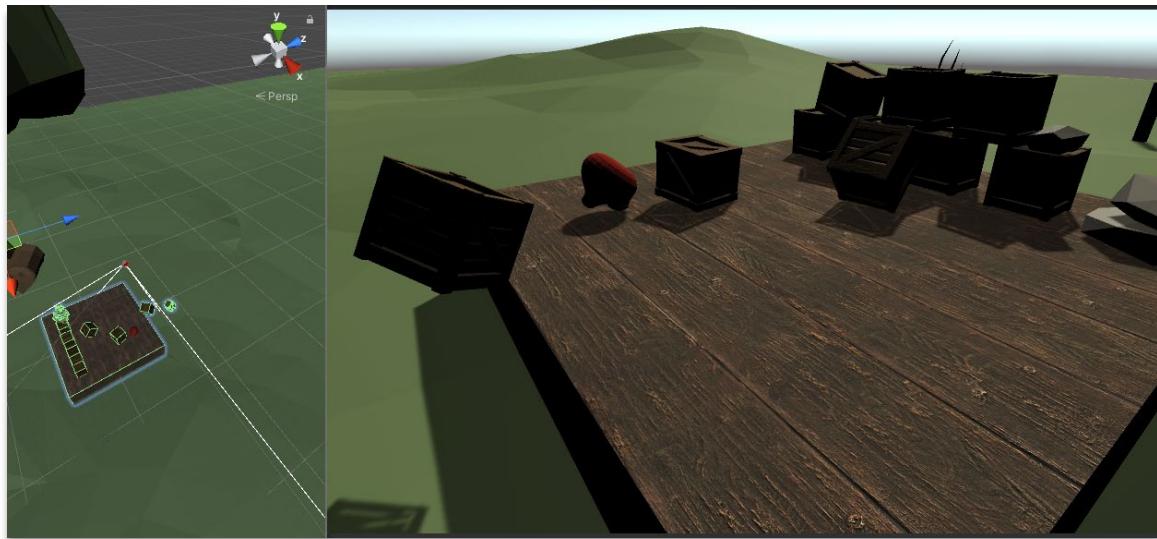
If any of your **objects** already have a **Mesh Collider**, make sure that **Convex** is enabled.



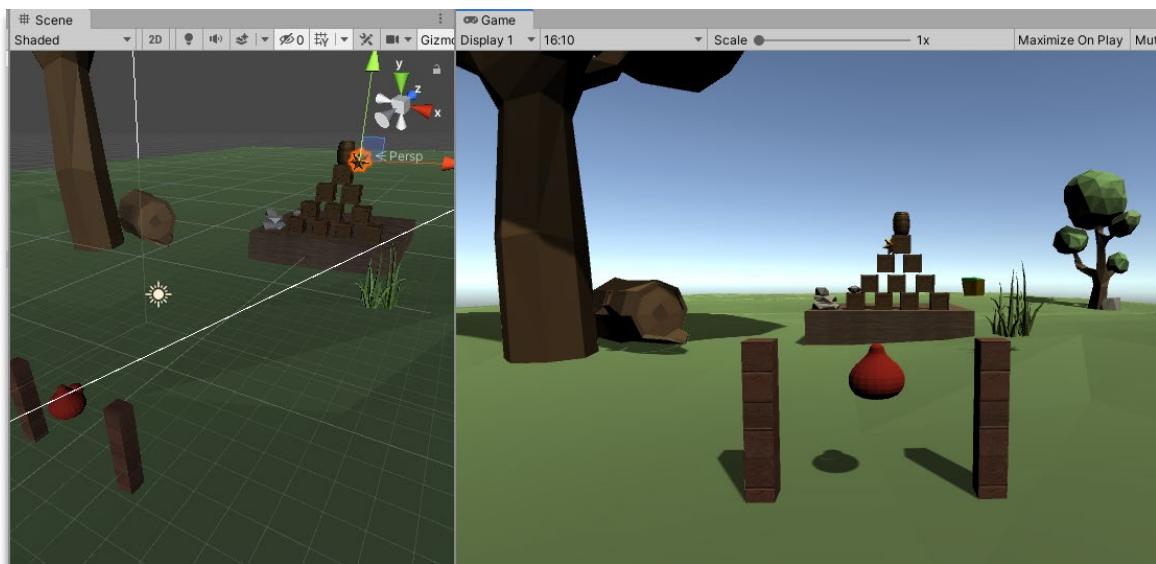
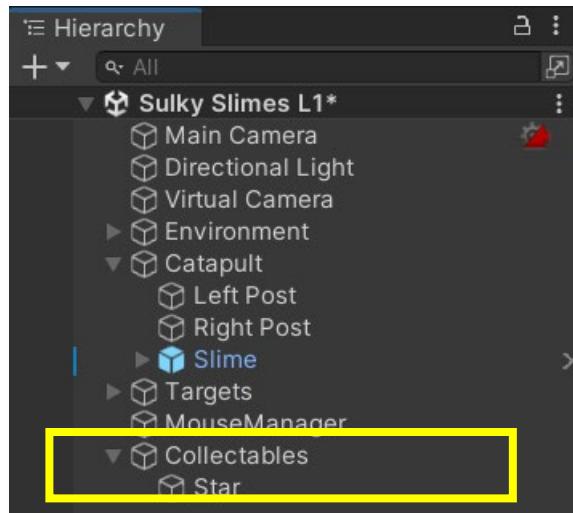
**34** Place your objects inside the **Targets** game object.



**35** Playtest your game and make sure that the slime can collide and interact with all the objects that you placed.



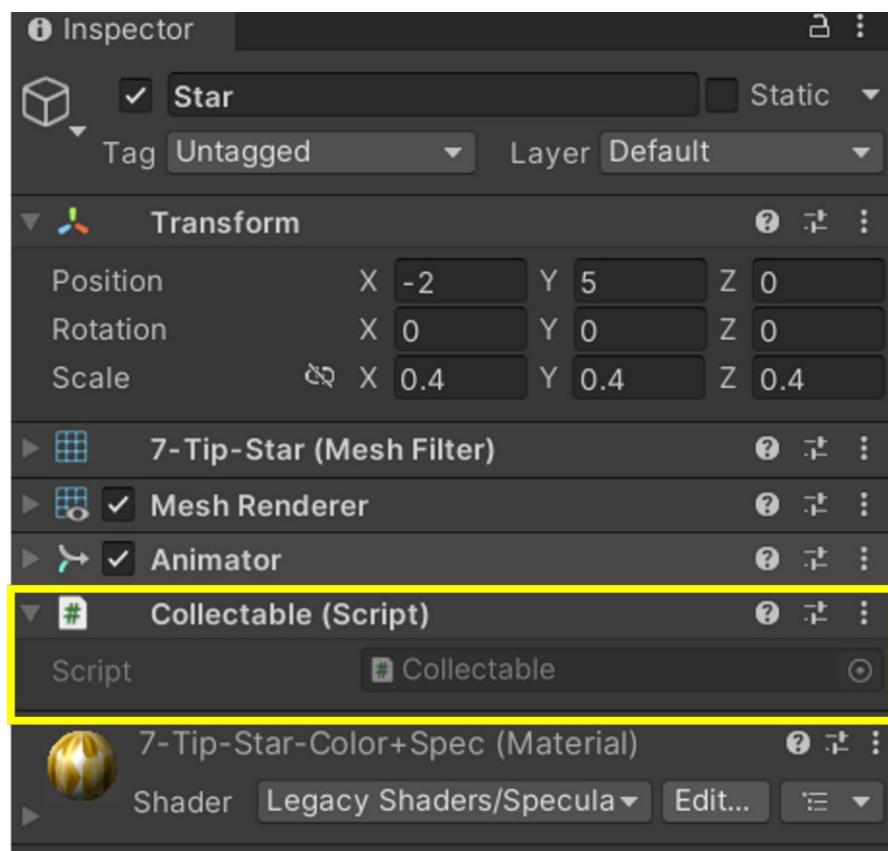
**36** Find an asset for your player to collect and place it behind the obstacles in the scene. Make it a child of the **Collectables** object.



**37** Based on your **Ninja Planning Document**, decide what challenge you want to present to the player. Are the **collectables** moving or changing size? Is there a time or attempt limit?

We will program a moving object and a lives system. Let's first work on making our star move so the player needs to aim their slime carefully.

Create a new **script** in the **Asset** panel, name it **Collectable**, and add it to the collectable **object**.



### Making It Your Own

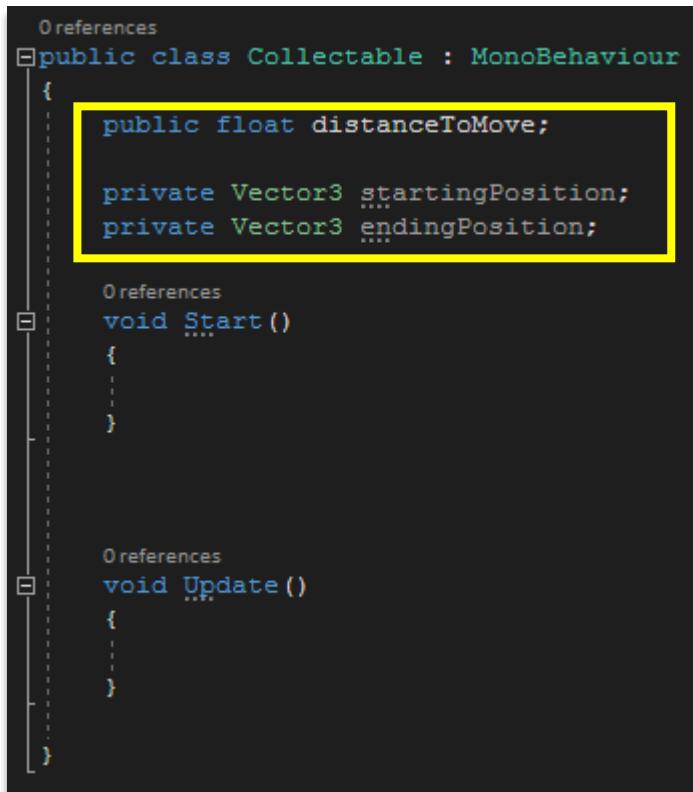
In our example game, we want the player to collect the object, but your game's goal might be different! Remember to use your Ninja Planning Document to help you make your game unique!

---

## 38 Open the Collectable script. We will program **movement** code like **Robomania** and **Gravity Trails**.

In Robomania, we knew where the edges of the screen were, and we changed the enemy's direction based on that information. Since we have an open 3D space in Sulky Slimes, we can calculate how far to move the collectable in its **Start function**.

In the Collectable **script**, add a **public float variable** named **distanceToMove**. We also need two variables to store the starting and ending positions. Since we are calculating the **values** inside the **script**, they can be **private**.



```
0 references
public class Collectable : MonoBehaviour
{
    public float distanceToMove;

    private Vector3 startingPosition;
    private Vector3 endingPosition;

    0 references
    void Start()
    {
    }

    0 references
    void Update()
    {
    }
}
```

**39** In the **Start function** we need to store the collectable's original **position** and set its ending **position** based on the **value** of **distanceToMove**.



Program the two lines of code described above. What direction do you want your collectable to move in?

Hint: Use the **value** of **transform.position** when you set the **values** of **startingPosition** and **endingPosition**.

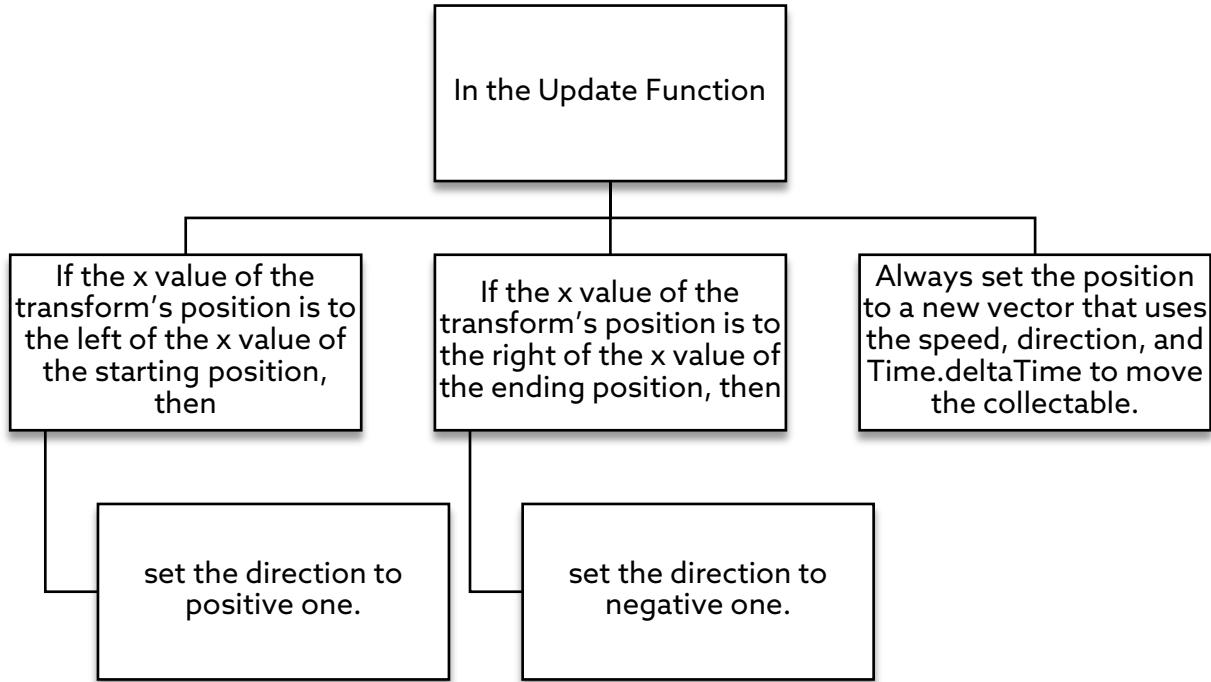
**40** If we **playtest** the game now, nothing will happen because we haven't written our **update function** yet.

We need to check the current **x position** of our collectable and compare it to our starting and ending **x positions**. We then need to change its direction based on if it has gone to the right of the ending position or to the left of the starting position. Then we need to move the collectable using a speed, a direction, and **Time.deltaTime**.

Create two **public variables** that store the direction and speed. You can change the speed **value** based on your **playtesting**. Your direction **variable** should be equal to either **1f** or **-1f** to move the collectable left or right, forward or backwards, or up or down.

```
public float distanceToMove;  
  
private Vector3 startingPosition;  
private Vector3 endingPosition;  
  
public float speed = 0.1f;  
public float direction = -1f;
```

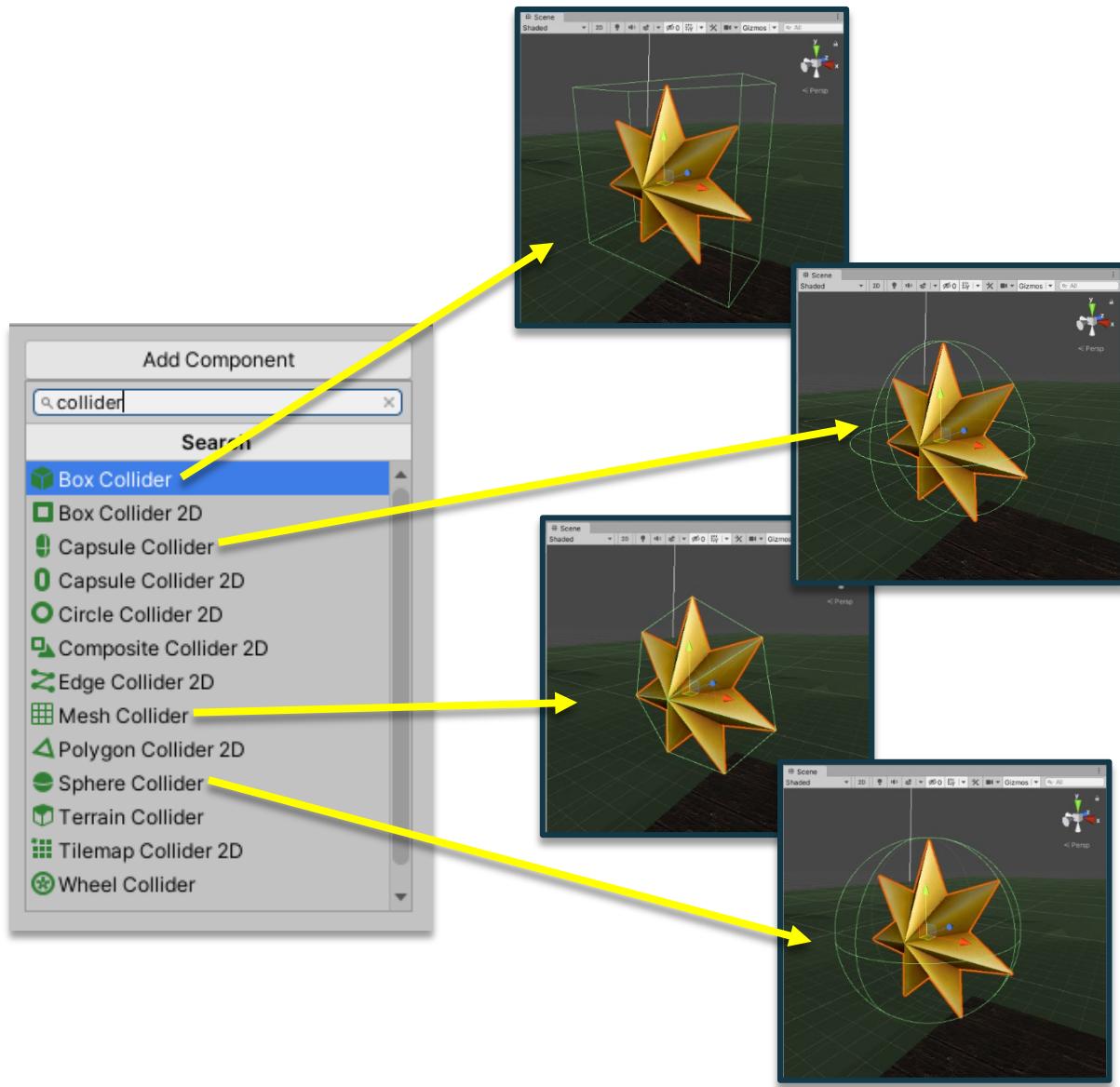
**41** Use the following pseudocode to help you write your own Update function.  
Make sure you create variables that store the direction and speed.



Use the pseudocode to help you write the collectable's Update function.

**42** Playtest your game. Adjust the **speed**, **distance**, and **direction** to fit your game design.

**43** We can now program the logic that lets our player collect the collectable object.

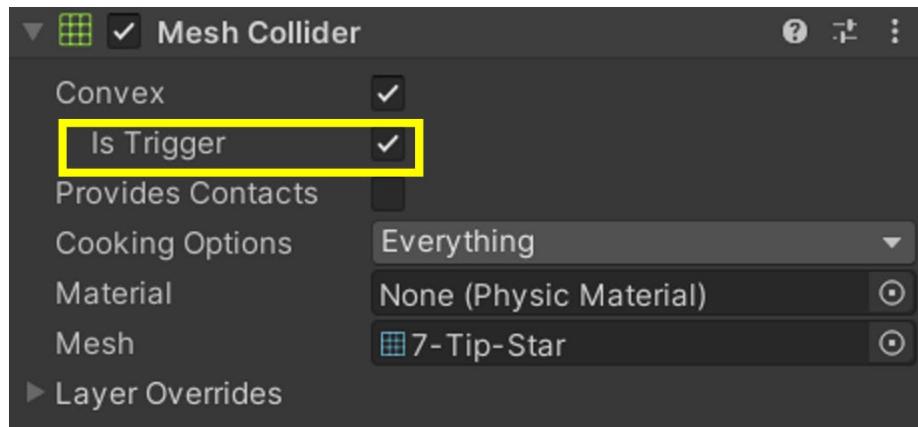


If your collectable does not already have a **collider component**, add one that makes sense for the shape of your object. Make sure you do not pick a **2D collider!**

**44** Unity does its best to set the size and position of **colliders** automatically. A **convex mesh collider** will most closely mold to the object's shape.

All the **colliders** use the same **functions** to detect when **collision** with another object starts (**OnCollisionEnter**), continues (**OnCollisionStay**), and stops (**OnCollisionExit**).

If we want to check **collision** without using **physics**, if the collider is a trigger then we can use **OnTriggerEnter**, **OnTriggerStay**, and **OnTriggerExit**.



### Sensei Stop

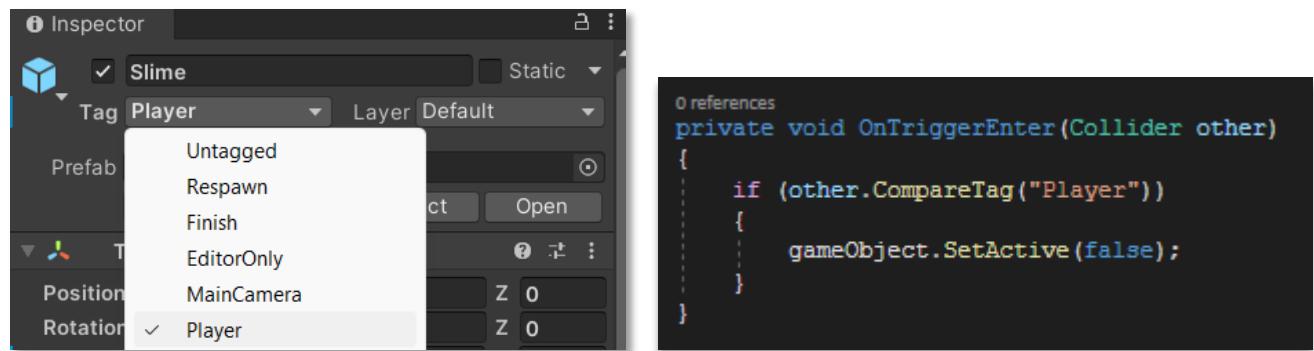
Use the **OnTriggerEnter** function to disable the collectable from the scene when the slime collides with it.

Hint: Use the **gameObject's SetActive** function.

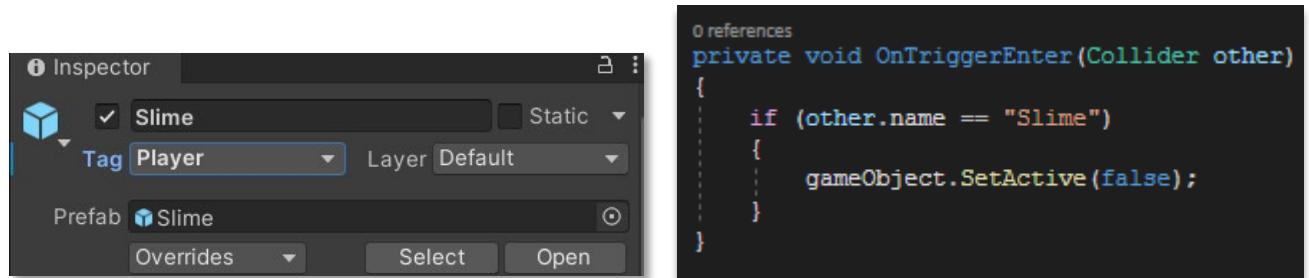
**45** Playtest your game. Can any **object** destroy the collectable? Right now, we are checking to see if any other **object collided** with our collectable. We need to make sure that we destroy it only if the slime touches it and not any of the obstacles.

Look back at previous games and decide how you want to check to see if the slime is **colliding** with the collectable. You can set up **tags** or use the **other object's name**.

### Tags



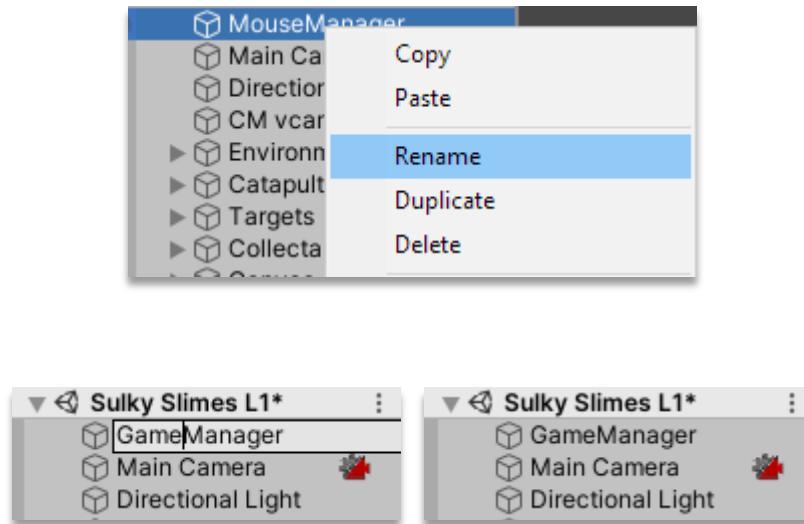
### Name



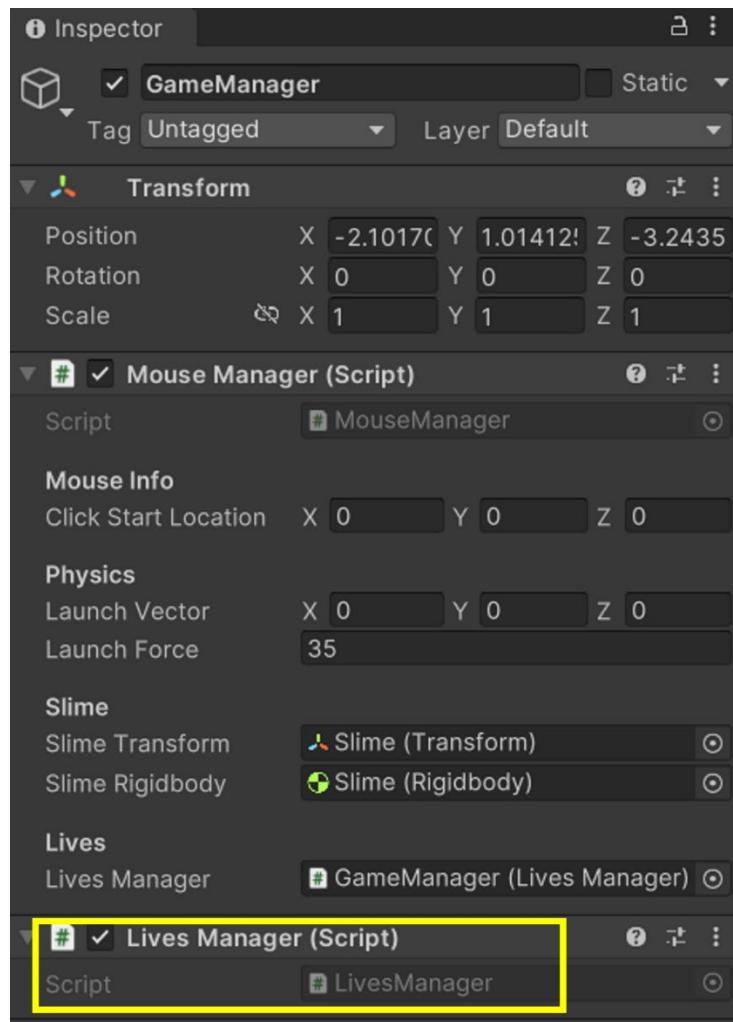
## A Slime's Life

**46** Now that the player has a goal, we need to program our lives system so they don't have infinite launches! As a game designer, you need to decide how you want to challenge your player. Providing a player with a clear goal with restrictions will give them a sense of accomplishment when they win!

Let's first program the **logic** and then work on the **UI**. We have a game object named **MouseManager** that right now has only one job – managing the player's interaction with the slime. We can create another new **game object** or repurpose this one to be a **GameManager**! Remember that we used a **GameManager** in Codey Raceway.



**47** Create a new **script** in the **Assets** tab, name it **LivesManager**, and attach it to the **GameManager** object.



**48** Before we program the script, we need to think about what it needs to do. Make a flowchart or diagram to help you organize your ideas!

### Sensei Stop

Plan out all the jobs that the LivesManager has. It needs to know how many lives the player has. What else? You can draw a flowchart or diagram to help you organize your thoughts.

---

## 49 Open the LivesManager script and create two variables.

Create a **public int** variable named **lives**. This number will represent how many lives the player has.

Create a **public GameObject[]** variable named **hearts**. This is an **array** that will contain **images** that will represent our lives.

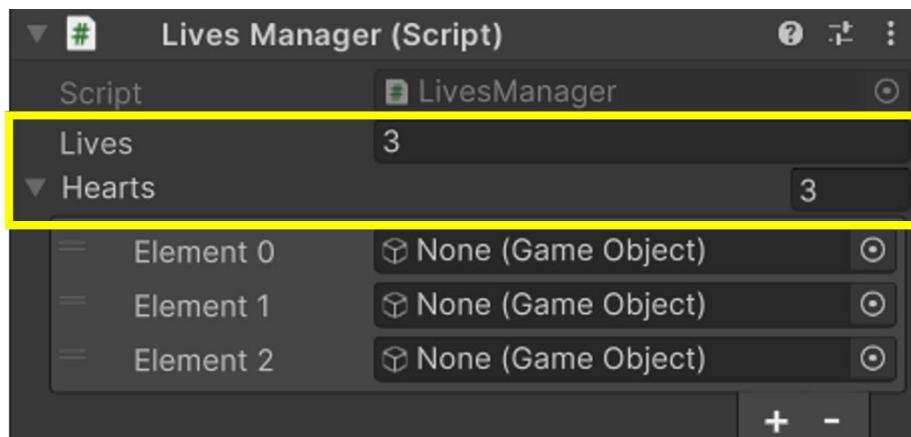
```
1 reference
public class LivesManager : MonoBehaviour
{
    public int lives;
    public GameObject[] hearts;

    0 references
    void Start()
    {
    }

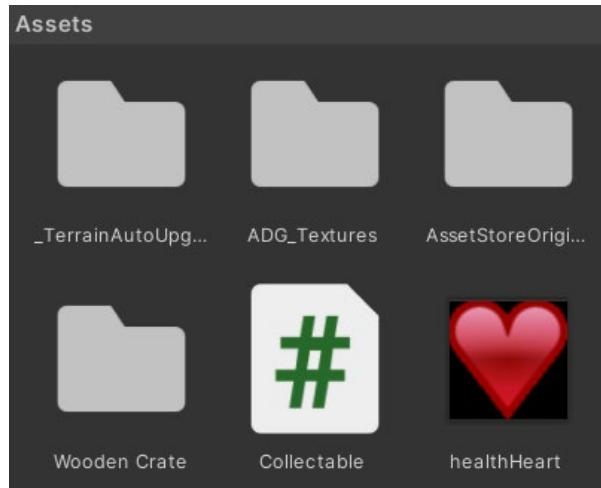
    0 references
    void Update()
    {
    }
}
```

---

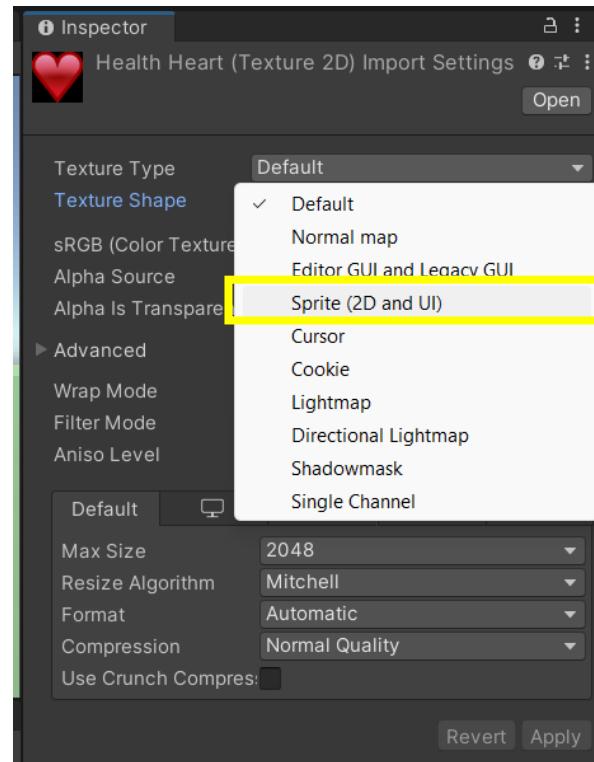
## 50 Save the script and look at the **inspector**. Set the **value of Lives** to **3** and the **size of Hearts** to **3**.



**51** We now need three hearts to fill our three empty **elements**. Find your own image or use the **healthHeart.png** file provided. Place the image in your Assets folder.



**52** Open the **image asset** in the **Inspector**. Change its **Texture Type** to **Sprite (2D and UI)**.



**53** Create a **public void function** named **RemoveLife**. Inside this function, subtract one from the **lives variable** and **print** a message to the **console**.

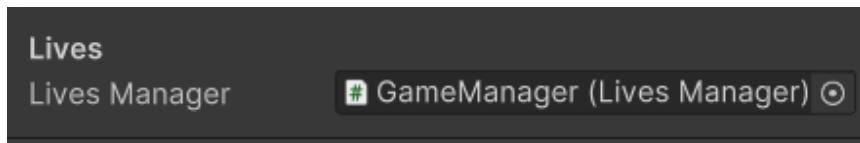
```
public class LivesManager : MonoBehaviour
{
    public int lives;
    public GameObject[] hearts;

    References
    public void RemoveLife()
    {
        lives -= 1;
        print("You lost a life! Lives: " + lives);
    }
}
```

**54** We need to connect the **LivesManager** with the **MouseManager** so we can call the **RemoveLife function** when the player resets the slime.

Open the **MouseManager script** and add a **public LivesManager variable** named **livesManager**. Since we declared our **variable** to be the **LivesManager** type, the **Inspector** will only give us one option. **Unity** knows to only show us the types of **objects**, **scripts**, and **variables** that we want!

```
[Header("Lives")]
public LivesManager livesManager;
```



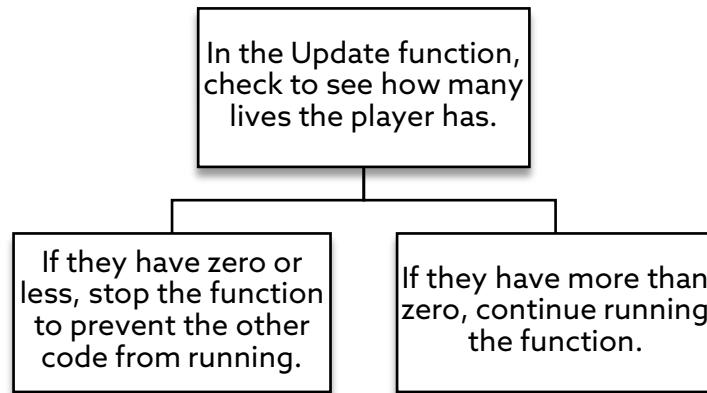
**55** Now that we have a reference to our **LivesManager** inside our **MouseManager**, we need to call our **RemoveLife** function.

 **Sensei Stop**

Where in the Update function should we call the RemoveLife function?  
Call the livesManager's RemoveLife function and playtest your game.  
Talk with your Sensei about where you placed it in your code.

Hint: Look out for **console** messages to see exactly when your game runs the **RemoveLife function**.

**56** We now need to make sure the game ends if the player runs out of lives. We can use **pseudocode** to help us add to our **MouseManager's Update function**.



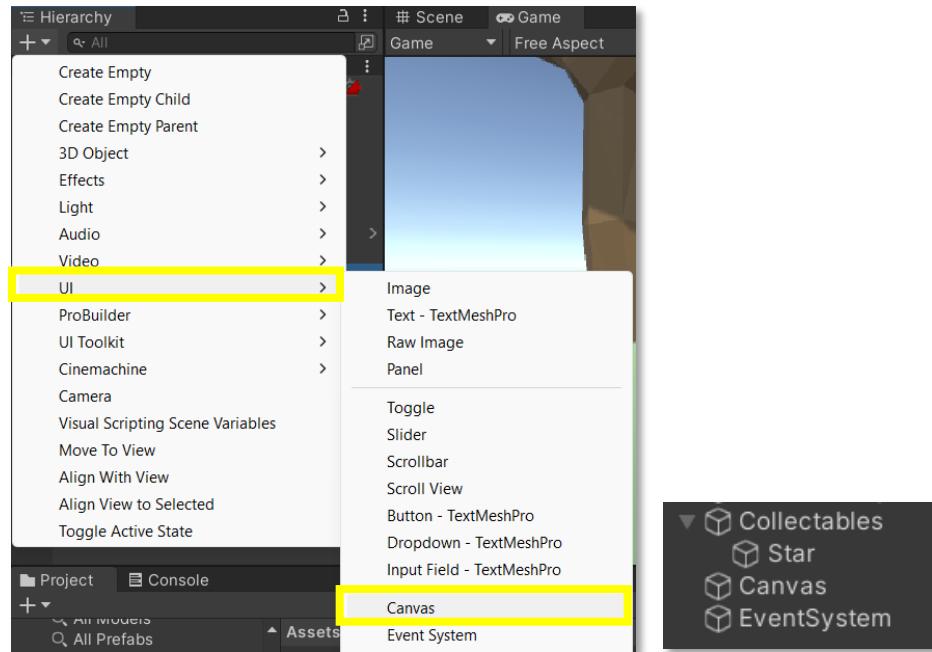
```
0 references
void Update()
{
    if (livesManager.lives < 0)
    {
        return;
    }

    if (Input.GetMouseButton(0))
    {
        clickStartLocation = Input.mousePosition;
    }

    if (Input.GetMouseButton(0))
    {
```

**57** Playtest the game to see what happens after you run out of lives. The player is not able to interact with the game!

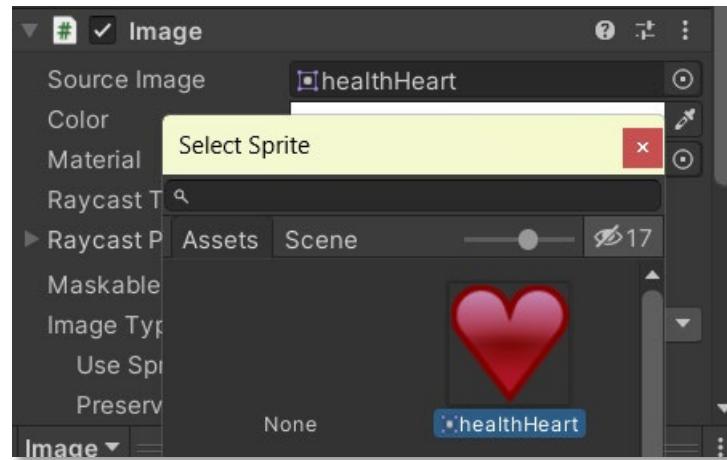
We now will let the player know how many lives they have left with a **UI**. First, add a **Canvas** object to your **scene**. This will also automatically add an **EventSystem** to the **hierarchy**.



**58** Add three **UI Images** objects to the **Canvas** object.



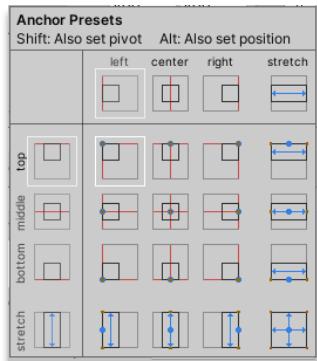
**59** Change the three **Image** object's **Source Image** property to your chosen image.



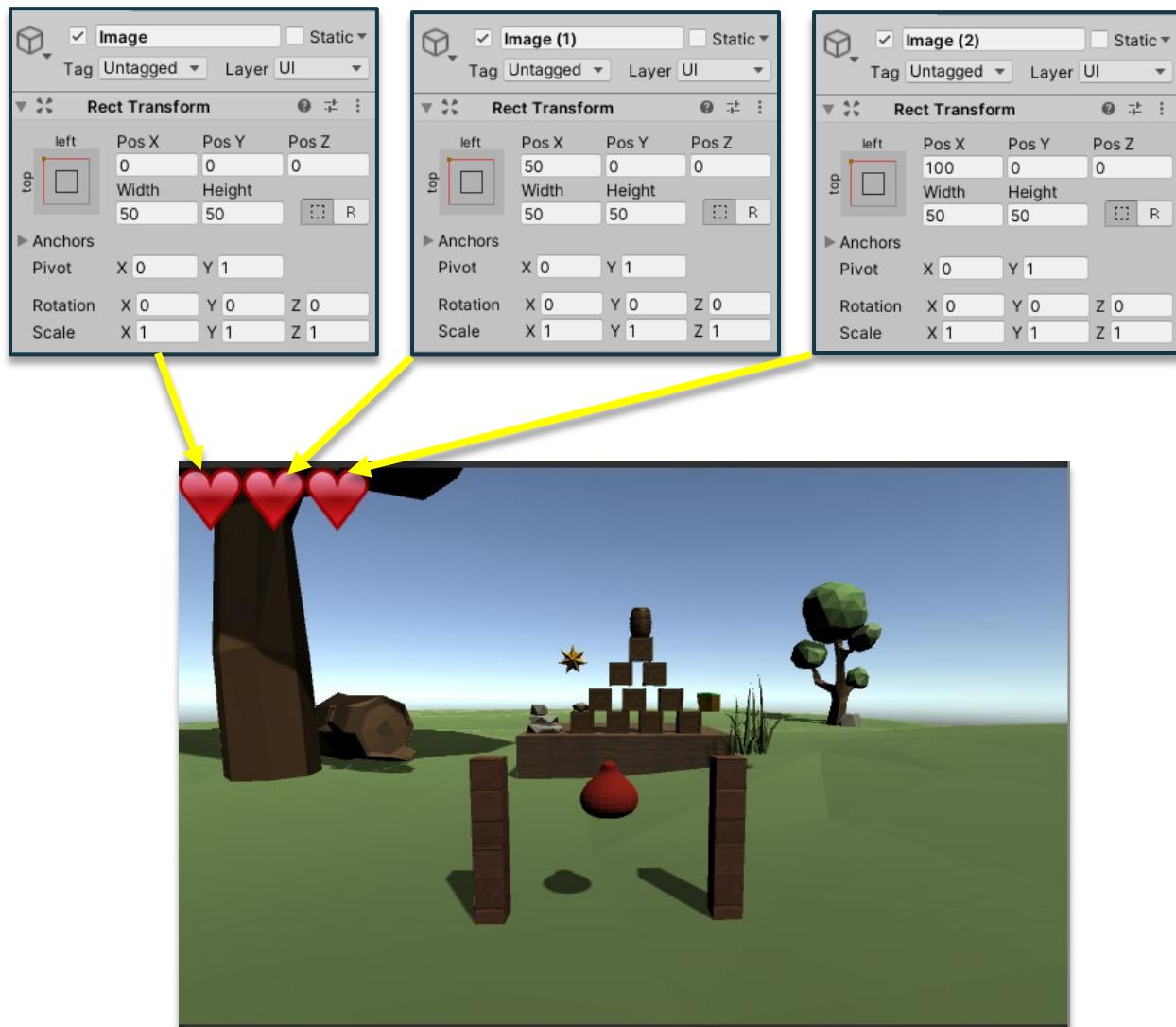
You should be able to see your **UI** in your **game** window.



**60** We now need to reposition our three images. Look at its Rect Transform in the Inspector. Use the Anchor settings to place it somewhere on your canvas. Hold shift and alt when you click the location to pin it in place.

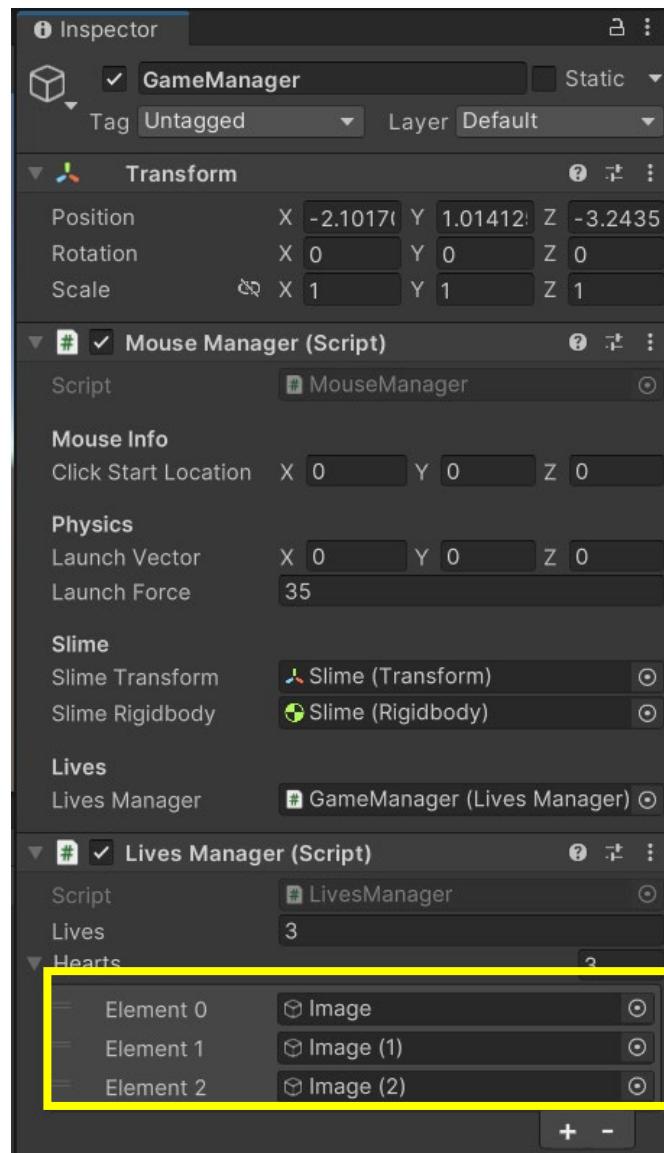


Adjust the width, height, and positions to align your three images. Make sure you keep them in the correct order!



**61** With our UI set up, we can write the code to update it with how many lives the player has left.

First, let's connect our three images to the LivesManager script. Place the images in the correct Element of the Hearts property.



**62** Since our three heart images are active when the game starts, we just need to disable a heart each time a player loses a life.

When the player loses a life, disable the heart object in the hearts array based on the current value of lives.

```
public void RemoveLife()
{
    lives -= 1;
    hearts[lives].SetActive(false);
}
```

**63** When the player runs out of lives, we want to reload the scene to let them try again.

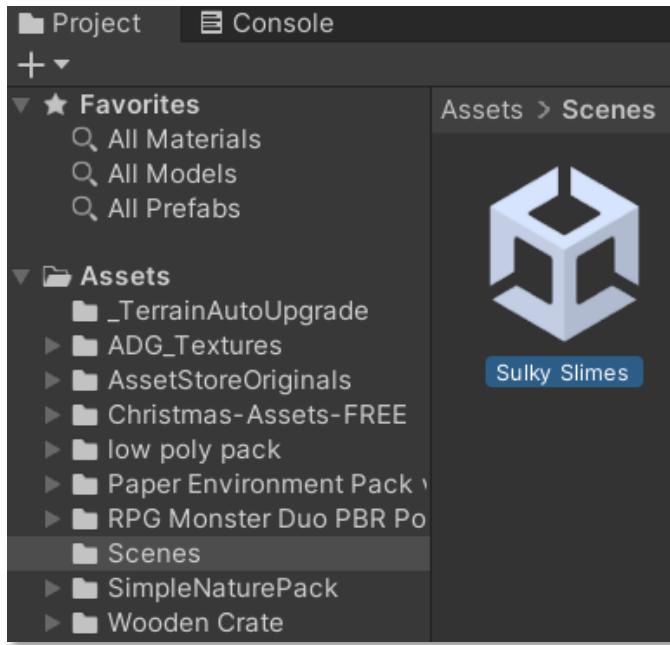
#### Sensei Stop

Use Brown Belt's World of Color to use Unity's Scene Manger to reload the game when the user runs out of lines. Start with Step 5.

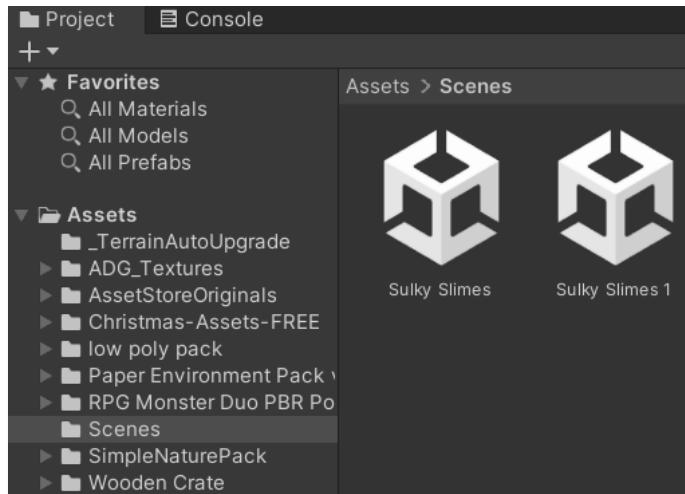
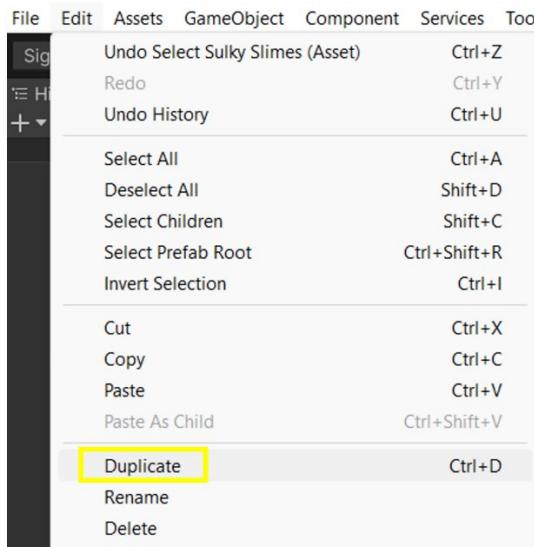
Playtest your game to make sure your game reloads when the lives run out! To speed up the playtest, you can right click however many times you have lives.

## A Change of Scenery

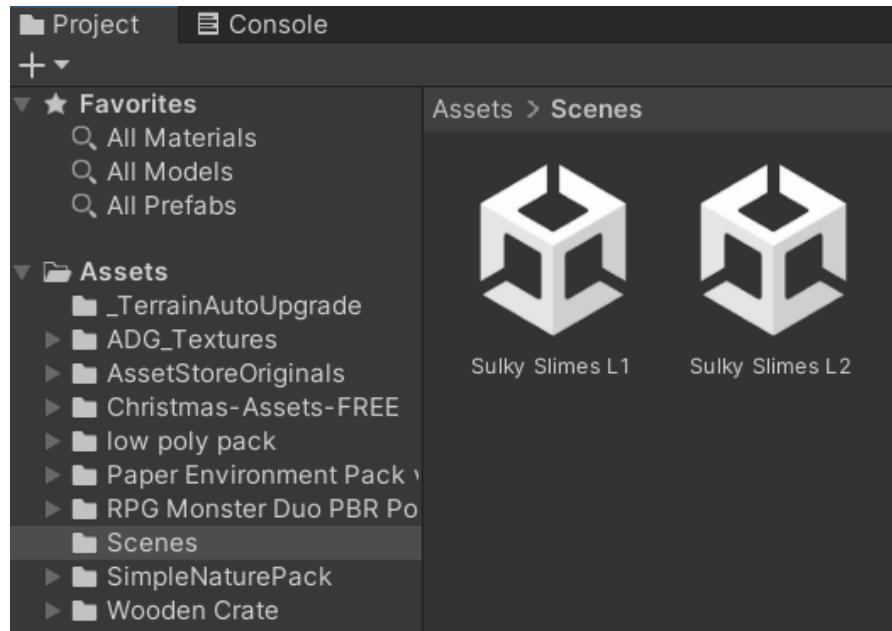
**64** As a reward for getting the collectable, we can load a second level for the player. Save your current **scene**. Instead of starting from scratch, we can use our first scene as a starting point. Select the scene in the **Project** window.



Press **Control + D** or go to **Edit -> Duplicate**



**65** Rename your **scenes** so it is easier to tell them apart.

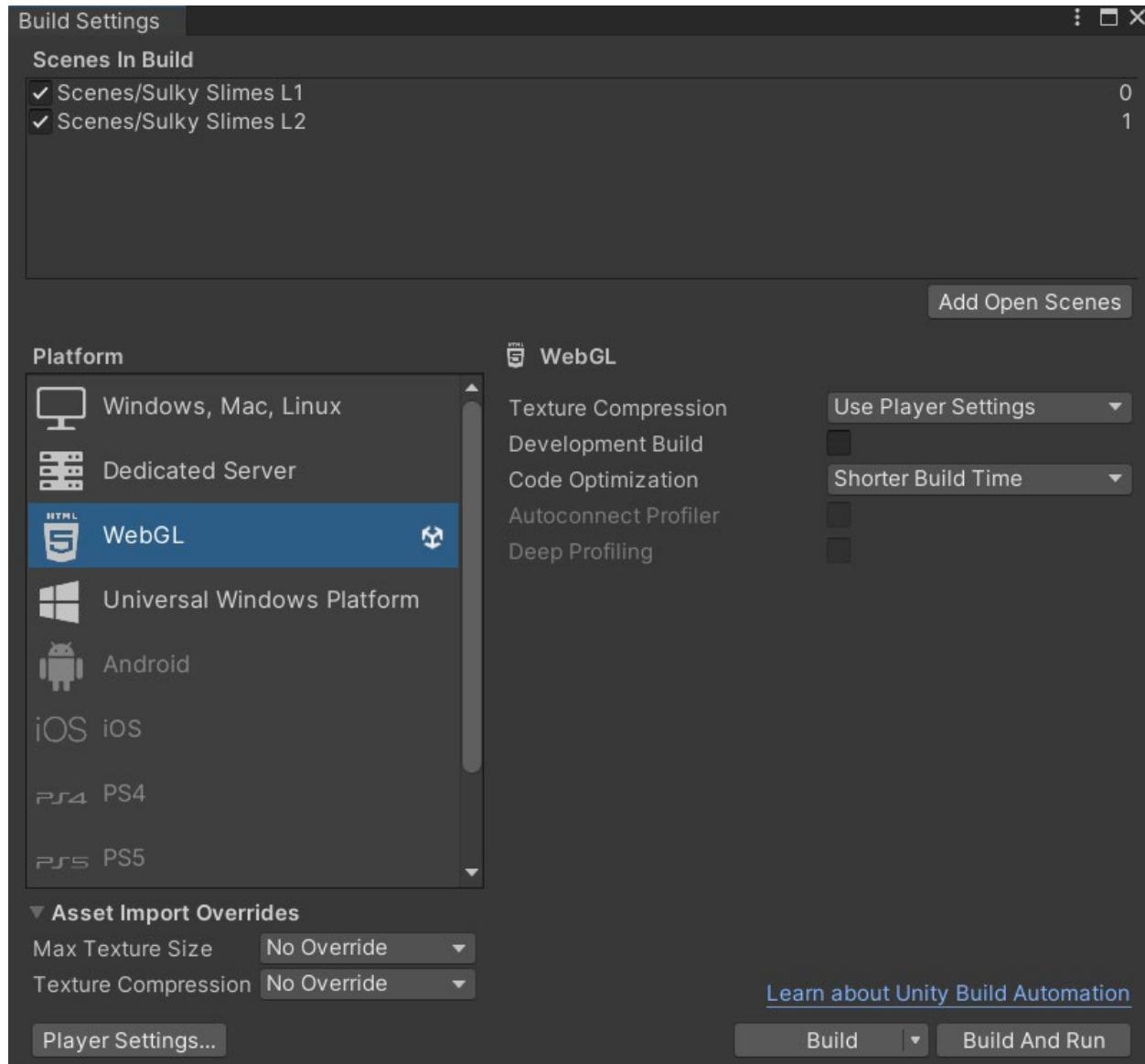


**66** Open the level 2 **scene** and make it unique. Change the **environment**, the **collectables**, or the **targets**. Be creative, like recreating a giant version of your Code Ninjas Dojo!



**67** Playtest your game and make sure everything is working properly. Since we changed only the **models**, all our **logic** still works. We just need to connect our **scenes** so we can dynamically load the next level!

First, go to **File -> Build Settings** and add your new **scene** to the “Scenes in Build” menu.



- 
- 68** When the player collects our collectable, we want to load the next scene. Open the **Collectable script** and add using **UnityEngine.SceneManagement** to the top.

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

- 
- 69** We also need to know which **scene** to load based on the number assigned in the "Scenes In Build" menu.

In the Collectable **script**, create a **public int variable** named **sceneNumber**. Now that we have a few different **public variables**, add **Headers** to help you stay organized.

```
[Header("Movement Values")]
public float distanceToMove;

private Vector3 startingPosition;
private Vector3 endingPosition;

public float speed = 0.1f;
public float direction = -1f;

[Header("Scene to Load")]
public int sceneNumber;
```

- 
- 70** In the Collectable **script**, create a new **private void function** named **LoadNextScene** that takes no **parameters**.

```
0 references
private void LoadNextScene()
{
}
```

**71** Use the **SceneManager's LoadScene function** to load the **scene** associated with the **sceneNumber variable**.

```
0 references
private void LoadNextScene()
{
    SceneManager.LoadScene (sceneNumber);
}
```

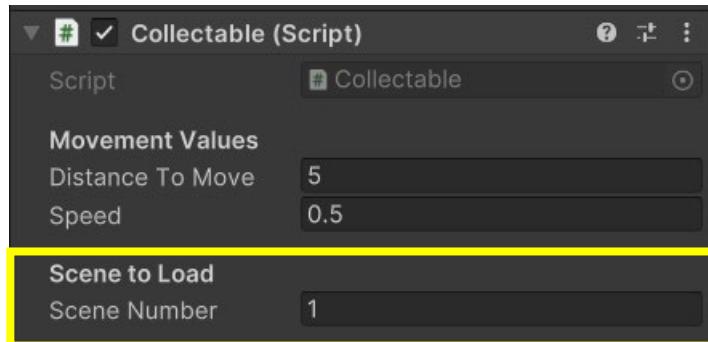
**72** We want to give the player some time to enjoy their victory.

In the Collectable **script's OnTriggerEnter function, Invoke** the **LoadNextScene function** after 2 seconds.

```
0 references
private void OnTriggerEnter(Collider other)
{
    if (other.name == "Slime")
    {
        gameObject.SetActive(false);
        Invoke("LoadNextScene", 2f);
    }
}

0 references
private void LoadNextScene()
{
    SceneManager.LoadScene (sceneNumber);
}
```

**73** In the **Inspector**, give **Scene Number** a **value** of 1 to load the corresponding **scene**.



**74** **Playtest** your game and see what happens after you get the collectable. The player is transported to the second **scene**!

## Project Submission and Reflection

Now that you have a working game, take time to add your own personal touches to the project. What aspects from other games could you add?

Once you feel like you have a good product that represents your vision of the game, have a Sensei and at least one other Ninja playtest it. Use the Playtest Survey Planning Document for questions to ask them when they finish. Record their answers in your Ninja Planning Document.

Based on the results of the playtest and survey, make changes to your game. Once you are complete, share the updates with your Sensei and fill out the reflection section of your Ninja Planning Document.

Before you submit your game for grading, use the Sulk Slimes Project Requirements Checklist to make sure your game has all of the required features.



### Ninja Planning Document

Use your Ninja Planning Document to record feedback from Senseis and other Ninjas in your Center.



## Chef Codey

Your goal is to plan, program, and playtest a game where the player interacts with objects in the environment! You will use objects from the Unity Asset Store to create your own themed environment for Codey.



The game we will create together finds Codey managing a café. As you plan and program, feel free to create your own theme – farming, mining, or anything else you can think of!

## Plan and Design

In our café, Codey must prepare three dishes that are built out of smaller parts. For example, coffee requires bringing a mug to the coffee station, waiting for the coffee to brew, and then placing the filled mug on the counter. Codey can only hold one object at a time, so the player must plan how they prep their dishes to get the fastest time.



Overcooked 2 by Team17  
Built in Unity



Untitled Goose Game by House House  
Built in Unity

Your diagram must contain a starting location for Codey, six stations where Codey can interact with items, and a location to place completed dishes.



### Ninja Planning Document

Take at least 5 but no more than 10 minutes and complete your Ninja Planning Document – Designing a Scene

Take a look at the sample projects below for some inspiration!



## Project Setup

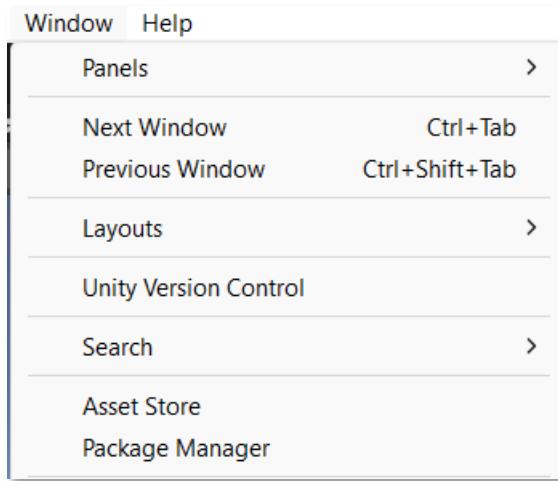
---

- 1 Start a new Unity Project and name it *YOUR INITIALS – Chef Codey*.  
Select **3D template**.

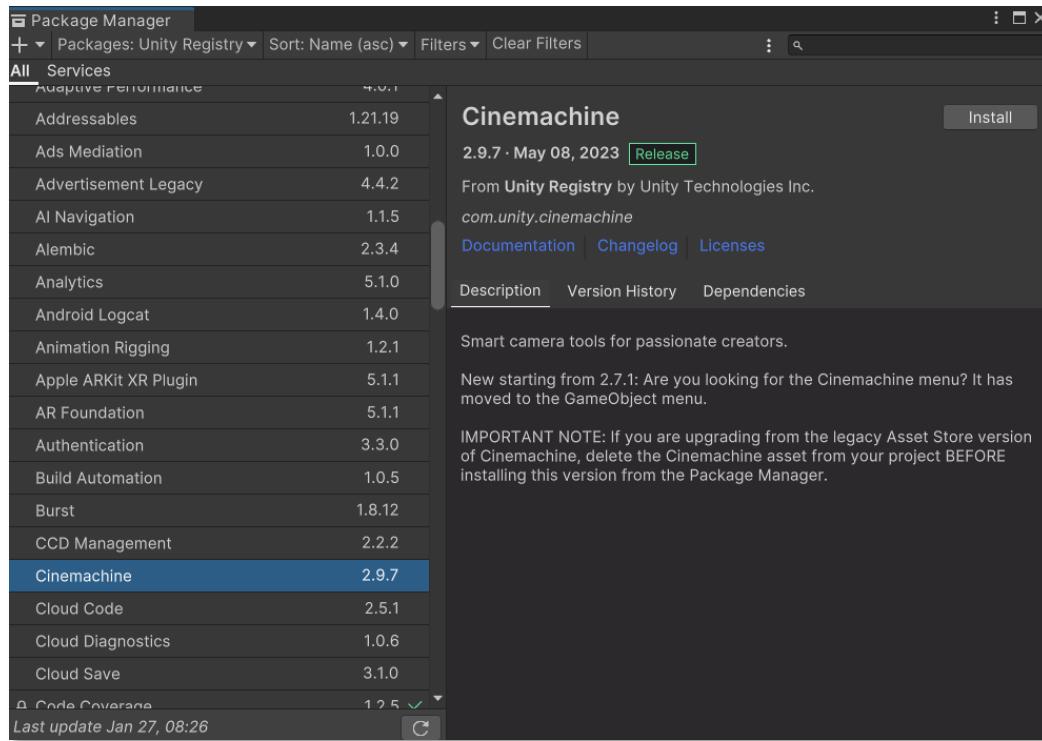
- 2 After it loads, rename the Sample Scene to Chef Codey.



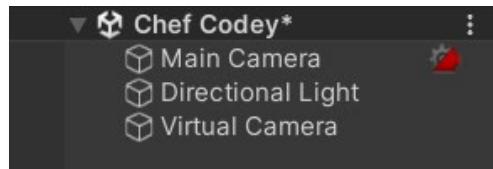
### 3 Open the Unity Package Manager from the Window Menu.



Find **Cinemachine** and click Install in the top right of the window.



- 
- 4** You should now see a **Virtual Camera** in your object **Hierarchy**. If not, go to **Cinemachine -> Create Virtual Camera**.

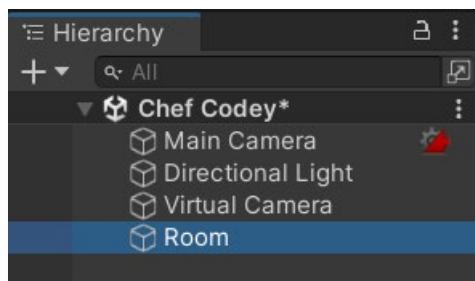


## A Room with a Ninja

5

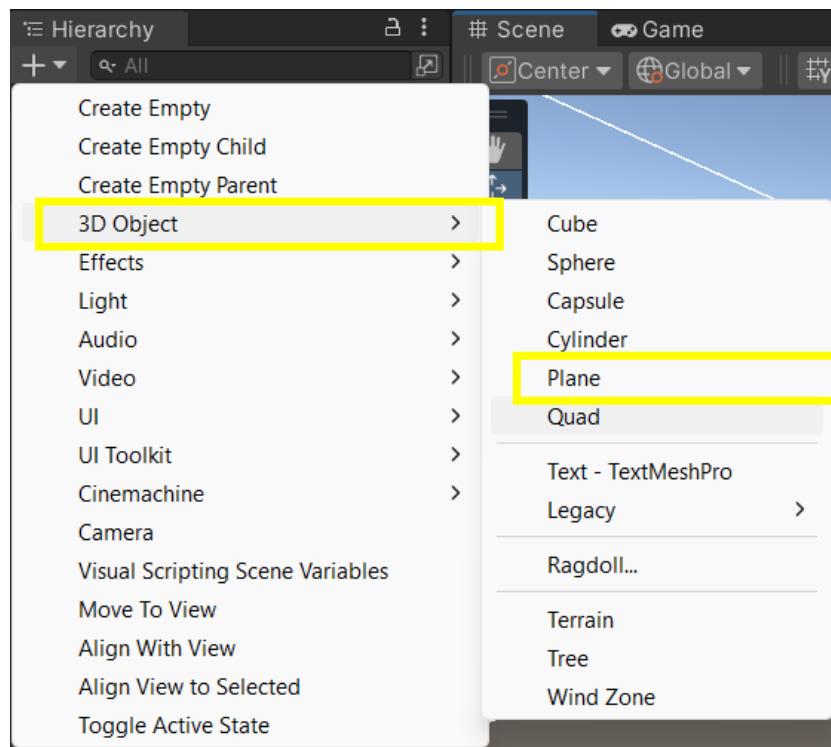
Use Code Ninjas **assets**, **assets** from the **Unity Asset Store**, or your own creations to build a basic room. If you need help getting started, you can follow the next few steps.

First, create an empty **Game Object** and rename it "room". We can create our room out of six **Plane objects**.

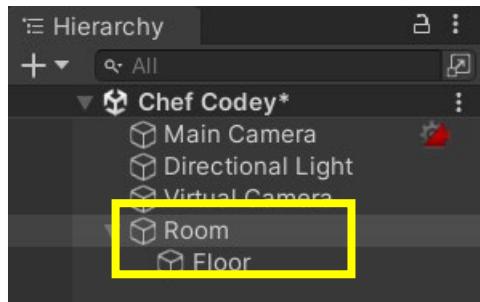


6

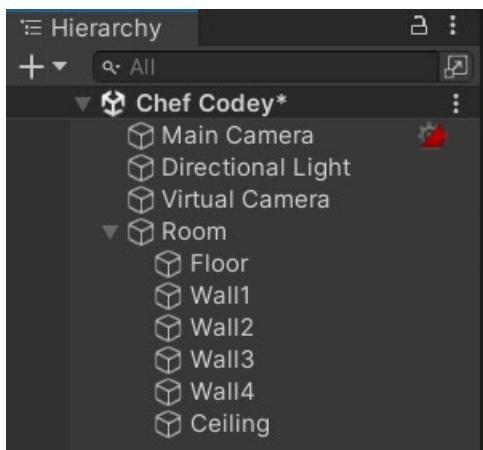
Create a new 3D Plane **game object**.



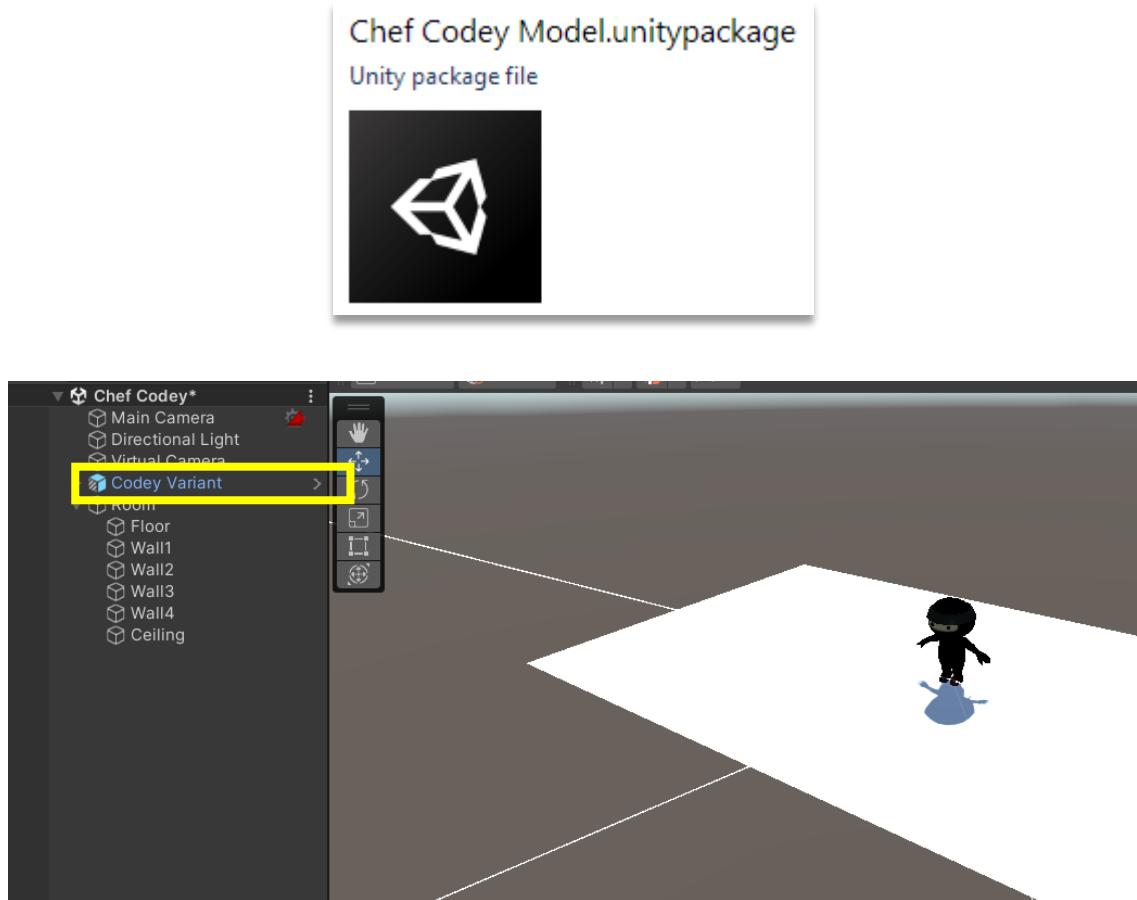
- 7** Rename the new **Plane object** “Floor” and place it inside of the Room **game object**.



- 8** Repeat this for the four walls and ceiling. You can create a new plane each time or duplicate your first plane five times.



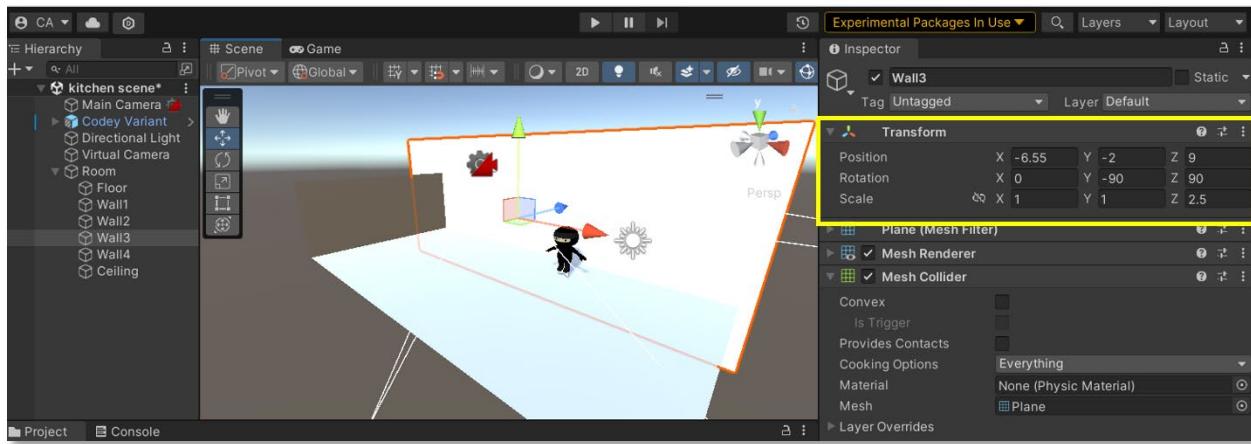
- 9** Import the **Codex asset** and place it in the scene to make sure your room is properly **scaled** to your character.



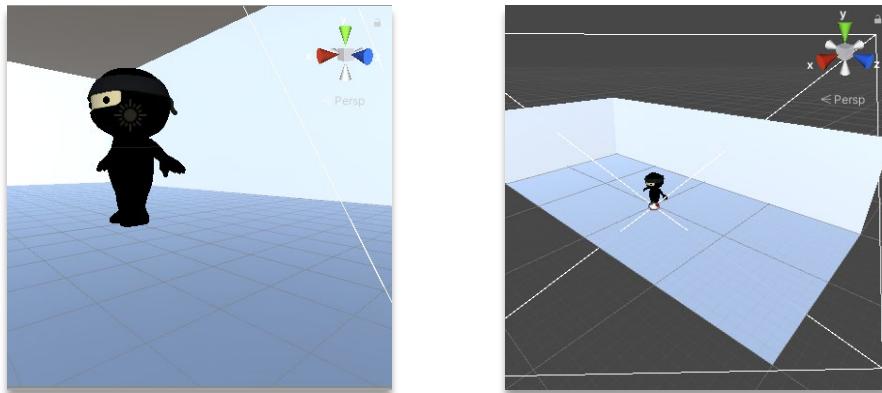
- 10** Each of the **planes** is positioned at the same **coordinate**. **Rotate**, **move**, and **scale** each of the planes into their proper positions. Use your **Ninja Planning Document** to help you build out your **scene**.

11

Did you notice that a **plane** is see-through on one side? Make sure you **rotate** the ceiling by 180 degrees in the x direction.



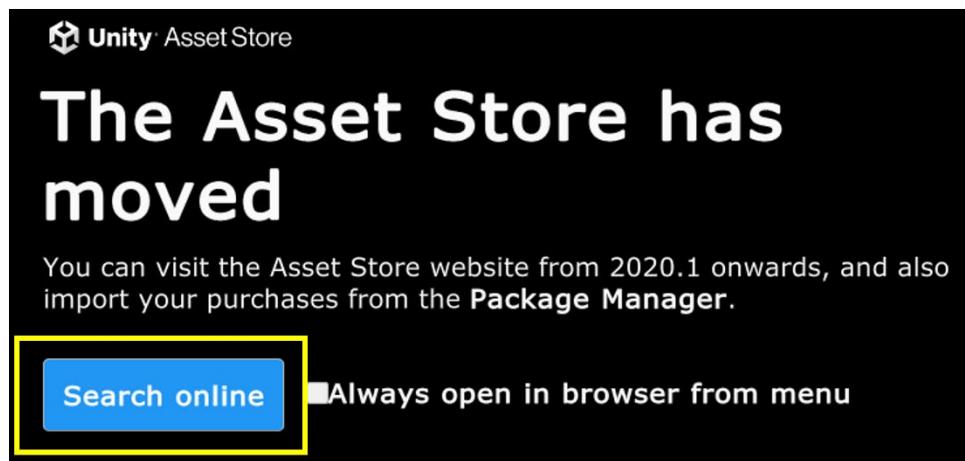
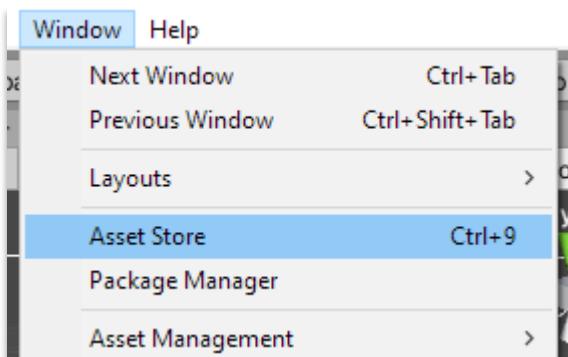
From Codey's perspective, it's an enclosed room. From the camera's perspective, it's a box with see-through walls!



12

We can use the **Unity Asset Store** to find some **textures** to place on these **planes** to liven up the environment.

Open the **Unity Asset Store** by clicking on **Asset Store** in the Window menu. Then click the **Search Online** button.



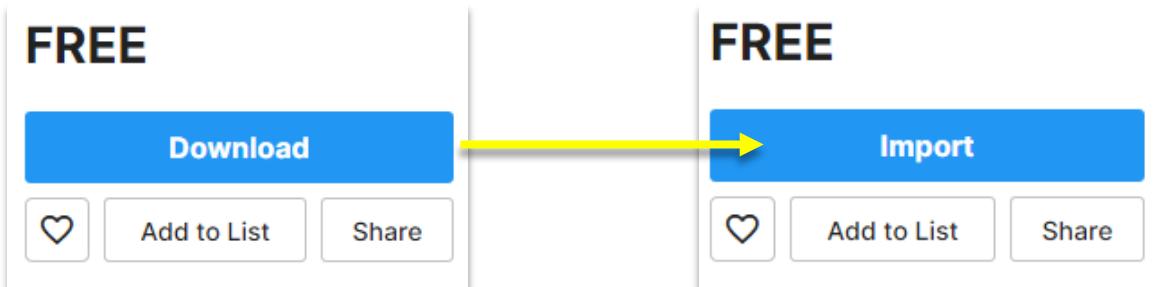
13

Search for “texture” and make sure you select “Free Assets” under **Pricing**.

A screenshot of the Unity Asset Store search results for 'texture'. On the left, there is a search bar with 'texture' and a 'Free Assets' filter applied. On the right, there is a 'Pricing' sidebar with a checked checkbox for 'Free Assets (452)'. This indicates that only free assets will be shown in the search results.

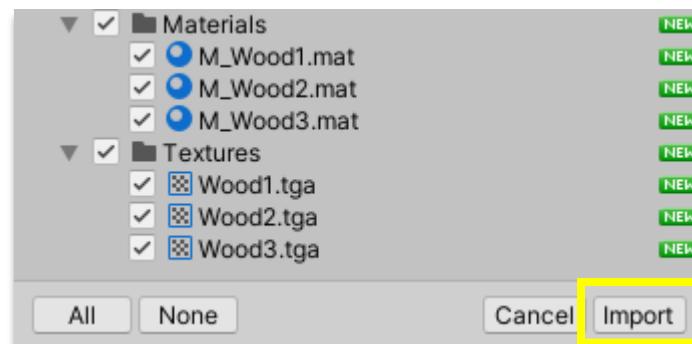
**14**

After you find an **asset** that you like, click **download**. Once it downloads, it will be in your **Unity** account. Click **Import** to put the **asset** from the **store** to your current **project**.



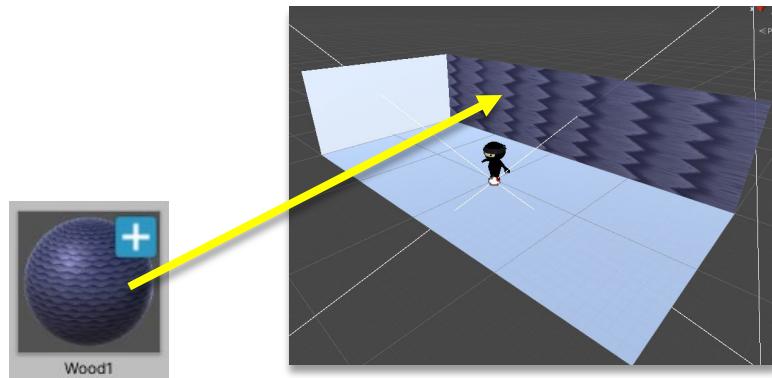
**15**

In the **Import Unity Package** window, click **Import**.



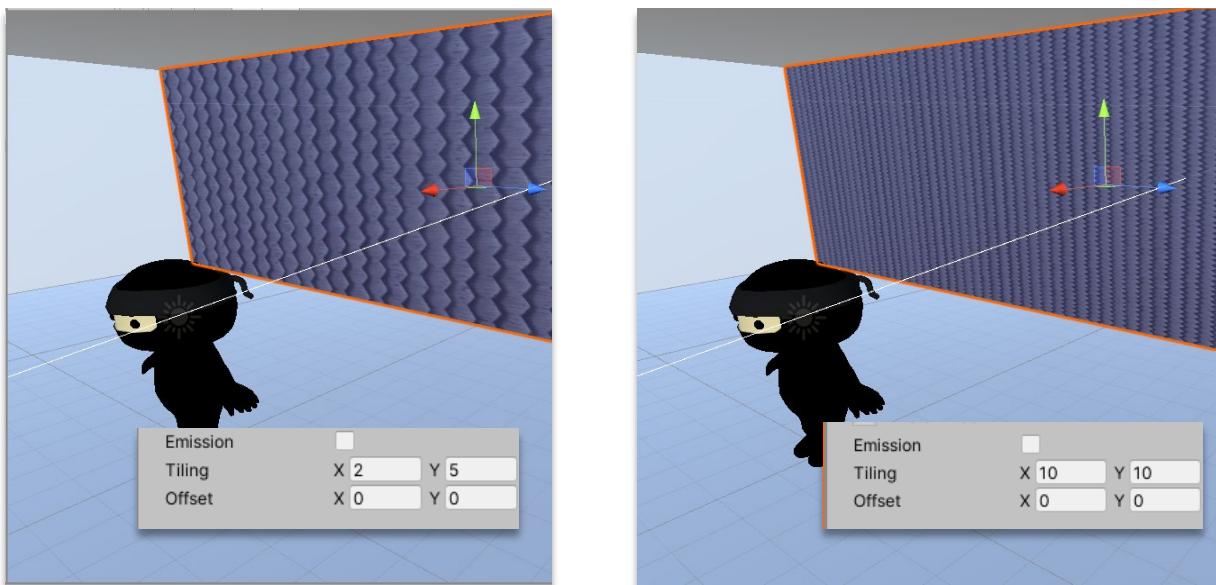
**16**

Look in your **Assets** folder in the **Project** tab for where the new **assets** are located. Drag your **texture asset** from the **Project** tab onto one of the **plane objects** in the **scene**.



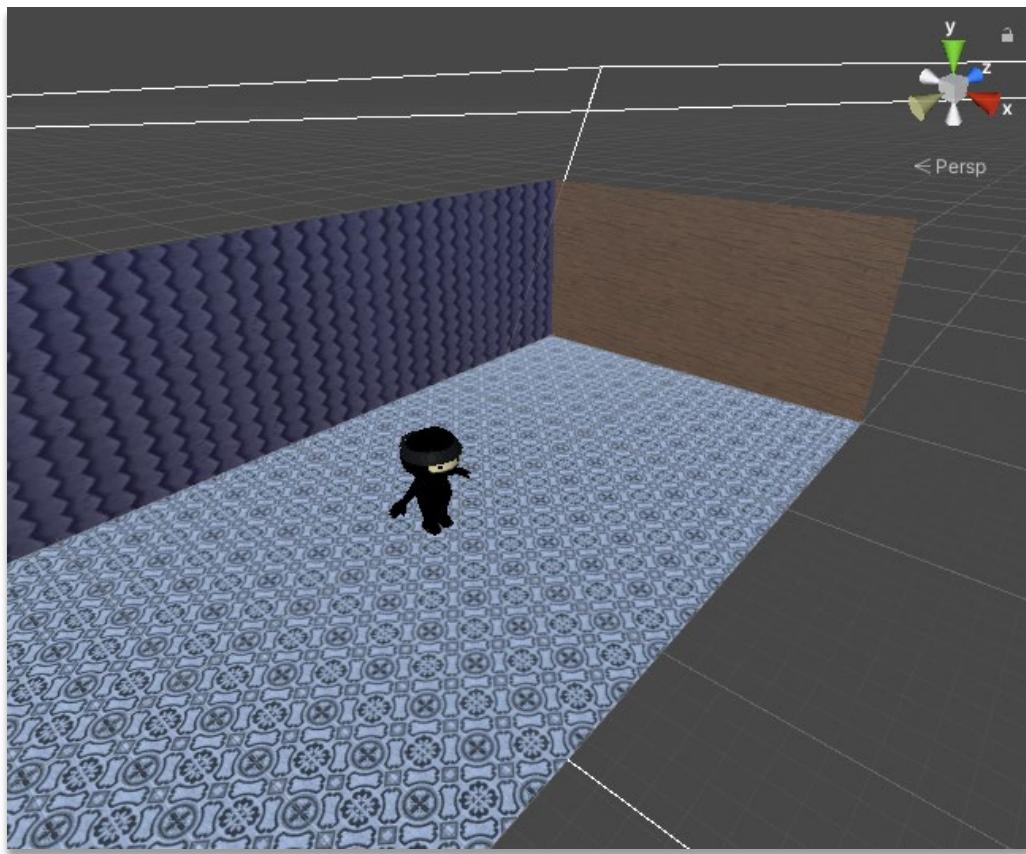
**17**

Select the **plane** and look at its **texture component**. Adjust the **Tiling X** and **Y** **parameters** to adjust how the **texture** is applied to the **object**.



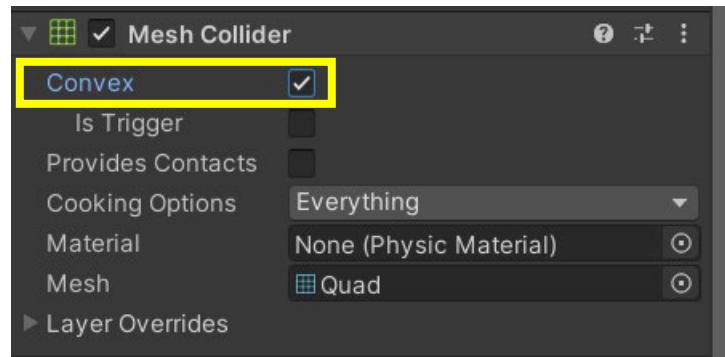
**18**

Repeat this process to design the other **planes** that make up your room.



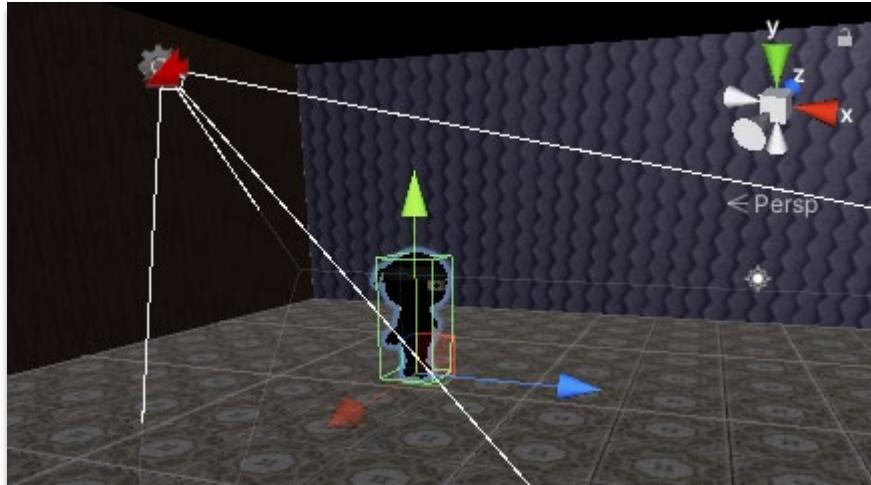
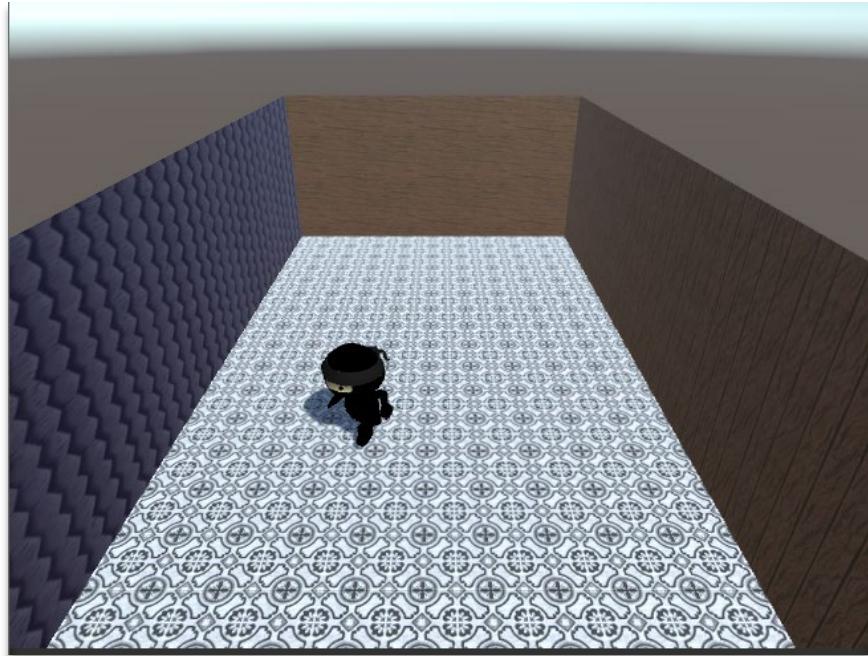
**19**

With a basic room complete, **playtest** your game. You might need to adjust the **scene** based on how you initially set it up. If Codey can run through your walls, make sure that your planes have **colliders** with the **Convex** property enabled.



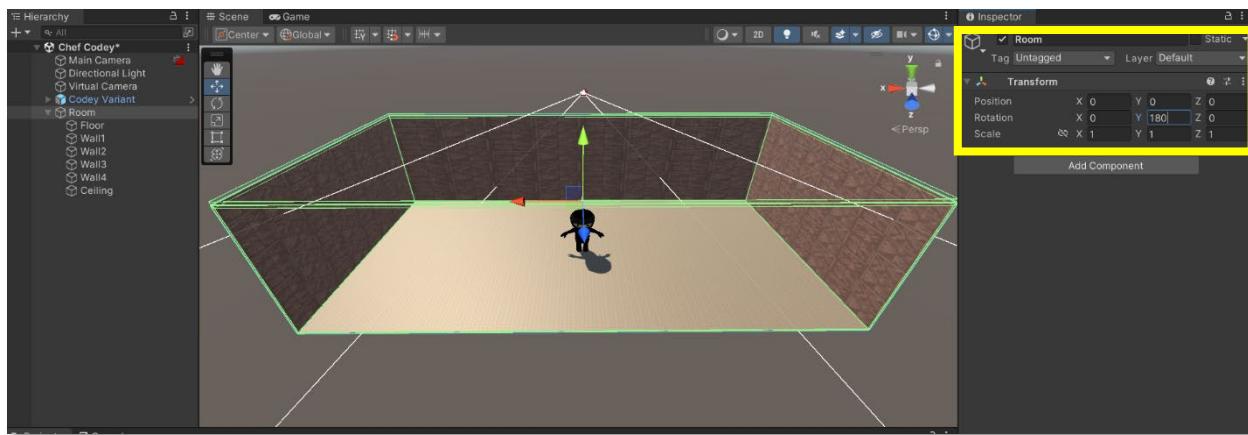
**20**

If Codey does not move up when you press up (or w), then you need to **rotate** and move the **camera**. To match Codey's movement **script**, the camera must be pointing in the **positive Z** direction. Since we are using **Cinemachine**, change the **transform properties** on the **CM vacam1** game object. Make sure that the **camera** is pointing to the back wall of your room.



21

Make sure to adjust the **rotation** of your room **game object** to align it if necessary. Instead of adjusting each **plane** individually, you can change the **rotation** of the room **object**.



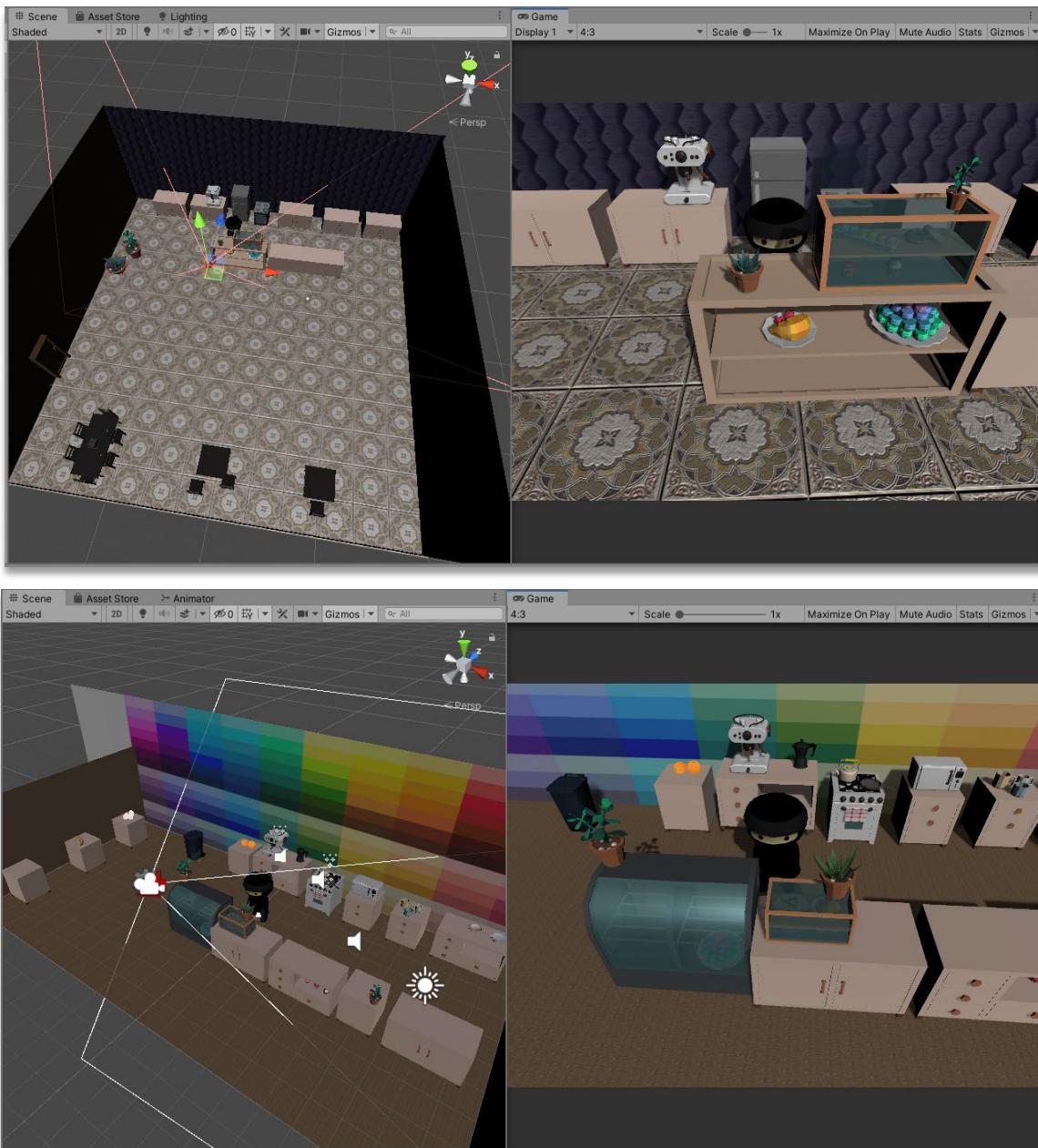
### Sensei Stop

Discuss your room design with your sensei. Does this align with your planning document? Play test to make sure that Codey cannot run through walls. What type of textures did you use? How does this fit your theme?

## Extreme Café Makeover

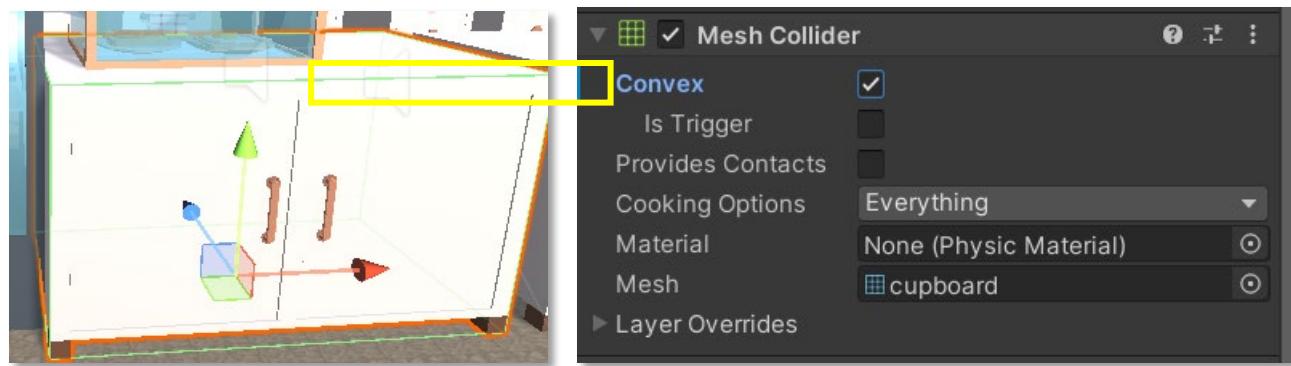
**22** Using your Ninja Planning Document, use the **Unity Asset Store** to find **assets** to place in your room. Use keywords like “coffee”, “park”, “shop”, and “classroom” to help you find cool **assets** to use!

As you place your items, be sure to **playtest** frequently. It’s a good idea to place your **objects** in an empty **game object**. This helps keep your **hierarchy** clean.



## 23 See if Codey can clip into the **objects** in your **scene**.

Each **asset** you download will have different **properties** and **components**. **Objects** in your **scene** might need to be given **box**, **capsule**, **sphere**, or **mesh colliders**. If you use a **mesh collider**, make sure the **Convex** checkbox is enabled.

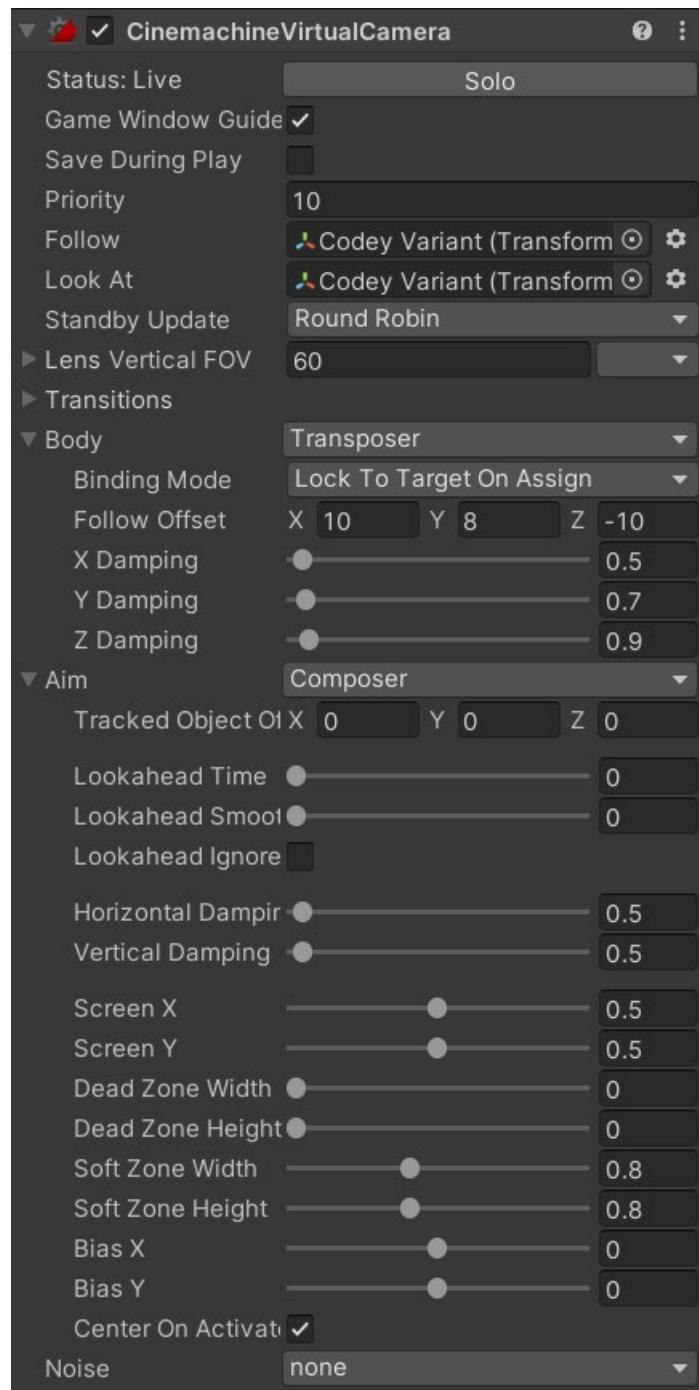


### Mesh Colliders

A Mesh Collider is special because it matches the shape of the object.

Because the mesh colliders are more complex and require many computations, Unity prevents interaction between mesh and colliders by default. To have mesh colliders interact properly with the other colliders (like the one on Codey), you must enable its Convex property.

**24** You can program the camera to track Codey around the scene by changing a few properties in **Virtual Camera Game Object**.



Don't be afraid to change the properties until you like the camera.

## The Toast with the Most

**25** With our **environment** set up, we can start on the game **logic**.

The player can pick up **items**, transform the **items**, and build new **items**. The player's goal is to cook all the **items** available as quickly as possible.

We can tackle the **logic** piece by piece. First, we can start with food "sources" that give an **item** to Codey to hold.

Start by creating an empty **object** to hold all our sources.



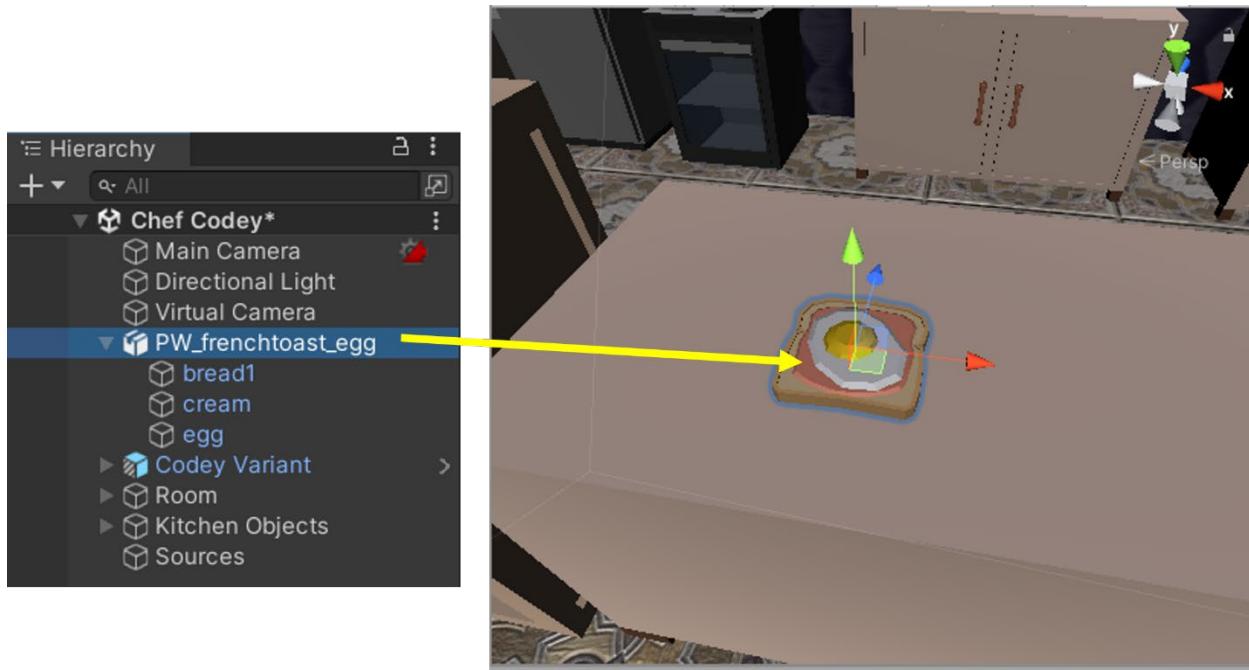
**26** Revisit the **Unity Asset Store** to find some objects for Codey to make. For example, you can use a French toast asset that is built out of three smaller parts.



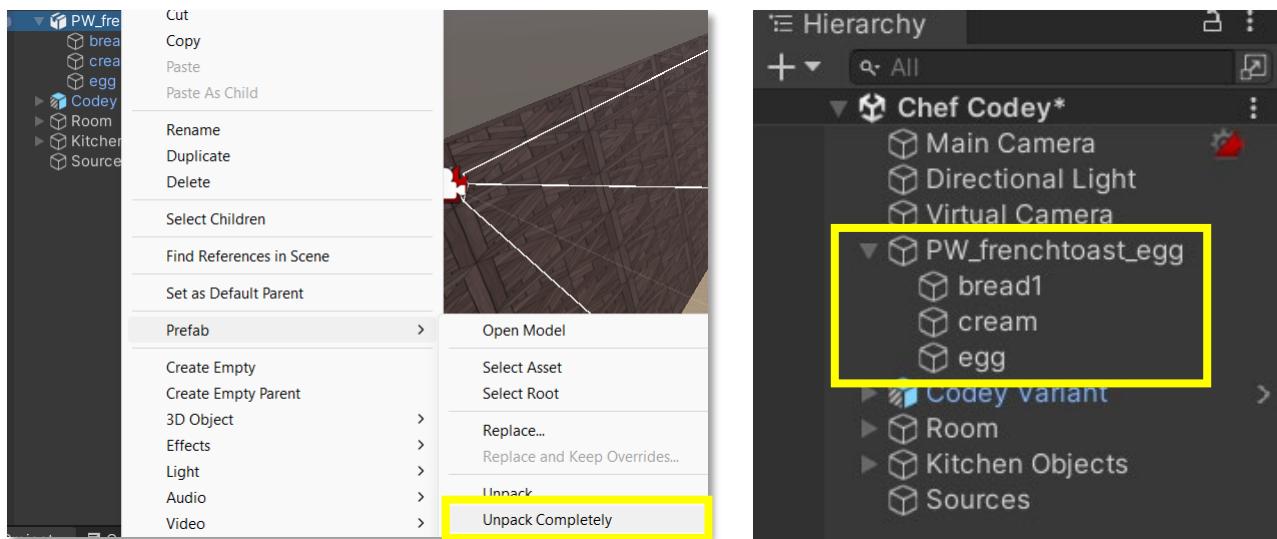
### Asset Parts

Every asset will be different. You might find one that is built out of many different smaller parts, or you might have to take several assets and combine them together to make your own!

**27** Once you have your **model**, you might have to separate the pieces to use in your **scene**. If your **object** is not made out of different parts, then the next steps might not be necessary.

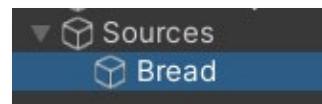


**28** Right click the blue object in the **hierarchy** and select "Unpack Prefab Completely". Remember to resize your **object** so it looks like it belongs in your **scene**.

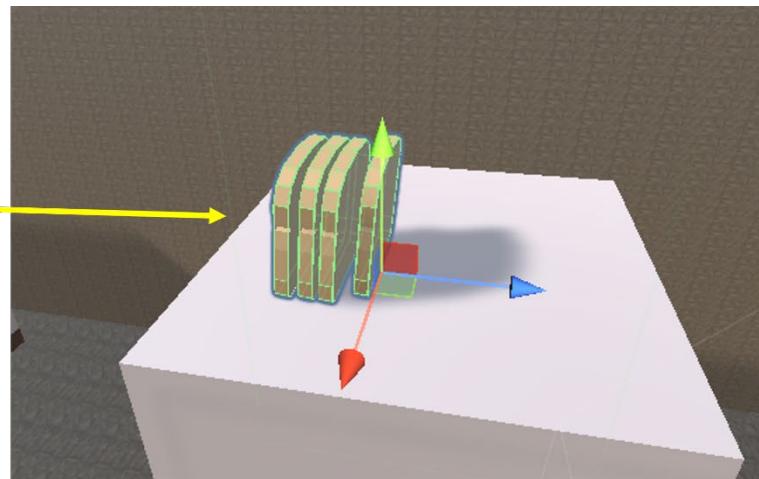


**29** Now the **object** names are black meaning we can edit them and save them as our own **prefabs**!

Let's start with the bread. Create a new empty **object** inside of the Sources **object** and call it Bread.



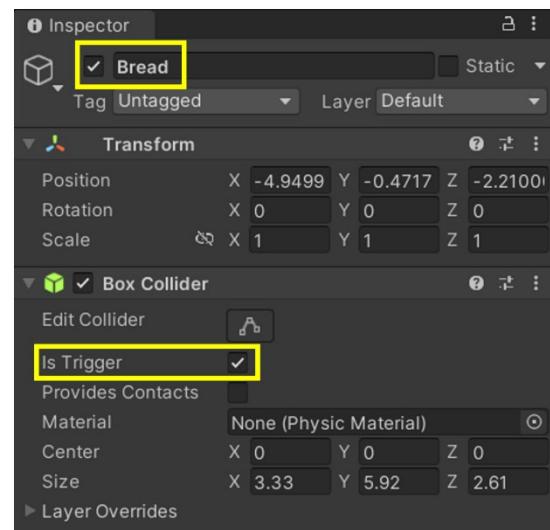
**30** Place the bread **object** into our new empty **object** and duplicate it several times. We want to place these slices on a counter to tell the player where to find bread.



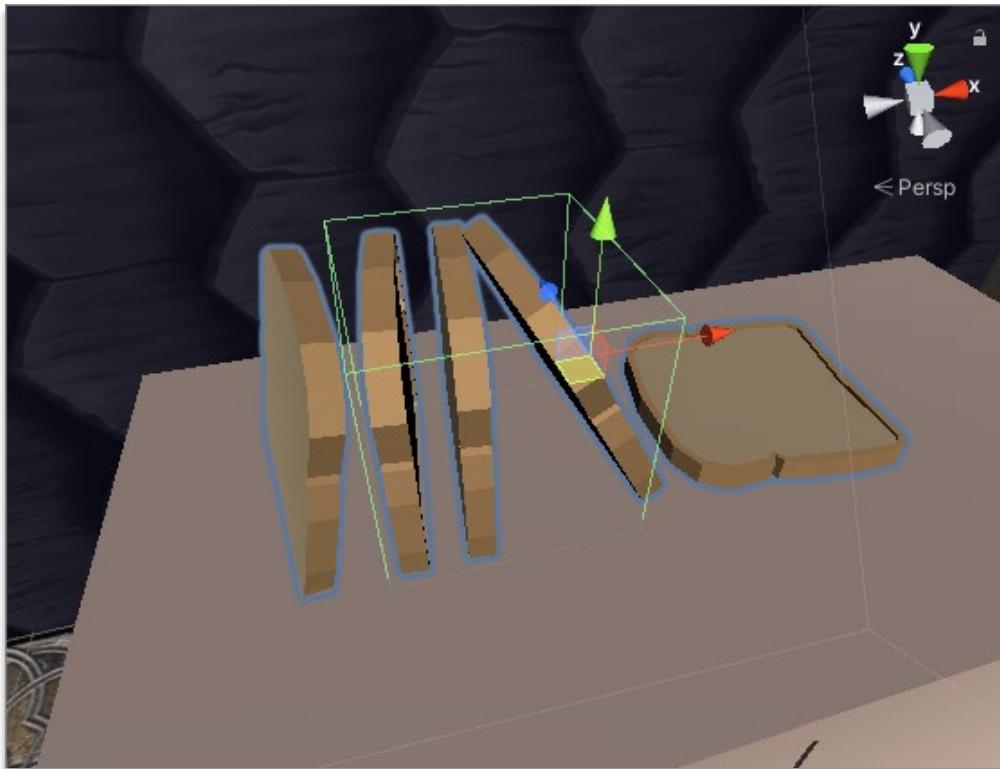
**31** We need a way to tell if Codey is close enough to pick up a piece of bread. We don't want to give Codey the ability to teleport **objects** from around the **scene**!

Add a **box collider** to the Bread game **object**. Make sure that you don't accidentally add a box collider to any of the individual game **objects** – we want our **collider** on the **parent container object**.

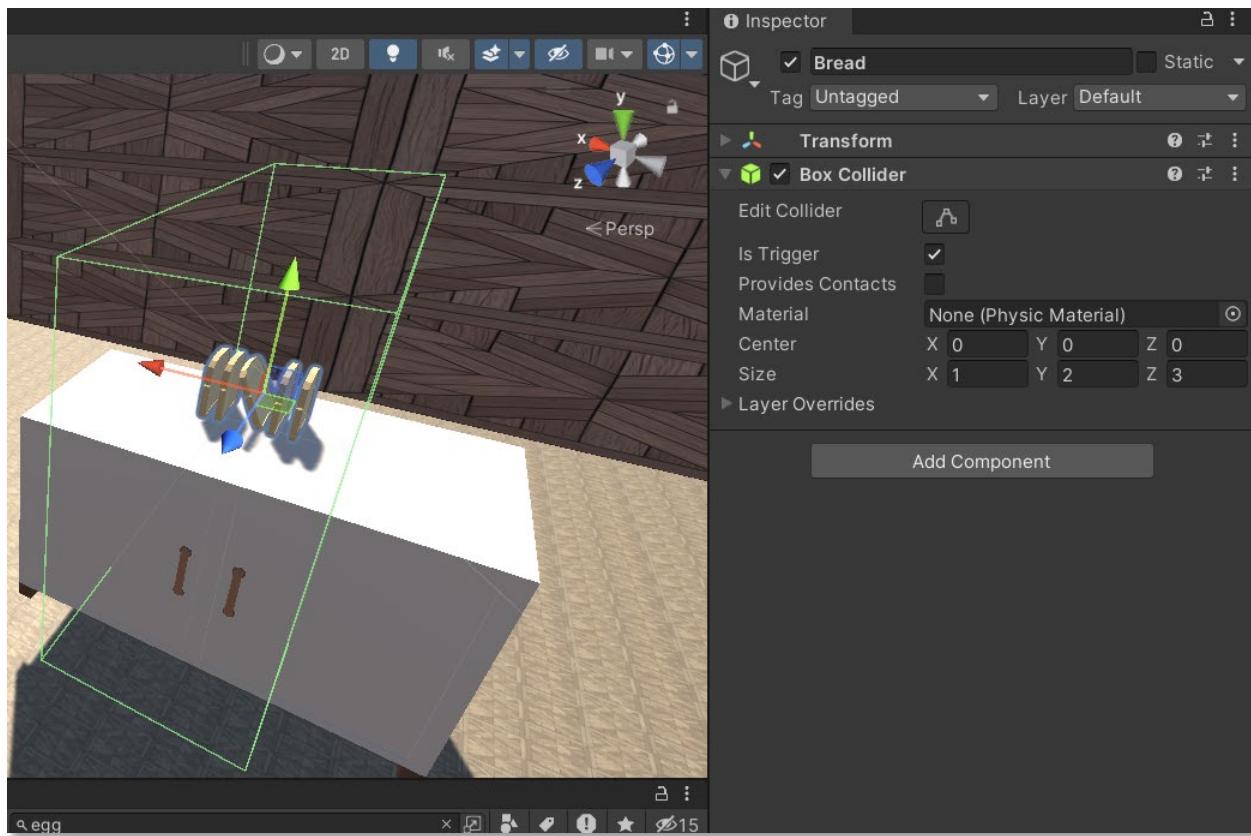
Enable **Is Trigger** on your **Collider**.



**32** If you notice that your **objects** are not aligned with your **box collider**, don't be afraid to reset each of the **model's coordinates** to zero and reposition them. Part of game making includes constant adjustments!



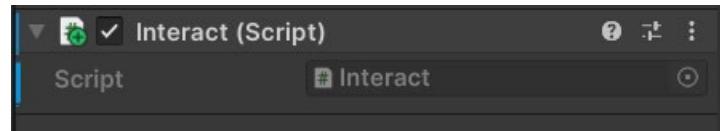
**33** Use the **collider's Center** and **size** properties to make a box that extends in front of the **models**.



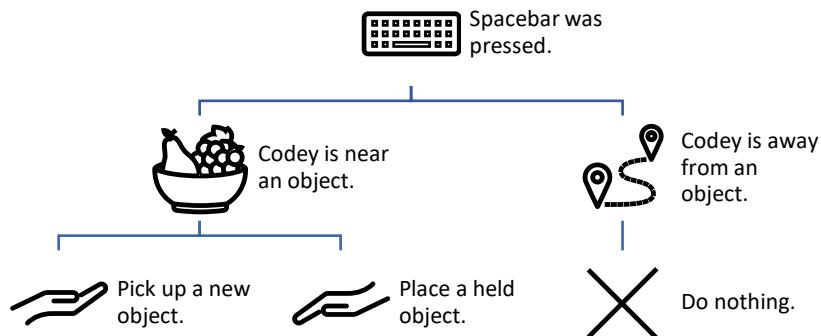
**34** We are going to use this **collider** to know when Codey is close enough to the bread to pick it up. We need to create a **script** and **program** Codey to pick up an item.

Create a new **script** in your Script folder in the **Assets** tab and name it **Interact**.

Attach the **script** to the Codey **game object** and open it in Visual Studio.



**35** Whenever the player presses the **spacebar**, we need to ask two things -first, is Codey standing near an object? Second, is Codey picking up or setting down an object? We can sketch out our logic in a chart.



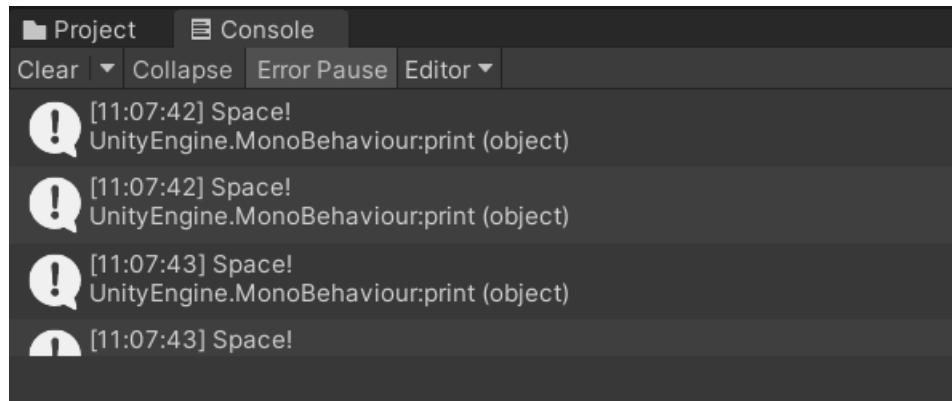
We can think of this logic as several branching **if** and **if else statements**. Since we are responding to when the **spacebar** is pressed, we will write our logic in the **Update function**.

**36** Knowing when the spacebar is pressed is very easy in Unity. Try to think about previous games you created and how you listened for player input.

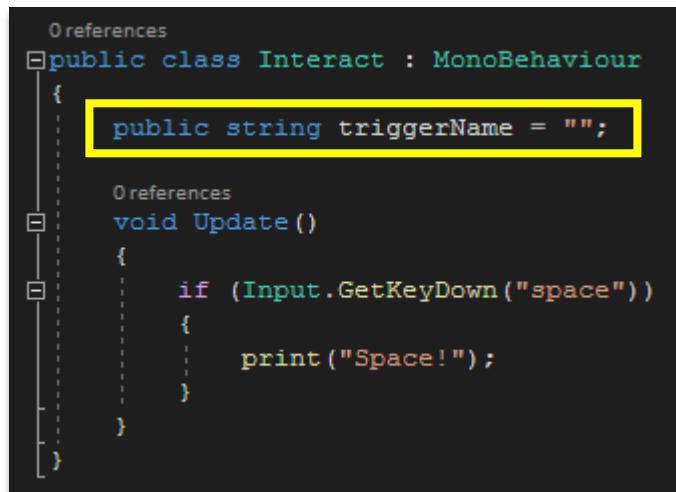
In the **Update function**, create an **if statement** that checks to see if the space **key** was pressed by using the **Input object's GetKeyDown function**. Inside the **if statement**, **print** a message to the **console**. You can remove the **Start function**.

```
public class Interact : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown("space"))
        {
            print("Space!");
        }
    }
}
```

**37** Save your **script** and **playtest** your game. What happens when you press the spacebar?



**38** Next, we need to know if Codey is near an **object**. This requires tracking when Codey enters and exits the food object's **collider trigger**. Back inside the Interact **script**, create a **variable** to store the name of the trigger that Codey collides with.



**39** Add the two Unity **functions** that run when a **game object** enters or exits a **trigger**.

```
0 references
public class Interact : MonoBehaviour
{
    public string triggerName = "";

    0 references
    void Update()
    {
        if (Input.GetKeyDown ("space"))
        {
            print ("Space!");
        }
    }

    0 references
    private void OnTriggerEnter(Collider other)
    {
    }

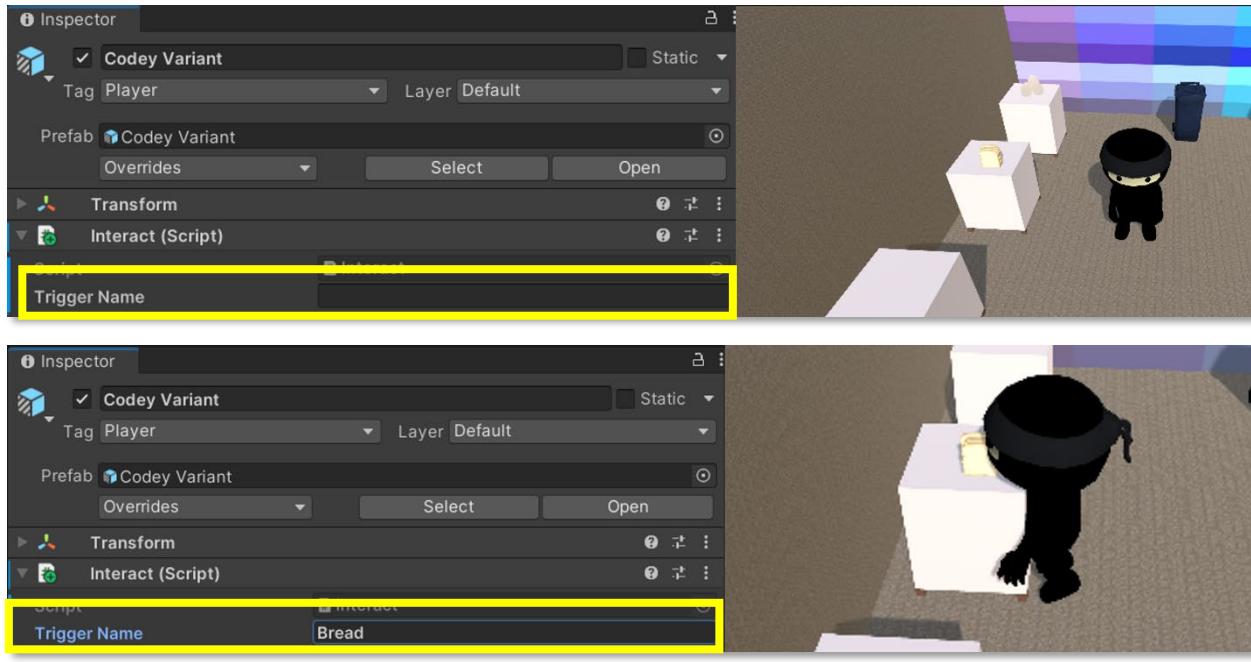
    0 references
    private void OnTriggerExit(Collider other)
    {
    }
}
```

**40** When Codey enters a **trigger**, we want to set the **triggerName variable** to the **name** of the **collider**. When Codey leaves a **trigger**, we want to reset the **triggerName variable** to an empty **string**.

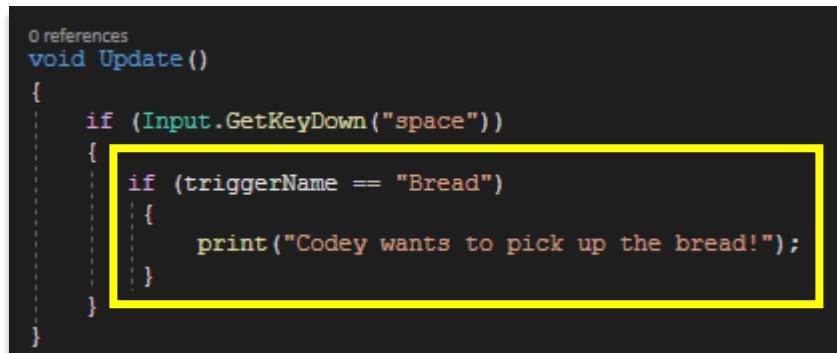
```
0 references
private void OnTriggerEnter(Collider other)
{
    triggerName = other.name;
}

0 references
private void OnTriggerExit(Collider other)
{
    triggerName = "";
}
```

**41** Playtest your game. What happens to the **triggerName variable** in Codey's **inspector** when you get close to the bread **objects**? What happens when you walk away from the bread **objects**?

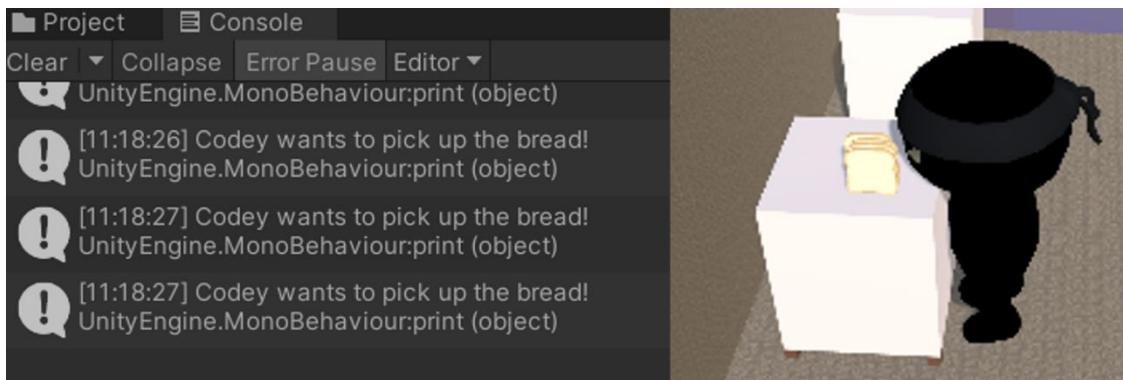


**42** We can now combine our **input code** with our new **trigger code**. Inside our **if statement** inside the update **function**, write a new **if statement** that checks to see if the **triggerName variable** is equal to "Bread". If it is, then print a message in the **console**.



If you used an **object** other than bread, make sure that you use the correct **name** of your **object**!

**43** Playtest your game. Now pressing the spacebar should only send a message if Codey is standing inside the bread **object's trigger**.



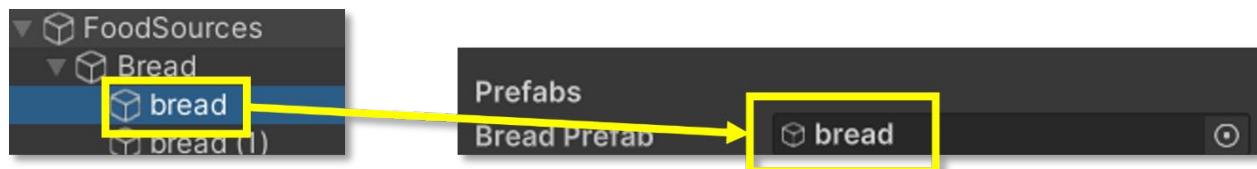
**44** First, Codey needs to know what **object** to pick up. In the **Interact script**, create a **public game object** and name it **breadPrefab**.

```
public class Interact : MonoBehaviour
{
    public string triggerName = "";

    public GameObject breadPrefab;

    void Update()
    {
```

**45** Save the **script** and go back to the Unity Engine. Drag a bread **object** from the **hierarchy** and drop it into the Bread Prefab **variable** in Codey's **inspector**.



**46** Codey also needs a way to hold an **item**. In the **Interact script**. Create a **public game object** named **heldItem**.

```
public class Interact : MonoBehaviour
{
    public string triggerName = "";

    public GameObject breadPrefab;

    public GameObject heldItem;

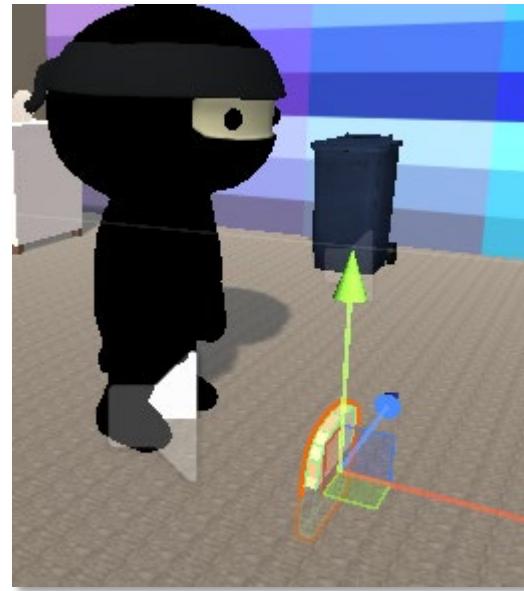
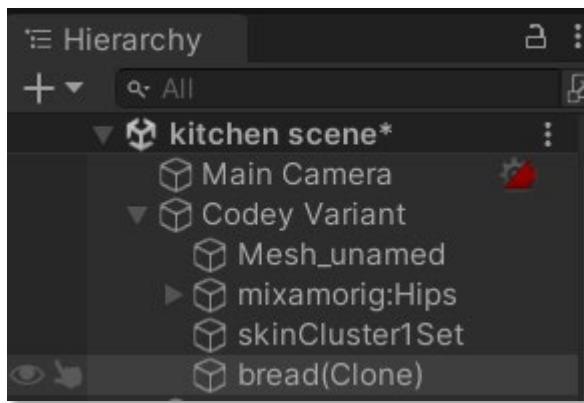
    void Update()
    {
```

**47** We need to set **heldItem** to a **new game object** that we create with **code**. Inside the **if** the **trigger** name is Bread **statement**, we will use a special version of the **Instantiate function**.

The first **argument** is the **model** or **prefab** of the **object** we want to create. The second **argument** is what **game object** we want as our new **object's parent**. The final **argument** is **false** to tell Unity how to position the new object in relation to the world.

```
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")
        {
            heldItem = Instantiate(breadPrefab, transform, false);
        }
    }
}
```

**48** Playtest your game and try to pick up a slice of bread.



You can check to see whether Codey has picked up the bread by clicking on Codey in the **hierarchy**.

**49** Codey now has a piece of bread, but it's not in the best **position**. Stop the game and open your Interact **script** again. After the Instantiate line, we can set the heldItem's **position**.

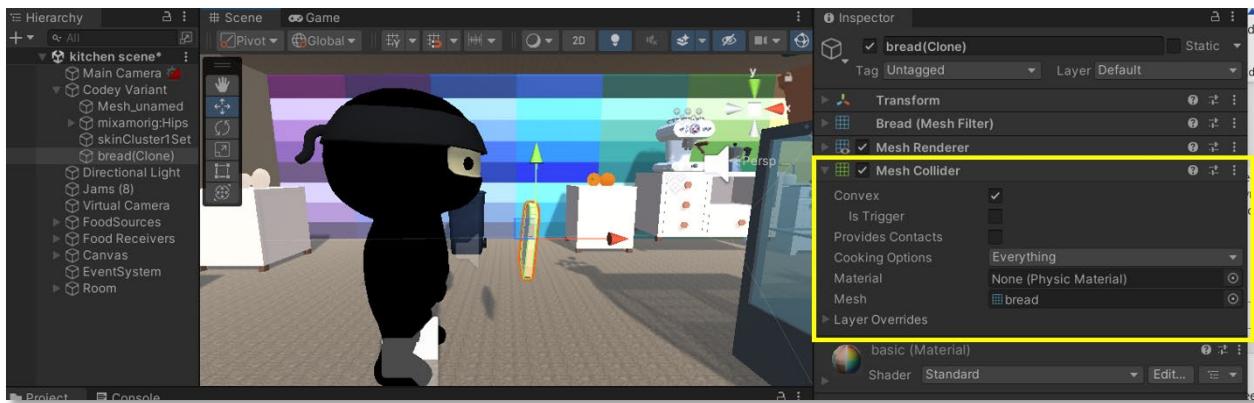
```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")
        {
            heldItem = Instantiate(breadPrefab, transform, false);
            heldItem.transform.localPosition = new Vector3(0, 2, 2);
        }
    }
}
```

We need to use the **transform's local position** to place the piece of bread based on Codey's **position**, not the world's **position**.

Using the **localPosition** property means that the **object** will be placed zero **x** units, 2 **y** units, and 2 **z** units away from Codey's **position**.

If we used the **position** instead, the **object** would be placed 0 **x** units, 2 **y** units, and 2 **z** units away from the world's center.

**50** Save and close the Interact script, and playtest your game again. You should now be able to see the piece of bread Codey has picked up.



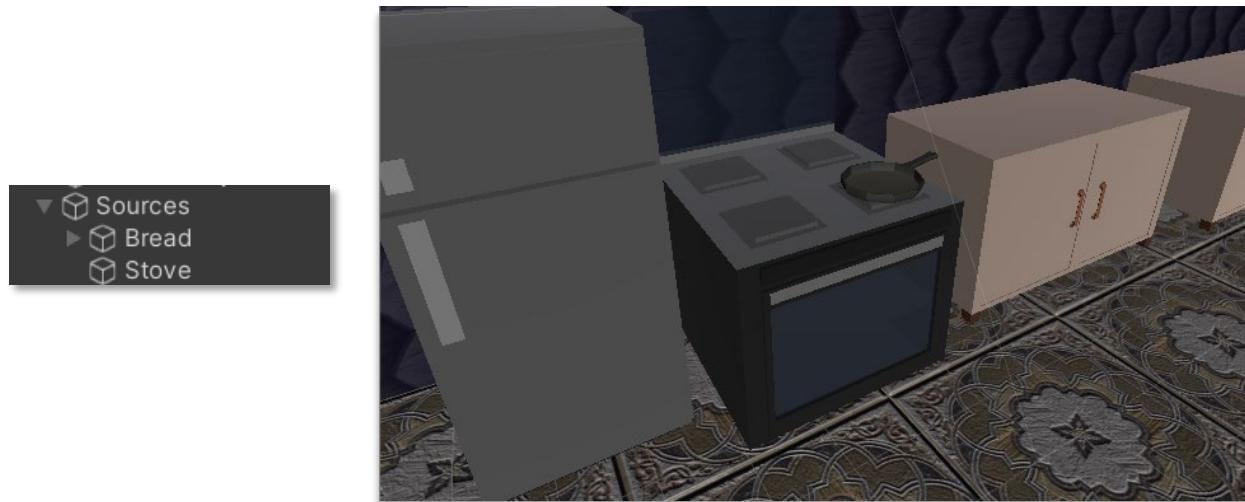
### Sensei Stop

While the slice of bread might look good at  $(0, 2, 2)$ , it might not be quite right for other objects. Change the values of your Vector3 in the code to position your object. Tell your Sensei how you tested your different values.

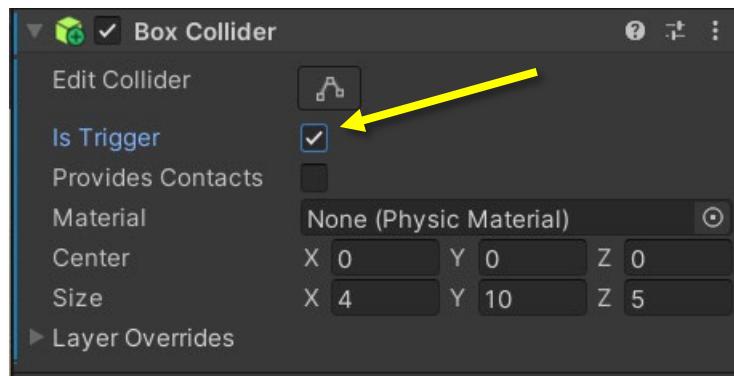
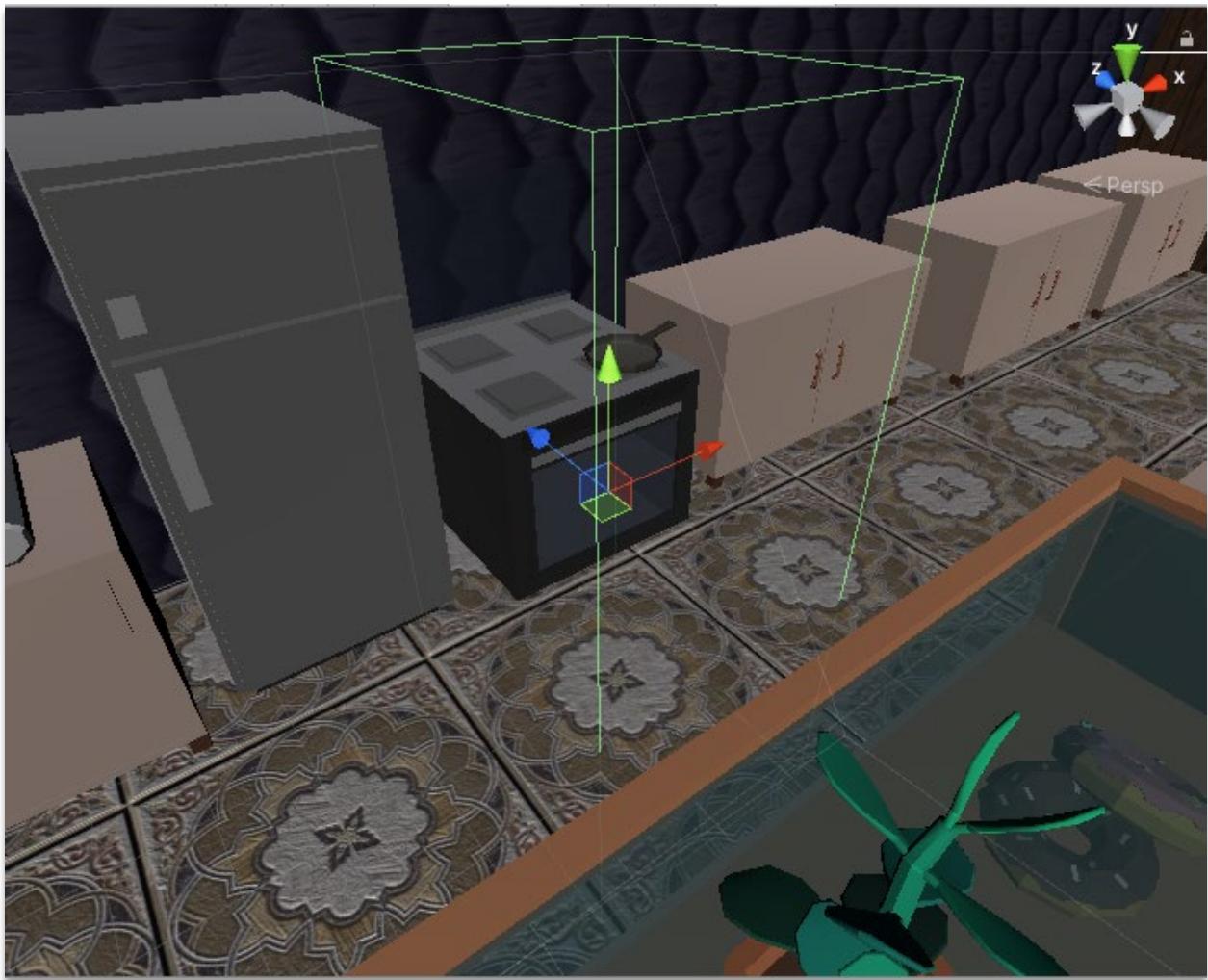
## Now We're Cooking

**51** In many resource gathering games, you need to process **items** before you can use them. You might have to chop wood, smelt stone, or even toast bread. In this game, we will need to process our bread **object**.

Next, we can set up a stove to cook our food. Find a stove or another **object** in the Unity Asset Store and place it in your **scene**. In your Source **game object**, create a new empty **object** named Stove or the name of your **object**.



**52** Add a **box collider** to this new stove **object**. Move and resize it so it overlaps with the **model**. Make sure you check the **Is Trigger** box.

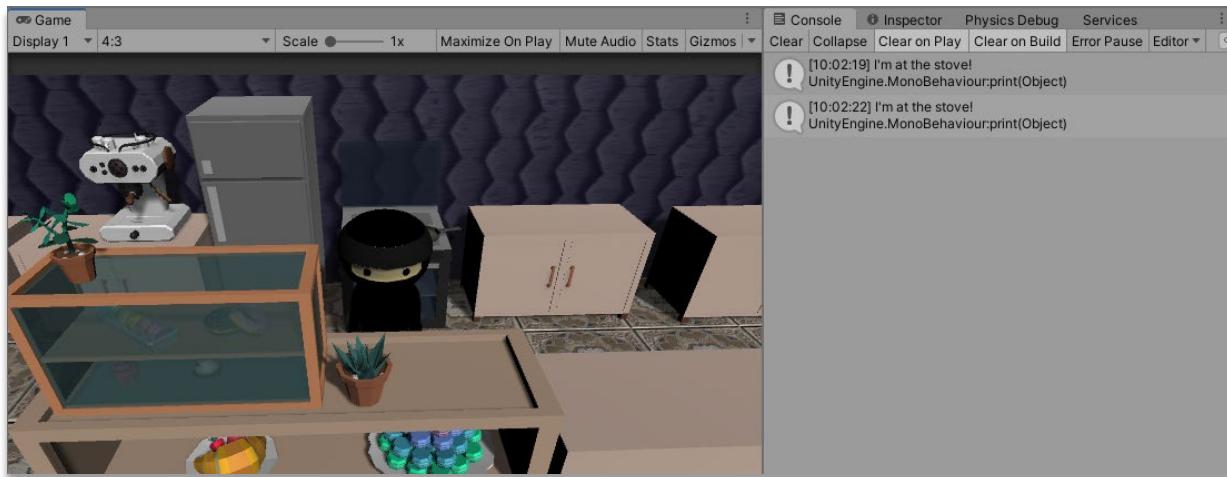


- 53** In the Interact script, we can add another **if statement** to print a message if the space key is pressed while Codey is inside the stove's **box collider trigger**.

 **Sensei Stop**

Using the same conditional that we used for the bread trigger, can you code a message to appear if the space bar is pressed when Codey is inside the stove's box collider trigger?

- 54** Play the game and make sure you can see the message in the **console** when Codey is near the stove and you press the spacebar.



**55** Now that Codey knows how to interact with the stove, we can toast the bread. We need an easy way to keep track of what item Codey is holding. In the Interact **script**, create a new **public string variable** named heldItemName.

```
public class Interact : MonoBehaviour
{
    public string triggerName = "";

    public GameObject breadPrefab;

    public GameObject heldItem;
    public string heldItemName;

    void Update()
    {
```

**56** We need to set the **value** of this **variable** when Codey picks up a piece of bread. Inside the **if statement** that checks the bread **trigger**, set the **variable's value** to "breadSlice".

```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")
        {
            heldItem = Instantiate(breadPrefab, transform, false);
            heldItem.transform.localPosition = new Vector3(0, 2, 2);
            heldItemName = "breadSlice";
        }

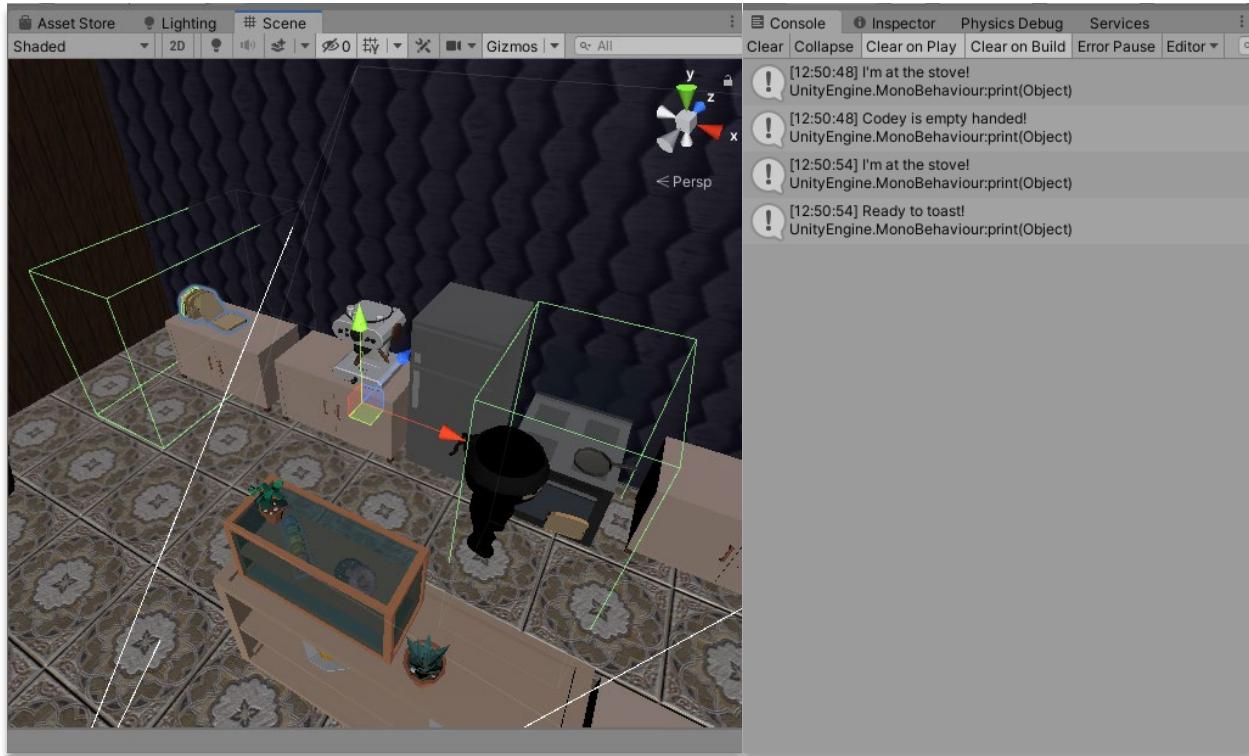
        if (triggerName == "Stove")
        {
            print("I'm at the stove!");
        }
    }
}
```

- 57** Now that we know what item Codey is holding, we can check to see if Codey is holding it when near the stove. Create **if else statements** for when Codey is holding the “breadSlice” and when Codey is not holding any items.

```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")
        {
            heldItem = Instantiate(breadPrefab, transform, false);
            heldItem.transform.localPosition = new Vector3(0, 2, 2);
            heldItemName = "breadSlice";
        }
    }

    if (triggerName == "Stove")
    {
        print("I'm at the stove!");
        if (heldItemName == "breadSlice")
        {
            print("Ready to toast!");
        }
        else
        {
            print("Codey is empty handed!");
        }
    }
}
```

**58** Playtest your game. Press the spacebar near the stove with and without holding bread. Make sure that you can see all the different **console** messages.



**59** Next, we need to place Codey's held item onto the stove. First, duplicate a bread object and add it to the empty Stove object. Position it wherever you want – on top or inside the stove. Rename it to “Toast” so it’s easier to differentiate from our bread source.



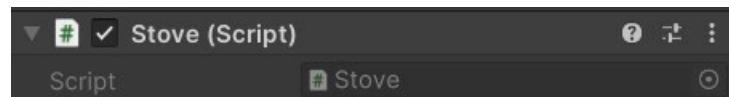
### Like Magic

In game development, there are a lot of tricks that make the player think something complicated is happening when something simple is actually taking place. Instead of using the exact piece of bread that Codey is holding, we can give the stove its own piece of bread and destroy the one Codey is holding.

**60** Codey's job is to pick up and place the items on the Stove, and the Stove's job is to cook the object. Since the Stove has its own job, we need to create a new **script** to manage the cooking logic.

Create a new **script** in your Script folder in the **Assets** tab and name it **Stove**.

Attach the **script** to the Stove **game object** and open it in Visual Studio.



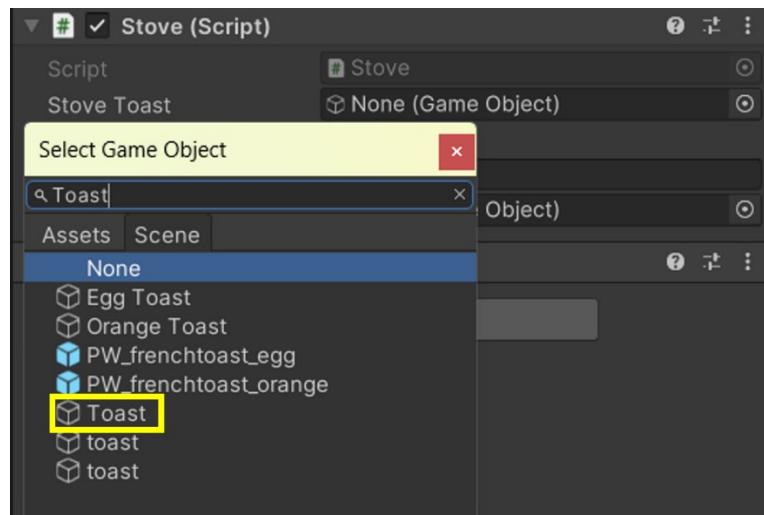
**61** The Stove needs to know about the Toast **object**, so create a **public game object variable** named **toast**. You can delete the **Update function**.

```
public class Stove : MonoBehaviour
{
    public GameObject toast;

    void Start()
    {
    }

}
```

**62** Save and close Visual Studio. In the Unity **inspector**, set the **value** to the toast **object**.



**63** Open the Stove **script** again and create a new **public void function** named ToastBread.

```
public class Stove : MonoBehaviour
{
    public GameObject toast;

    void Start()
    {
        ...
    }

    public void ToastBread()
    {
        ...
    }
}
```

### Sensei Stop

Write a few sentences or design a flowchart to describe the ToastBread function's job. What's the first thing that it needs to do when Codey interacts with the stove? What does it need to do before Codey can interact with the stove again?

**64** This **function** needs to take Codey's bread, place it on the stove, cook it, and let Codey pick it back up. Let's first have Codey put down and pick up the bread. We need to make sure there isn't a piece of toast on the stove when the game starts.

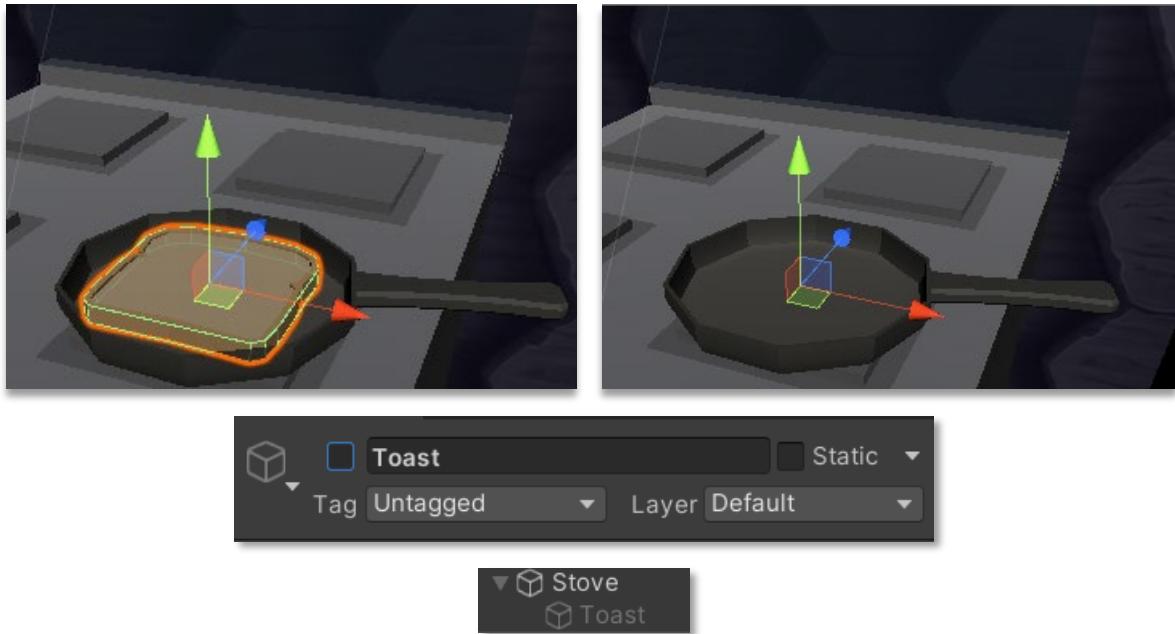
In the Start **function**, deactivate the toast **game object** when the game begins using the **game object**'s SetActive **function**.

```
public class Stove : MonoBehaviour
{
    public GameObject toast;

    void Start()
    {
        toast.SetActive(false);
    }

    public void ToastBread()
    {
    }
}
```

**65** Playtest your game to see the toast disappear when the game starts running.



**66** In our ToastBread **function**, we need to **enable** our toast object and make it available for Codey to pick up. In the **function**, we need to do the opposite of what we did in the Start **function**.

We can use the same SetActive **function**, but with **true** as the **parameter**.

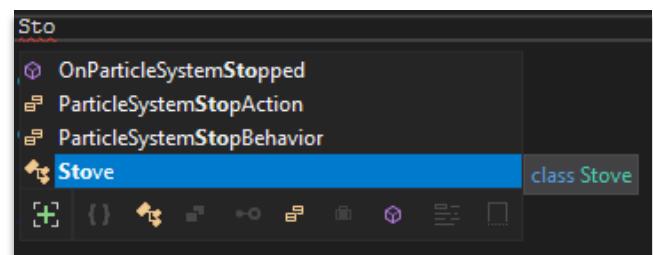
```
public class Stove : MonoBehaviour
{
    public GameObject toast;

    void Start()
    {
        toast.SetActive(false);
    }

    public void ToastBread()
    {
        toast.SetActive(true);
    }
}
```

**67** This **function** needs to run when the player presses the spacebar when Codey is standing next to the stove with a piece of bread. Open the Interact **script**. Add a **public variable** of the Stove **script**. This will let us reference any **variables** and **functions** from the Stove **script** inside our Interact **script**.

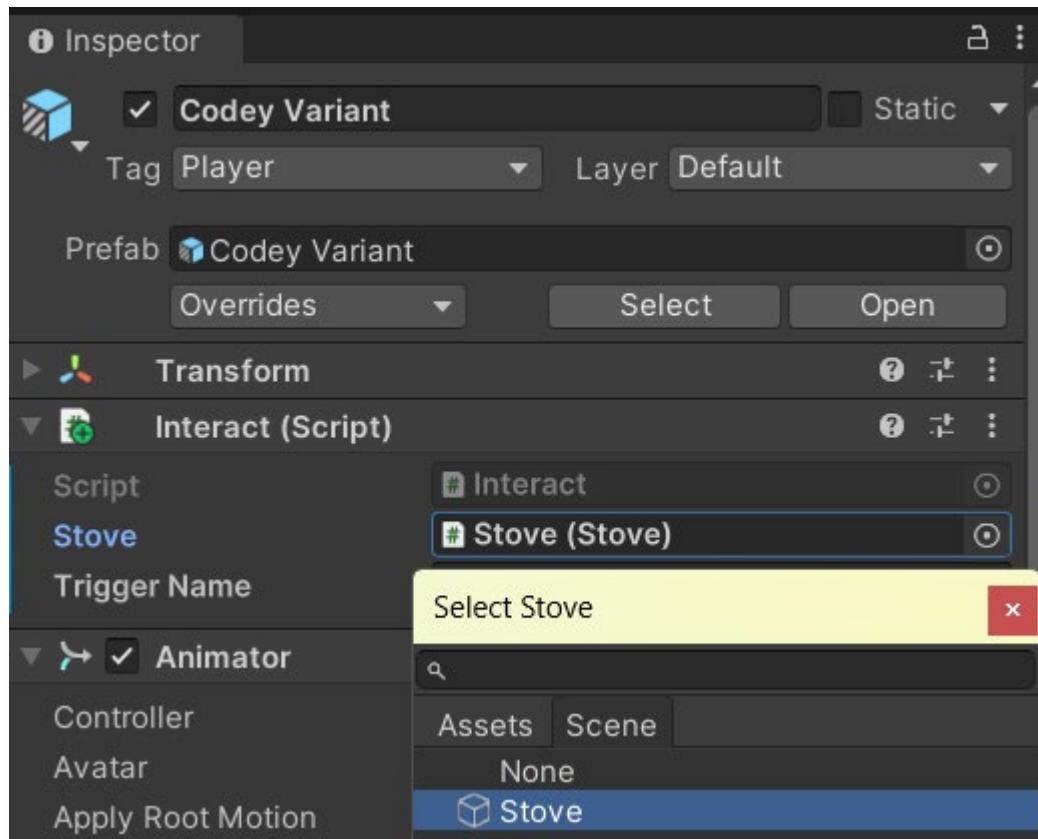
Visual Studio knows that we have a Stove **script** and will suggest it for us!



```
public class Interact : MonoBehaviour
{
    public Stove stove;

    public string triggerName = "";
}
```

**68** Save the script and go back to Unity. Click on Codey in the hierarchy to open the components in the inspector. Set the Stove variable to the Stove object.



**69** With the stove connected to our Interact **script**, we can have Codey run the stove's **ToastBread function** when the **conditions** are correct.

### Sensei Stop

Find the correct if statement and call the stove's **ToastBread** function, destroy the object that Codey is holding, and reset the value of **heldItemName** to an empty string.

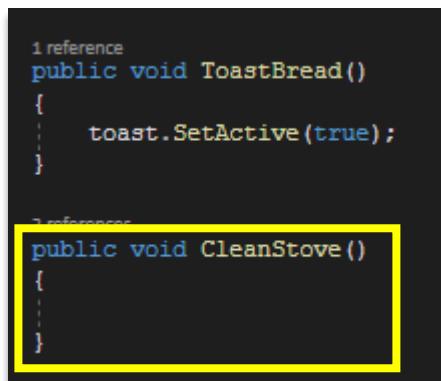
---

**70** Playtest your game. Test what happens when Codey interacts with the stove with and without a piece of bread.

---

**71** We can now have Codey pick up the toast from the stove. This requires the stove giving an object to Codey, so we need to build a little more **logic** inside of the Stove **script**.

Open the Stove **script** and create a new **public void function** in the named CleanStove.

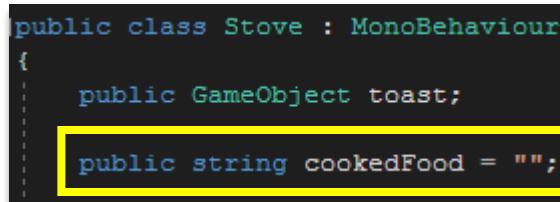


```
1 reference
public void ToastBread()
{
    toast.SetActive(true);
}

2 reference
public void CleanStove()
{
}
```

---

**72** We need to **disable** the toast object and tell Codey what **object** was picked up. Create a **public string variable** named cookedFood and **initialize** it to an empty **string**.



```
public class Stove : MonoBehaviour
{
    public GameObject toast;

    public string cookedFood = "";
}
```

**73** Add a line to the ToastBread function that sets the cookedFood string to "toast".

```
1 reference
public void ToastBread()
{
    toast.SetActive(true);
    cookedFood = "toast";
}
```

**74** Inside the CleanStove function, **disable** the toast **object** and reset the **value** of the cookedFood **variable** to an empty **string**.

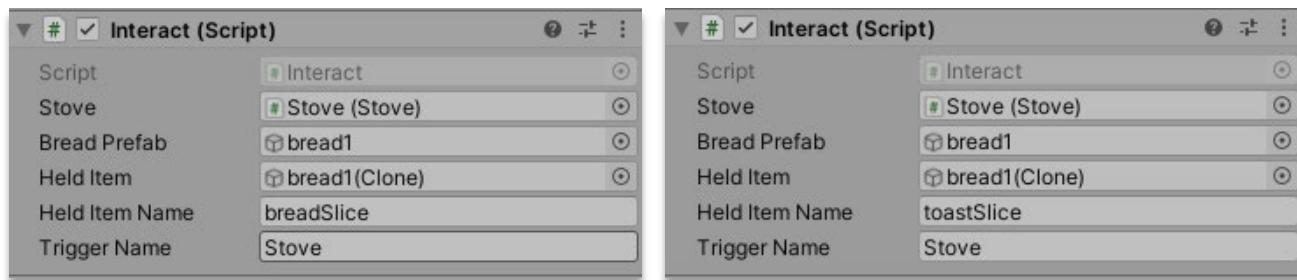
```
public void CleanStove()
{
    toast.SetActive(false);
    cookedFood = "";
}
```

**75** Back in the Interact **script**, we need to have Codey check to see if the cookedFood is ready.

If the stove has cooked toast, **then** give Codey the bread **prefab** and position it. Change the **value** of heldItemName to “toastSlice” and run the CleanStove **function**.

```
if (triggerName == "Stove")
{
    if (heldItemName == "breadSlice")
    {
        stove.ToastBread();
        Destroy(heldItem);
        heldItemName = "";
    }
    else
    {
        if (stove.cookedFood == "toast")
        {
            heldItem = Instantiate(breadPrefab, transform, false);
            heldItem.transform.localPosition = new Vector3(0, 2, 2);
            heldItemName = "toastSlice";
            stove.CleanStove();
        }
    }
}
```

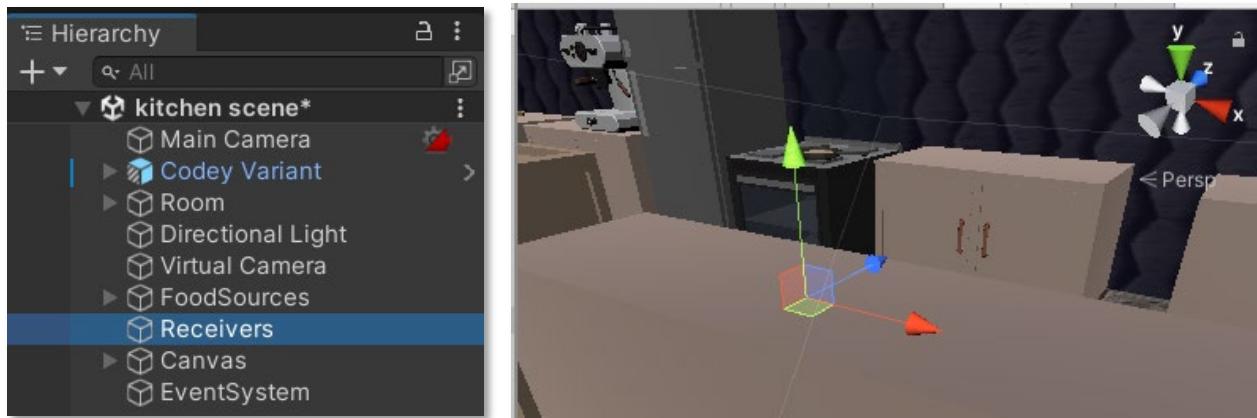
**76** Playtest your game. Place a piece of bread on the stove and try to pick it up again. It should look like Codey cooked the bread!



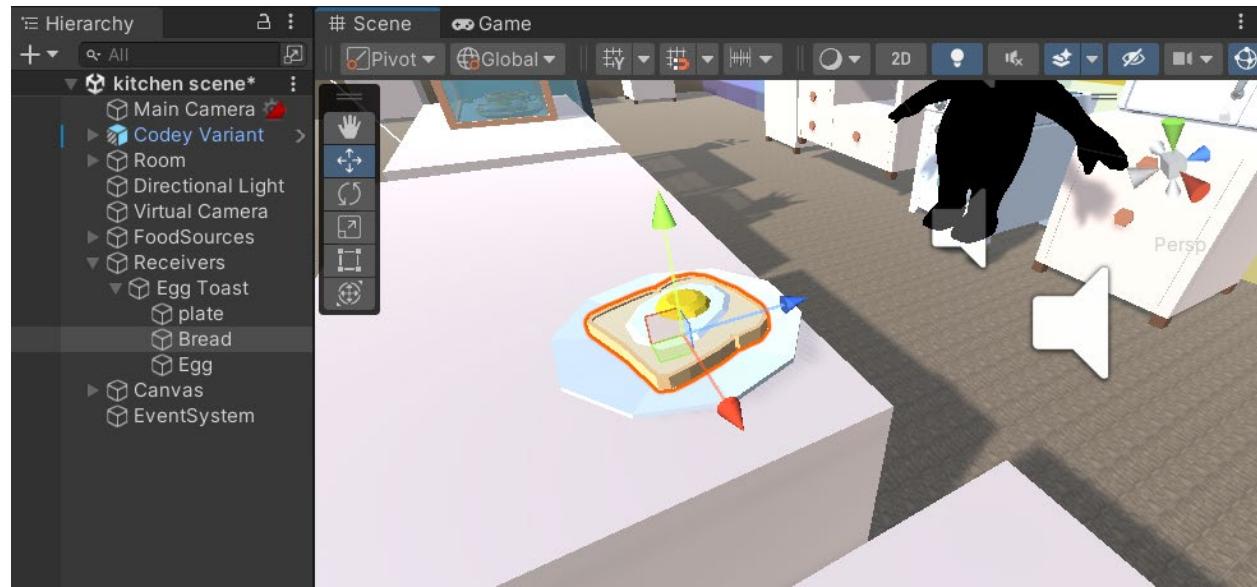
Even though they use the same **model**, our held item name tells us that Codey is holding a slice of toast.

## Order Up

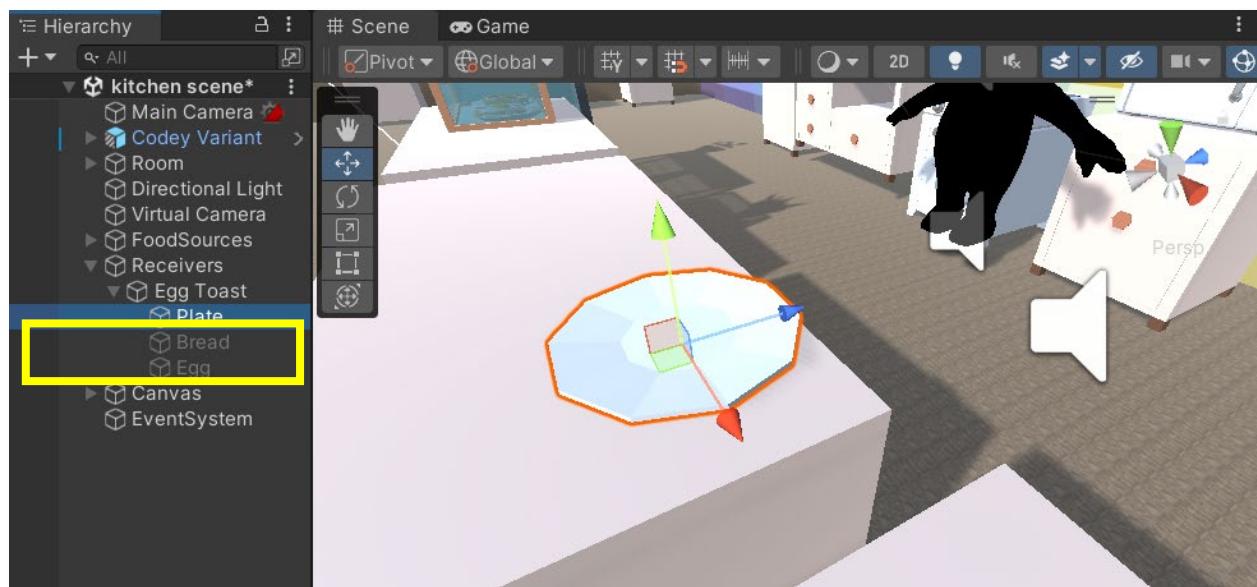
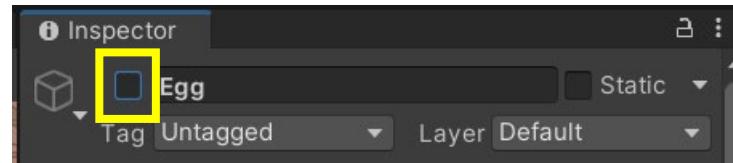
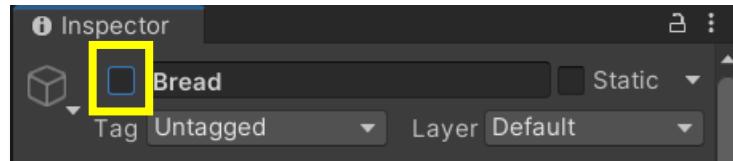
77 Now we need a place to put our toast. Create a new empty **object** named Receivers and place it somewhere on a counter in your **scene**. This object will hold all our finished food items and dishes.



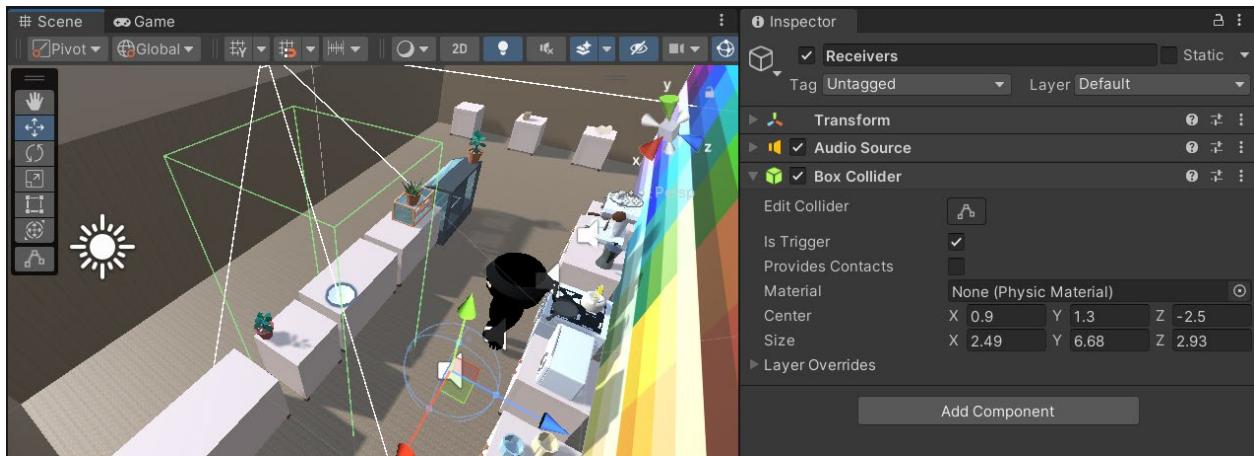
78 Find or build a finished **object** using **models** from the Unity Asset Store. Remember that we are making French Toast with an egg. Unpack your **prefab** and **rename** your objects in you need to. It is important for the object **names** to match up with the values we use for heldItemName.



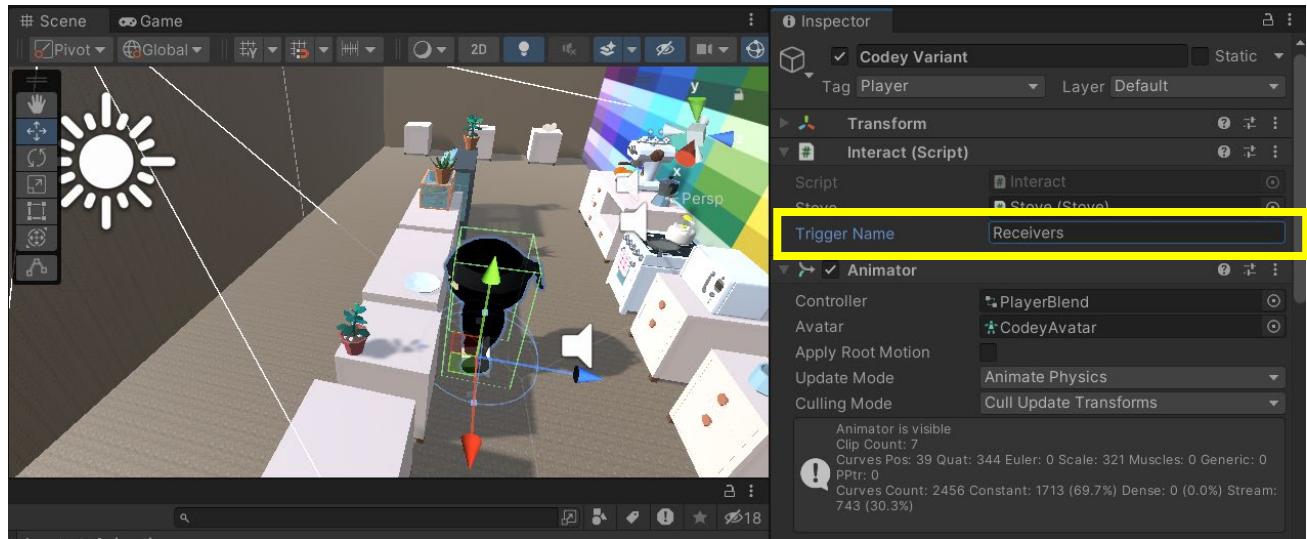
**79** Once you have your items placed, **disable the objects** that you want Codey to cook by clicking the checkbox next to the name of the **object** in the **inspector**.



**80** Add a **box collider** to the Receivers **object**. Make sure you set its **position** and **size**. Make sure that **Is Trigger** is enabled.



**81** Playtest your game and move Codey inside of the new **collider**. Make sure the **name** is properly displaying for the **value** of Trigger Name in the **inspector**.



- 
- 82** Open the Interact **script**. Add a new **if statement** in Codey's Update **function** inside the **if space key pressed conditional** that checks to see **if** Codey is inside the Receivers **trigger**.

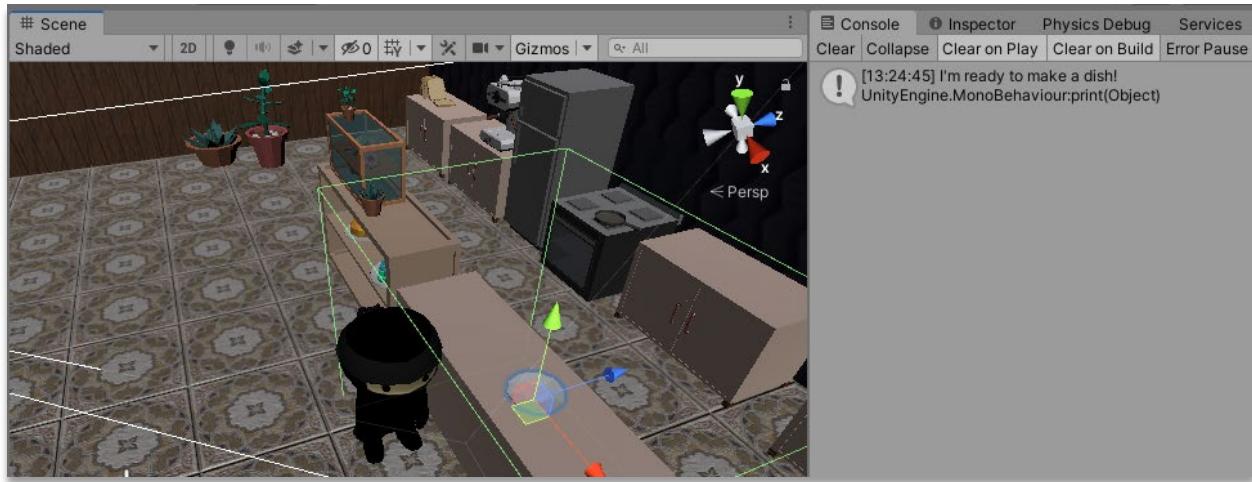
```
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")...
        if (triggerName == "Stove")...
        if (triggerName == "Receivers")
        {
            print("I'm ready to make a dish!");
        }
    }
}
```



### Pro Tip

You can use the [-] buttons to the right of the line numbers to collapse or highlight groups of code to help you focus on coding in the right place.

**83** Save and return to the Unity Engine. Playtest your game and make sure you see the message in the **console** when you press the space bar while Codey is standing close to the Receivers game **object**.



Notice how our **code** just works with our new **collider**? It's because we set up **OnTriggerEnter** and **OnTriggerExit** to handle all the work for us.

## 84

We can now write **code** to know when Codey is holding toast and when Codey's hands are empty.

Inside this new **if statement**, create an **if statement** that checks to see **if** Codey is holding the "toastSlice" **item**. In this **statement**, we want to clear out Codey's inventory by **destroying** the heldItem and resetting the value of heldItemName to an empty **string**. Use the following **pseudocode** to help you write your **statements**.

- If Codey is touching the trigger with the Receivers name, and
  - If Codey is holding an item with the toastSlice name,
    - Then destroy the item that Codey is holding, and
    - Change the value of heldItemName to nothing.

```
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")...
        if (triggerName == "Stove")...
        if (triggerName == "Receivers")
        {
            if (heldItemName == "toastSlice")
            {
                Destroy(heldItem);
                heldItemName = "";
            }
        }
    }
}
```

**85** Hmmm... this **code** should seem very, very familiar! We used these exact two **lines** when we wanted to toast our bread slice. Since we are repeating code, we can create a new **function** and use it any time we want to perform these actions.

Outside of the Update **function**, create a **private void function** named PlaceHeldItem. The **body** of this **function** should **destroy** the held item and reset the held item's name to an empty string.

```
private void PlaceHeldItem()
{
    Destroy(heldItem);
    heldItemName = "";
}
```

**86** We can now **refactor** our **code**! This just means that we are reorganizing our **code** to make it more efficient. Wherever you used those two lines of **code** in your new **function's body**, we can instead **call the function**!

```
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")...
        if (triggerName == "Stove")
        {
            if (heldItemName == "breadSlice")
            {
                stove.ToastBread();
                PlaceHeldItem();
            }
            else
            {
                if (stove.cookedFood == "toast")...
            }
        }
        if (triggerName == "Receivers")
        {
            if (heldItemName == "toastSlice")
            {
                PlaceHeldItem();
            }
        }
    }
}
2 references
private void PlaceHeldItem()
{
    Destroy(heldItem);
    heldItemName = "";
}
```

**87** Playtest your game to make sure we didn't break anything. Pick up bread and try to place it at the Receivers – nothing should happen because we are looking for toast!

 **Sensei Stop**

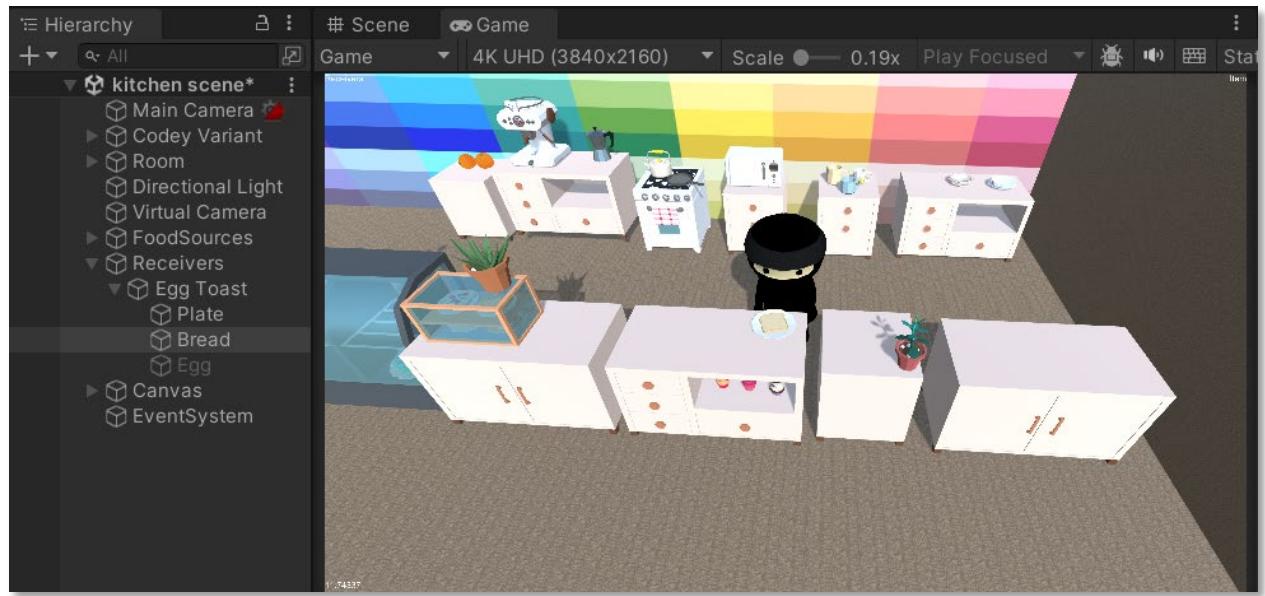
There's one other place that we have repeated code – we will create a function to help us soon! Talk with your Sensei about what code can be placed into a function.

**88** We now need to make sure that Codey's toast doesn't disappear, and instead appears on the plate. We need to use the **GameObject class**'s **Find function** to search our **hierarchy** for the toastSlice **object** and set it **active**. It sounds like a lot, but it can be done in one line.

```
if (triggerName == "Receivers")
{
    if (heldItemName == "toastSlice")
    {
        PlaceHeldItem();
        GameObject.Find("Receivers/French Toast/toastSlice").SetActive(true);
    }
}
```

The slashes show that toastSlice belongs to French Toast which belongs to Receivers. Save the **script**.

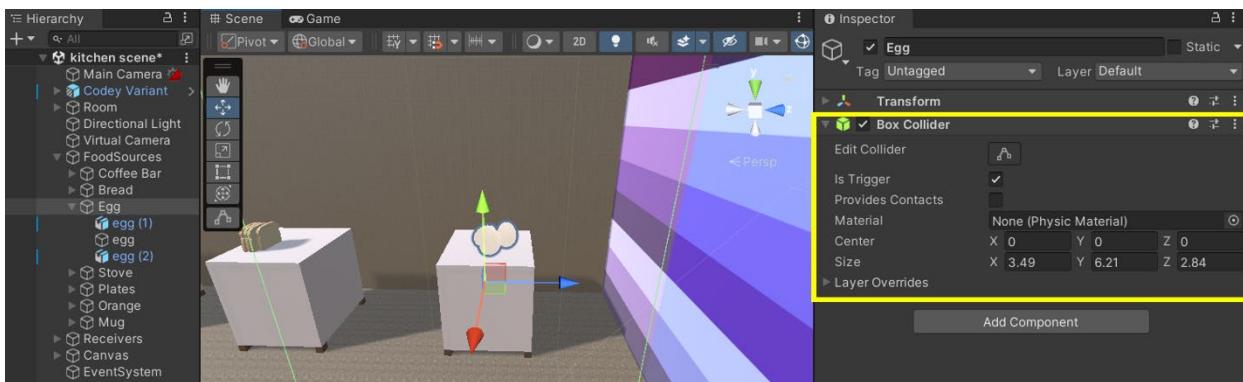
**89** Playtest your game. Pressing the spacebar while Codey is holding the toastSlice at the Receivers will now put a piece of toast on the plate.



## How to Program an Egg

**90** We want to make sure we have a finished dish. We can repeat a lot of the same steps with eggs! Use the following checklist to help you complete all of the tasks to get a second interactable object working.

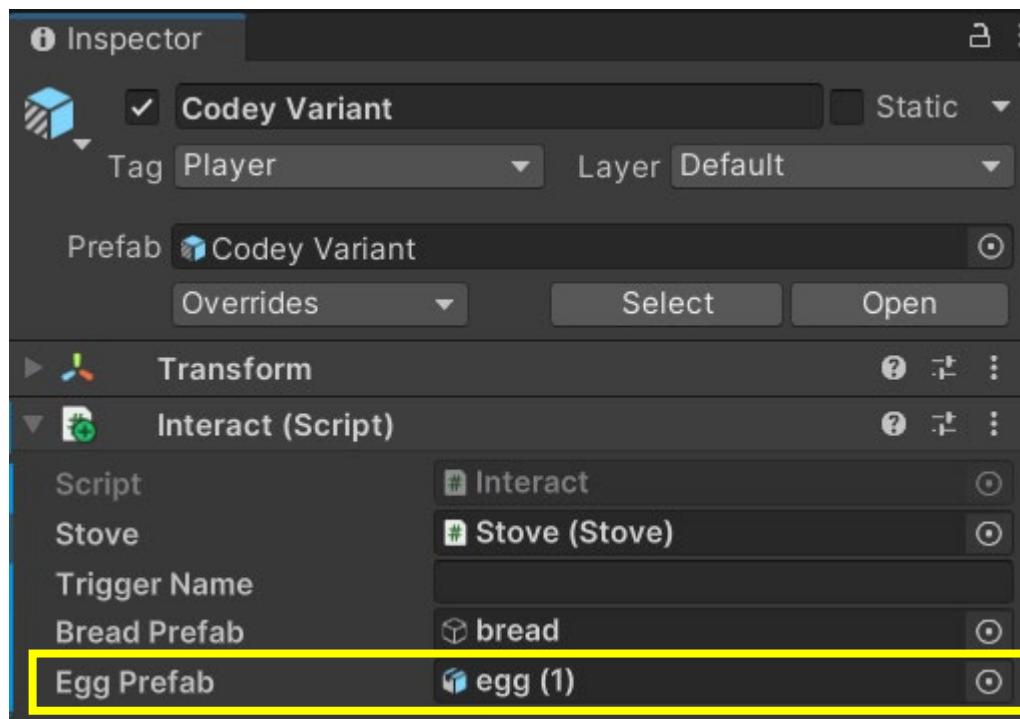
- Design an egg source location in your scene.
- Add an empty game object to the Sources object and name it.
- Search the asset store to find an egg object or make your own out of a capsule or sphere.
- Place your objects inside your new source object.
- Add a box collider to the new source object.
- Enable “Is Trigger” on the box collider.
- Change the position and size so Codey can reach it.
- Playtest your game and make sure Codey’s “Trigger Name” property updates when inside the collider.



**91** Just like the bread, we need to tell Codey what **prefab** to use when we want to pick up an egg. In the **Interact script**, add a **public game object** named eggPrefab.

```
public GameObject breadPrefab;  
public GameObject eggPrefab;
```

**92** Save your **script**. In the Unity **inspector**, assign the **value** of eggPrefab to one of the egg **game objects**.



**93** We can now use the same code that we used to pick up the bread and the toast to pick up our egg.

Create a new **private void function** named PickUpItem. Copy and paste the three lines that **instantiate** the breadPrefab, set the heldItem's **position**, and changes the **value** of heldItemName into the **body** of the new **function**.

```
0 references
private void PickUpItem()
{
    heldItem = Instantiate(breadPrefab, transform, false);
    heldItem.transform.localPosition = new Vector3(0, 2, 2);
    heldItemName = "breadSlice";
}
```

**94** We need to modify these three lines of Code to work for any potential item we want Codey to pick up.

 **Sensei Stop**

Discuss with your Sensei what two elements of the code determine the held item's model and name. What special feature can we add to a function that lets us change its inputs?

**95** We need to change breadPrefab and "breadSlice" into generic **parameters**. Change the declaration of the **function** to require one **GameObject** and one **string as parameters** and use these two **parameters** in the **body** of the **function**.

```
0 references
private void PickUpItem(GameObject itemPrefab, string itemName)
{
    heldItem = Instantiate(itemPrefab, transform, false);
    heldItem.transform.localPosition = new Vector3(0, 2, 2);
    heldItemName = itemName;
}
```

**96** We can now use this **function** in our Update **conditional statements**. This **function** also means that Codey can now pick up any new item that you've created as a **source**. You just call this **function** with a **prefab** and an item name, and Codey will pick up the **object**.

```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")
        {
            PickUpItem(breadPrefab, "breadSlice");
        }

        if (triggerName == "Stove")
        {
            if (heldItemName == "breadSlice")
            {
                stove.ToastBread();
                PlaceHeldItem();
            }
            else
            {
                if (stove.cookedFood == "toast")
                {
                    PickUpItem(breadPrefab, "toastSlice");
                    stove.CleanStove();
                }
            }
        }

        if (triggerName == "Receivers")
        {
            if (heldItemName == "toastSlice")
            {
                PlaceHeldItem();
                GameObject.Find("Receivers/French Toast/toastSlice").SetActive(true);
            }
        }
    }
}

2 references
private void PickUpItem(GameObject itemPrefab, string itemName)
{
    heldItem = Instantiate(itemPrefab, transform, false);
    heldItem.transform.localPosition = new Vector3(0, 2, 2);
    heldItemName = itemName;
}
```

---

**97** Use this new function in a new if statement that checks to see if Codey is standing in the Egg trigger.

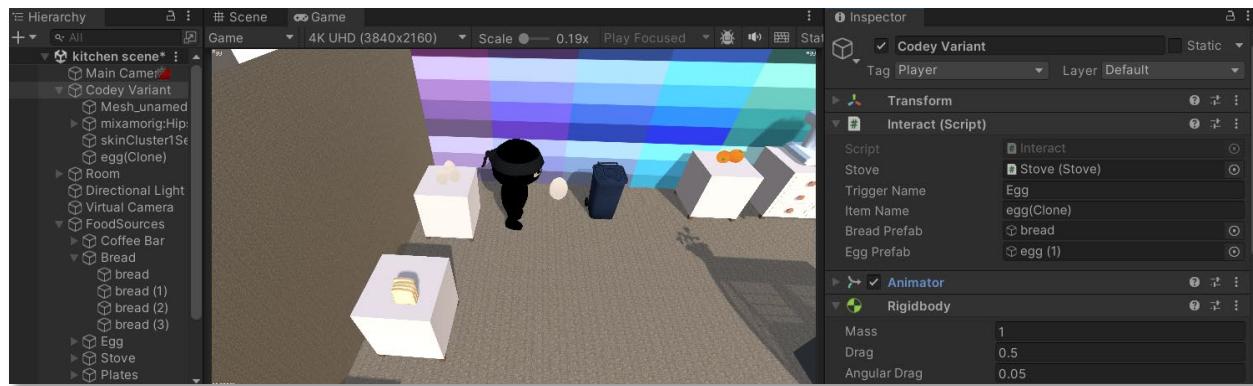
```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")
        {
            PickUpItem(breadPrefab, "breadSlice");
        }

        if (triggerName == "Egg")
        {
            PickUpItem(eggPrefab, "egg");
        }
    }

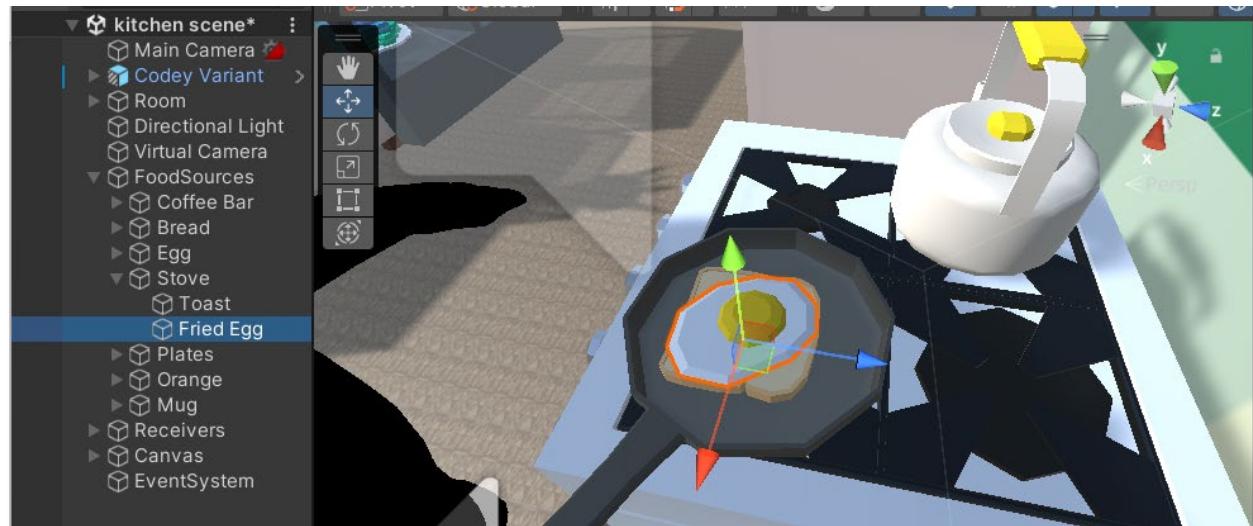
    if (triggerName == "Stove")
    {
        if (heldItemName == "breadSlice")
        {
            stove.ToastBread();
            PlaceHeldItem();
        }
        else
        {
            if (stove.cookedFood == "toast")
            {
                PickUpItem(breadPrefab, "toastSlice");
                stove.CleanStove();
            }
        }
    }

    if (triggerName == "Receivers")
    {
        if (heldItemName == "toastSlice")
        {
            PlaceHeldItem();
            GameObject.Find("Receivers/French Toast/toastSlice").SetActive(true);
        }
    }
}
```

**98** Save your **script** and playtest your game. What happens when you press the spacebar when Codey is in front of the eggs?



**99** Codey is holding an egg! We now need to program the logic so the stove can fry it for us. Place a model for the fried egg inside our Stove Source object. Position it so it is inside the frying pan with the toast.



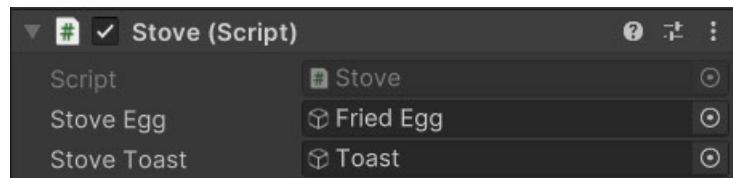
**100** In the Stove **script**, create a **public GameObject** named friedEgg.

```
public class Stove : MonoBehaviour
{
    public GameObject toast;
    public GameObject friedEgg;

    public string cookedFood = "";
```

**101**

Save the **script** and attach the **variable** to the **game object** in the **inspector**. No matter what type of **object** you used, make sure that you drag the **object** that is on our **source** to the **script**!



**102**

Open the Stove **script**. Just like the toast, we need to **disable** the egg **object** on **Start** and create a **public function** that fries the egg so Codey can cook.

### Sensei Stop

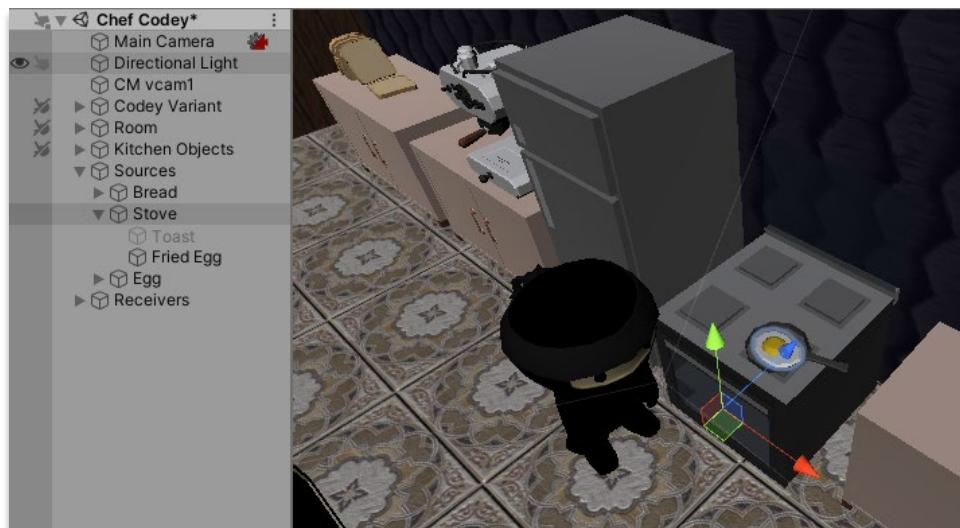
Look back at how we first disabled the toast and then wrote a public function for Codey to use. Can you do the same thing for the fried egg? First, disable the fried egg object when the script starts. Create a public void function to fry the egg that sets the object to active and updates the value of cookedFood to "friedEgg".

**103** Back in the Interact script, we need to write the **logic** that cooks the egg if Codey is near the stove and holding an egg. Find the section of the code where Codey is **triggering** the stove. Add an else if statement that checks to see if the value of heldItemName is "egg" and then asks the stove to fry the egg and places the item being held.

```
if (triggerName == "Stove")
{
    if (heldItemName == "breadSlice")
    {
        stove.ToastBread();
        PlaceHeldItem();
    }
    else if (heldItemName == "egg")
    {
        stove.FryEgg();
        PlaceHeldItem();
    }
    else
    {
        if (stove.cookedFood == "toast")
        {
            PickUpItem(breadPrefab, "toastSlice");
            stove.CleanStove();
        }
    }
}
```

Pay close attention to where your if, else if, and else statements are! Use brackets and indentations to help you put code in the right place.

**104** Playtest your game and make sure that Codey can take an egg and crack it onto the stove.



**105** Now we can **program** Codey to pick up the fried egg from the stove. We can again use the **functions** we created to make this step very easy!

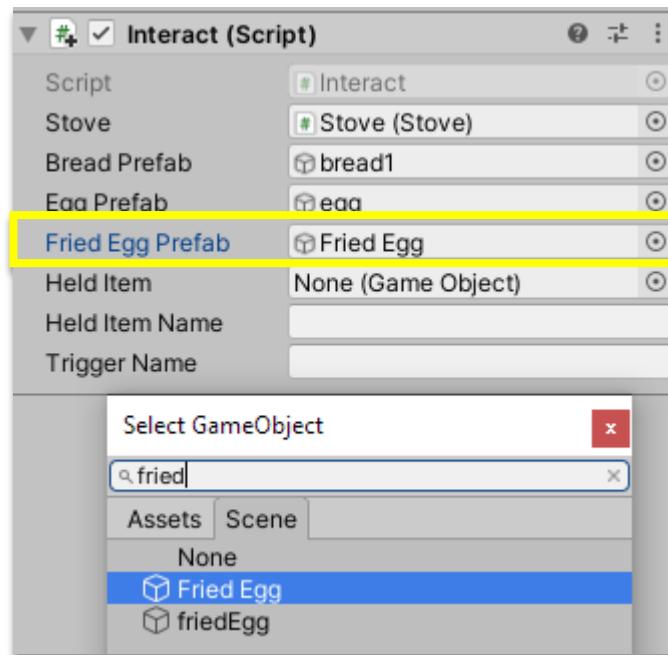
In the **Interact script**, add a new **public GameObject** variable named FriedEgg and attach a fried egg **model** to the **variable** in the **inspector**.

```
public class Interact : MonoBehaviour
{
    public Stove stove;

    public string triggerName = "";

    public GameObject breadPrefab;
    public GameObject eggPrefab;
    public GameObject friedEggPrefab; highlighted

    public GameObject heldItem;
    public string heldItemName;
```



**106** Back in the Interact **script**, add the **logic** that asks the stove to pick up food and runs the PickUpItem **function** with the fried egg **prefab** and "friedEgg" as the item name.

```
if (triggerName == "Stove")
{
    if (heldItemName == "breadSlice")
    {
        stove.ToastBread();
        PlaceHeldItem();
    }
    else if (heldItemName == "egg")
    {
        stove.FryEgg();
        PlaceHeldItem();
    }
    else
    {
        if (stove.cookedFood == "toast")
        {
            PickUpItem(breadPrefab, "toastSlice");
            stove.CleanStove();
        }
        if (stove.cookedFood == "friedEgg")
        {
            PickUpItem(friedEggPrefab, "friedEgg");
            stove.CleanStove();
        }
    }
}
```

**107** Playtest your game to make sure Codey can pick up the fried egg.



 **Sensei Stop**

Oh no! There's still egg in the stove. Something must be wrong with our CleanStove function. Investigate and tell your Sensei what one line of code needs to be added to fix this bug.

**108** Playtest your game to make sure both the toast and the fried egg disappear from the stove.



## 109

We can finally finish our dish! Open the Interact script.

Look back at how you **coded** the toast slice. Add the **code** that **finds** and **enables** the fried egg **object** on our French Toast **receiver**. If Codey is standing inside the Receivers **trigger** and **if** Codey is holding the friedEgg **object**, **then** place Codey's held item, **find** the French Toast's fried egg **object**, and **activate** it.

```
if (triggerName == "Receivers")
{
    if (heldItemName == "toastSlice")
    {
        PlaceHeldItem();
        GameObject.Find("Receivers/French Toast/toastSlice").SetActive(true);
    }
    if (heldItemName == "friedEgg")
    {
        PlaceHeldItem();
        GameObject.Find("Receivers/French Toast/friedEgg").SetActive(true);
    }
}
```

## 110

Playtest your game and cook some French Toast! Make sure that none of your **trigger boxes** overlap. They should be far enough apart so Codey cannot stand in two at the same time.



## Time to Sizzle

**111** The core of our game is complete! We will now cover some visual touches that help communicate mechanics to the player. In the Prove Yourself you will add additional objects for Codey to make.

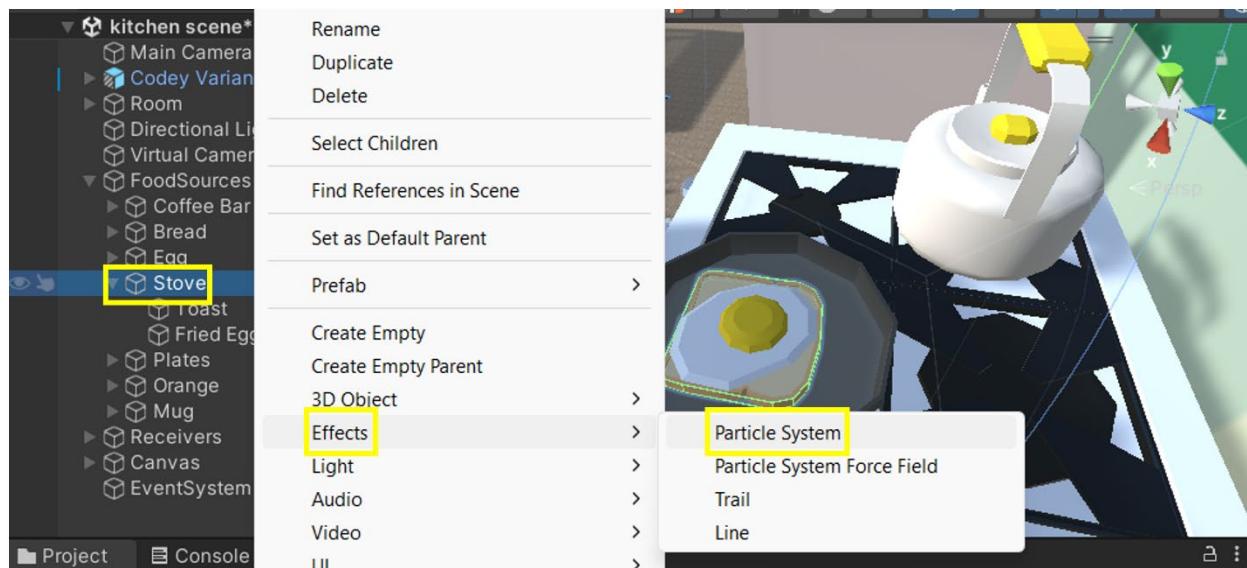
Right now, the stove toasts the bread and fries the egg instantly. We can add some effects to make it seem like it's actually cooking our items.



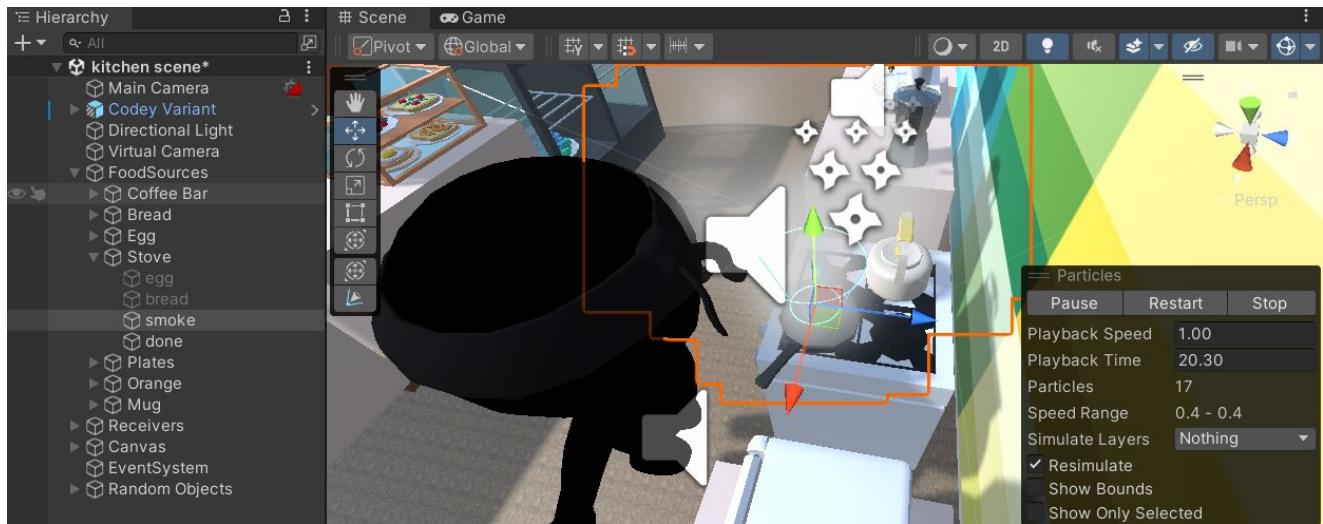
### Ninja Planning Document

Take at least 5 but no more than 10 minutes and complete your Ninja Planning Document – Communicating to the Player

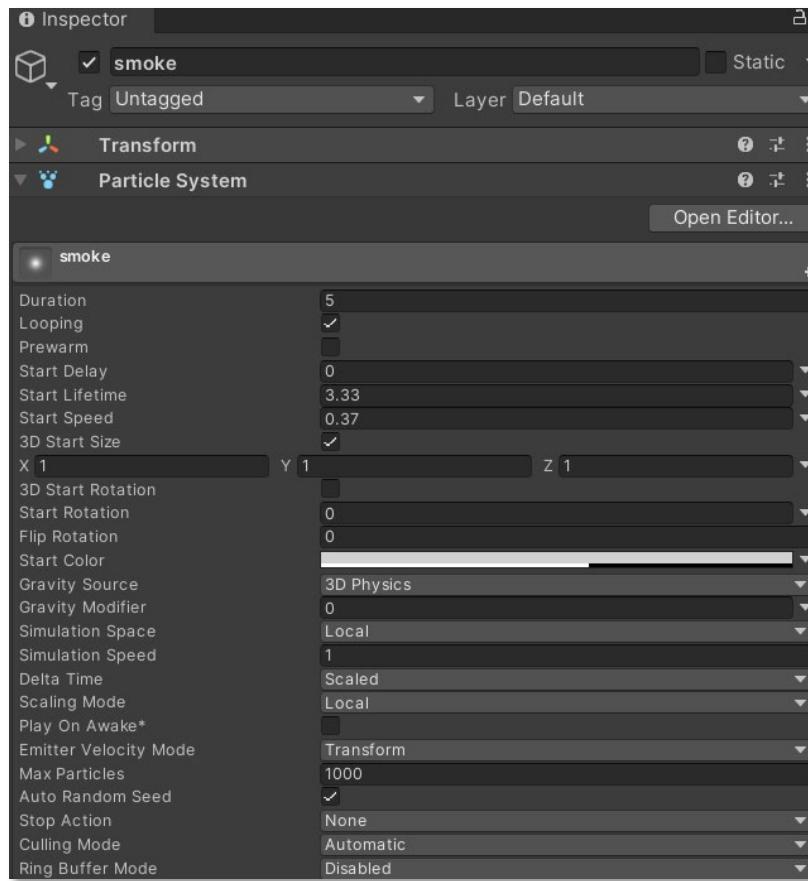
**112** We can use the Unity particle system to make the stove cook our food. Add a **particle system object** to the Stove Source **object**.



**113** Rename your particle system to “smoke” and position it on the frying pan.

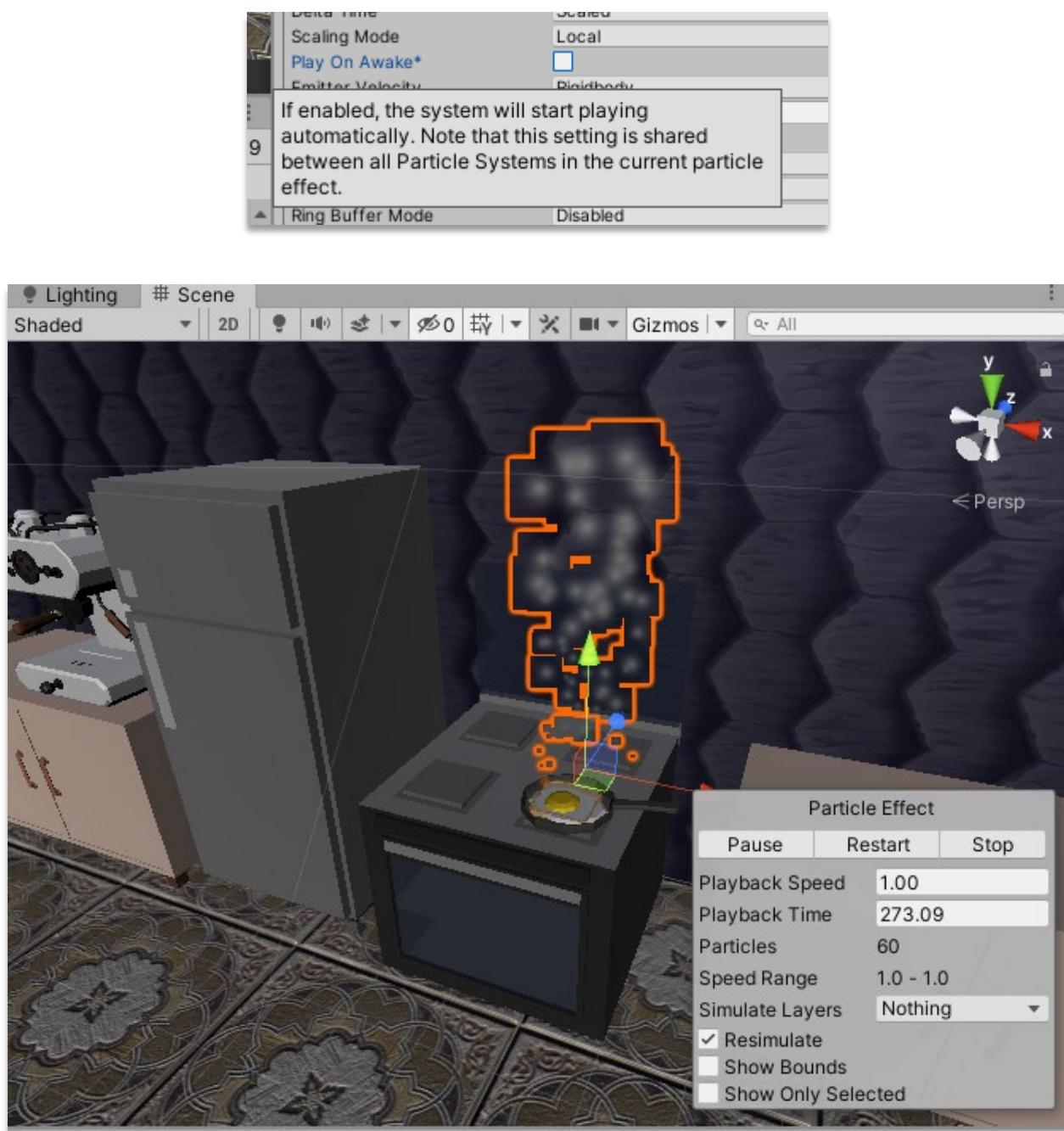


**114** Experiment with all of the options in the **Particle System component** to create the effect you designed in your Ninja Planning Document.



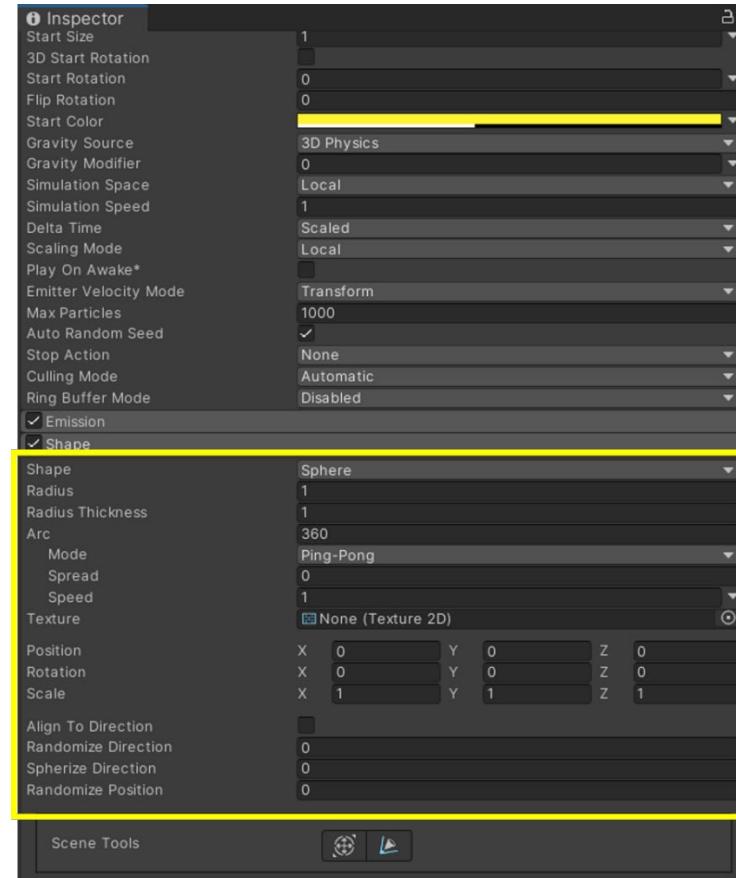
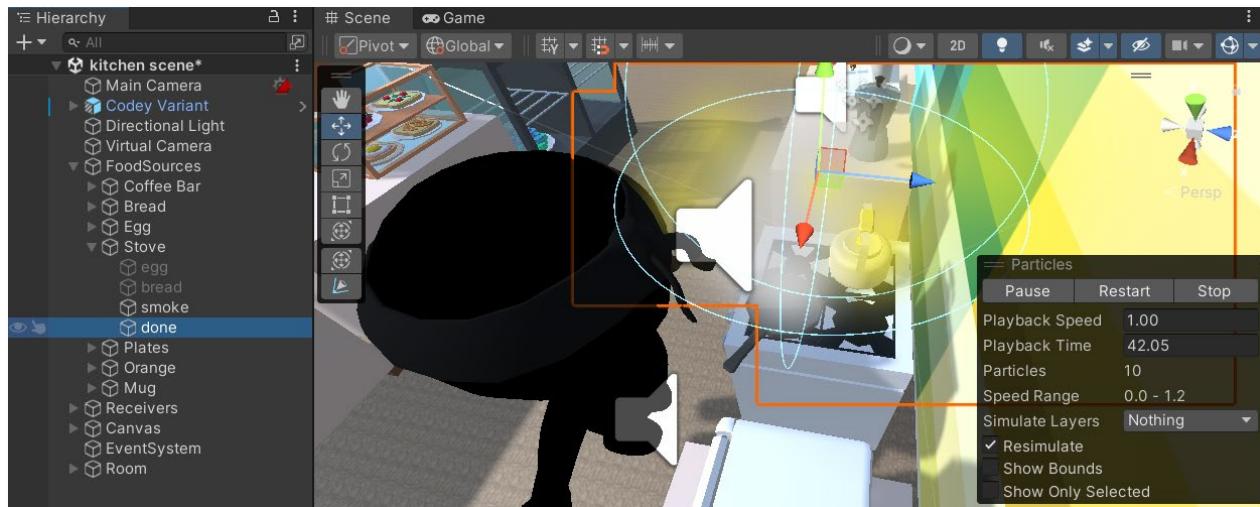
115

Don't be afraid to edit any of these **values**! The only one that you need to be sure is to **disable** is **Play On Awake**. If enabled, **Play On Awake** tells the **particle system** to play as soon as the game starts. Since we want Codey to control when the stove starts cooking, we need to **disable** the option.



**116** Once you are satisfied with your first **particle system**, create a second **particle effect system** named “complete” that will indicate that cooking has finished.

Experiment with changing the **texture** located in the Shape properties.



**117** We can now move to the Stove **script** and **program** the **particle effects** to play and stop based on some simple cooking **logic**.

Create two **public ParticleSystem variables** named smoke and complete.

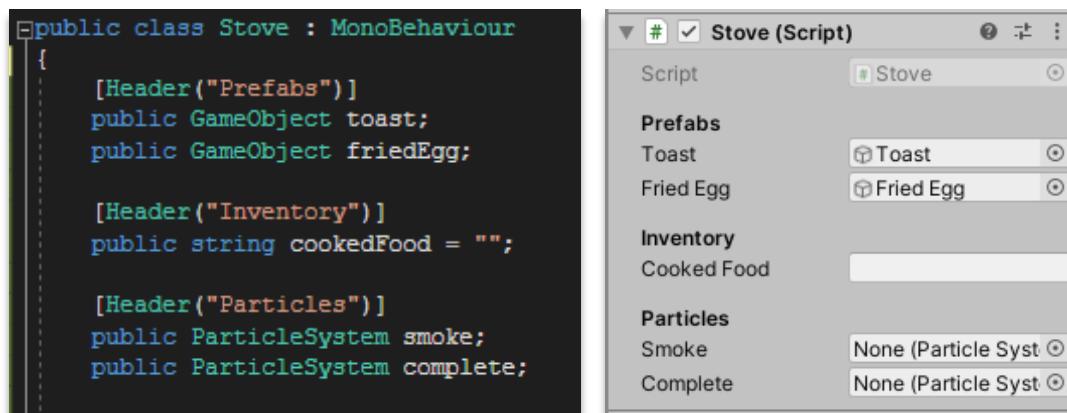
```
public class Stove : MonoBehaviour
{
    public GameObject toast;
    public GameObject friedEgg;

    public string cookedFood = "";

    public ParticleSystem smoke;
    public ParticleSystem complete;
```

**118** We now have a few **scripts** with several **public variables**. We can add a special line of code that helps us organize these **variables**. Add [Header("Prefabs")], [Header("Inventory ")], and [Header("Particles")] above the appropriate **variables**.

You can see what these headers did in the Unity inspector.



**119** Assign the smoke and complete **particle systems** to the Stove **script**.



**120** Inside our two cooking **functions**, tell the smoke **particle system** to play.

```
1 reference
public void FryEgg()
{
    smoke.Play();
    friedEgg.SetActive(true);
    cookedFood = "friedEgg";
}

1 reference
public void ToastBread()
{
    smoke.Play();
    toast.SetActive(true);
    cookedFood = "toast";
}
```

**121** Playtest your game and see what happens when you try to make toast or fry an egg.



**122** We now need to **program** our stove to take time to cook. We can accomplish this by **Invoking** a **function** after a number of seconds.

In the Stove **script**, create a **private void function** named CompleteCooking. This **function** should **stop** the **smoke particle effect** and **play** the **complete particle effect**.

```
0 references
private void CompleteCooking()
{
    smoke.Stop();
    complete.Play();
}
```

**123** In our two cooking **functions**, use Unity's **Invoke function** to call CompleteCooking after a few seconds.

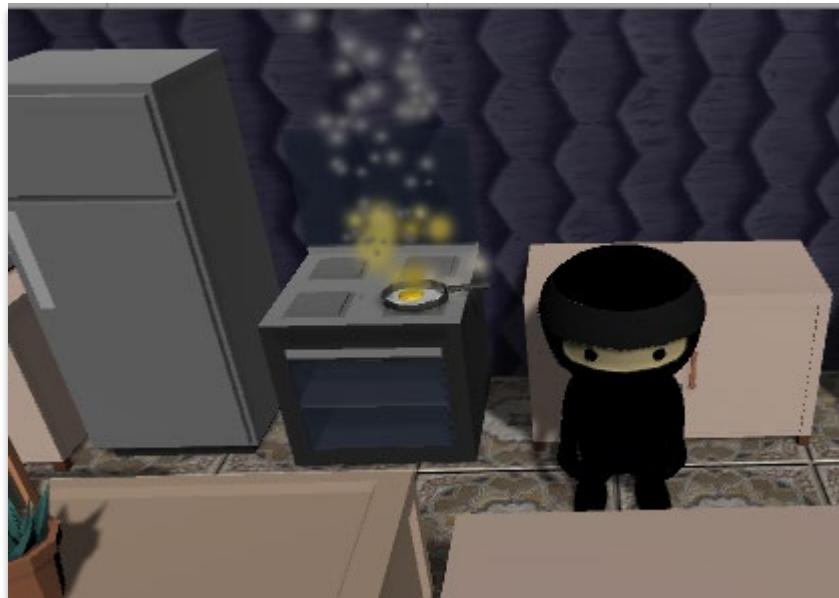
```
1 reference
public void FryEgg()
{
    smoke.Play();
    friedEgg.SetActive(true);
    cookedFood = "friedEgg";
    Invoke("CompleteCooking", 8f);
}
1 reference
public void ToastBread()
{
    smoke.Play();
    toast.SetActive(true);
    cookedFood = "toast";
    Invoke("CompleteCooking", 6f);
}
```



### Invoke

The **Invoke** function takes two parameters. The first parameter is a string of the function name that you want to run. The second parameter is the time you want to wait until you run the function. The second parameter has an "f" after the number because it needs to be a float, a special type of number that can use decimals.

**124** Now playtest your game and wait to see if your **particles** start and stop properly.



**125** We now need to stop the complete **particle effect** when Codey picks up the food in the CleanStove **function**.

```
2 references
public void CleanStove()
{
    toast.SetActive(false);
    friedEgg.SetActive(false);
    cookedFood = "";
    complete.Stop();
}
```

**126** Playtest your game and make sure the **particles** stop after Codey picks up the food.

### Sensei Stop

What happens when you try to pick up food before it's complete?  
Does anything stop Codey from picking up food immediately?  
Discuss with your Sensei how you could fix this bug.

Think about the following questions to help you think about a solution.

- Do you need to create a new **variable**?
- Should this **variable** be **public** or **private**? Why?
- What type of **variable** can be used to answer the question "is the stove cooking, yes or no?"
- What **functions** need to update the **value** of this **variable**?

**127** Right now, Codey can always interact with the stove. We need Codey to know that the stove is in use so you can't pick up food too early or cook more than one item at a time.

Add a **public bool variable** named `isCooking` to the Stove **script**. Initialize it to **false**.

```
public class Stove : MonoBehaviour
{
    [Header("Prefabs")]
    public GameObject toast;
    public GameObject friedEgg;

    [Header("Inventory")]
    public string cookedFood = "";
    public bool isCooking = false;

    [Header("Particles")]
    public ParticleSystem smoke;
    public ParticleSystem complete;
```

**128** We need to toggle this **variable** to **true** whenever the stove is cooking and toggle it to **false** whenever it is done.

```
1 reference
public void FryEgg()
{
    isCooking = true;
    smoke.Play();
    friedEgg.SetActive(true);
    cookedFood = "friedEgg";
    Invoke("CompleteCooking", 8f);
}
1 reference
public void ToastBread()
{
    isCooking = true;
    smoke.Play();
    toast.SetActive(true);
    cookedFood = "toast";
    Invoke("CompleteCooking", 6f);
}
```

```
0 references
private void CompleteCooking()
{
    isCooking = false;
    smoke.Stop();
    complete.Play();
}
```

**129** Now that the stove can tell Codey when it is cooking, we need to program the logic to prevent Codey from interacting with a cooking stove.



### Sensei Stop

Talk with your Sensei about how you can accomplish this. Before you add any code to your scripts, write a few sentences of pseudocode that describe when Codey can and cannot interact with the stove.

Implement your code and playtest your game.

## Project Submission and Reflection

Now that you have a working game, take time to add your own personal touches to the project. What aspects from other games could you add?

Once you feel like you have a good product that represents your vision of the game, have a Sensei and at least one other Ninja playtest it. Use the Playtest Survey Planning Document for questions to ask them when they finish. Record their answers in your Ninja Planning Document.

Based on the results of the playtest and survey, make changes to your game. Once you are complete, share the updates with your Sensei and fill out the reflection section of your Ninja Planning Document.

Before you submit your game for grading, use the Chef Codey Project Requirements Checklist to make sure your game has all of the required features.



### Ninja Planning Document

Use your Ninja Planning Document to record feedback from Senseis and other Ninjas in your Center.